JUSTUS-LIEBIG-
UNIVERSITÄT
GIESSEN

Engineering complex mathematical models in systems biology with Modelica using the example of the human cardiovascular system

A Dissertation

Submitted to the department of biology and chemistry of the Justus Liebig University Giessen in partial fulfillment of the requirements for the degree of

Doctor rerum naturalium

(Dr. rer. nat.)

by

Christopher Schölzel

July 4, 2022

Referees:

Prof. Dr. Alexander Goesmann

Prof. Dr. Christoph Müller

Prof. Dr. Andreas Dominik

I declare that I have completed this dissertation single-handedly without the unauthorized help of a second party and only with the assistance acknowledged therein. I have appropriately acknowledged and cited all text passages that are derived verbatim from or are based on the content of published work of others, and all information relating to verbal communications. I consent to the use of an anti-plagiarism software to check my thesis. I have abided by the principles of good scientific conduct laid down in the charter of the Justus Liebig University Giessen "Satzung der Justus-Liebig-Universität Gießen zur Sicherung guter wissenschaftlicher Praxis" in carrying out the investigations described in the dissertation.

Signature

Abstract

Biological systems are complex and full of interconnected feedback loops, which require going beyond reductionist endeavors to map the genome, transcriptome, and proteome and consider the whole system instead. This is the goal of systems biology, and it often involves the integration of multiple descriptions of biological systems at different scales of time and space. Since predictions about such complex systems are hard to make, mathematical simulations are used to quantitatively assess the phenomena under study. However, most mathematical models of biological systems are unfit for the sort of hierarchical composition required for this task both due to their structure and due to the programming or modeling language used. In engineering, systems of much larger size and similar complexity have been successfully modeled using the language Modelica, which is largely unknown in systems biology. This dissertation therefore asks if Modelica can be used to tackle the challenges of multi-scale modeling in systems biology. In place of the vast amount of biological models available, the dissertation focuses on models of the cardiovascular system, since this is an active and relevant field of research that showcases a lot of the typical complexity of biological systems.

To assess the benefits of Modelica for systems biology, I first establish a set of requirements for modeling languages in systems biology in general by examining the properties of a subsystem in detail. I assess whether Modelica fulfills these requirements using models of the human baroreflex, the Hodgkin-Huxley model of the squid giant axon, and a one-dimensional model of the human atrioventricular node. As there are other languages that aim to solve similar issues, I then contrast their abilities with those of Modelica. This bridges to a broader investigation of the benefit of software engineering techniques in general, such as object orientation, structured documentation, or unit testing. Finally, I discuss and provide some improvements for the usability of Modelica in a biological context.

The results of this dissertation indicate that Modeling languages used for systems biology should be modular, declarative, human-readable, open, graphical, and hybrid. From all investigated language candidates, Modelica fulfills these requirements to the fullest extent. Using other languages is possible, but brings drawbacks either in modularity, openness, or the graphical representation of models. However, SBML and CellML, which are recommended standard languages in systems biology, have the clear benefit of including domain-specific features such as semantic annotation using ontologies, and they also benefit from a high

acceptance and interoperability with other tools in the community. Regardless of the concrete language, software engineering techniques should be applied to mathematical modeling similar to other pieces of software. Among other benefits, this could actually guarantee that the methods of a simulation study are reproducible. In the case of Modelica, this means that the language has to fit better into a typical software engineering workflow, which can be achieved by separate tools for code editing, vector graphics editing, and structured documentation, which are provided as part of this dissertation. At the bottom line, Modelica is not the perfect solution to every problem of systems biology, but at the very least it is a great source of inspiration that should either be used as the basis of or be partly incorporated into future languages.

Zusammenfassung

Biologische Systeme sind komplex und voller miteinander verbundener Rückkopplungsschleifen, die es nötig machen, über reduktionistische Bemühungen zur Kartierung des Genoms, Transkriptoms und Proteoms hinauszugehen und stattdessen das gesamte System zu betrachten. Das ist das Ziel der Systembiologie und erfordert oft die Integration mehrerer Beschreibungen biologischer Systeme auf unterschiedlichen Zeit- und Raumskalen. Da Vorhersagen über solch komplexe Systeme schwer zu treffen sind, werden mathematische Simulationen verwendet, um die studierten Phänomene quantitativ zu bewerten. Die meisten mathematischen Modelle biologischer Systeme sind jedoch sowohl aufgrund ihrer Struktur als auch aufgrund der verwendeten Programmier- oder Modellierungssprache nicht für die hierarchische Zusammensetzung, die für diese Aufgabe erforderlich ist, geeignet. In den Ingenieurwissenschaften wurden viel größere und ähnlich komplexe Systeme erfolgreich mit der Sprache Modelica modelliert, die in der Systembiologie noch weitgehend unbekannt ist. Diese Dissertation stellt daher die Frage, ob Modelica verwendet werden kann, um die Herausforderungen der Multiskalenmodellierung in der Systembiologie anzugehen. Stellvertretend für die große Menge an verfügbaren biologischen Modellen konzentriert sich die Dissertation auf Modelle des Herz-Kreislauf-Systems, da dies ein aktives und relevantes Forschungsgebiet ist sowie ein gutes Beispiel für die typische Komplexität biologischer Systeme.

Um den Nutzen von Modelica für die Systembiologie zu beurteilen, stelle ich zunächst eine Reihe von Anforderungen für Modellierungssprachen in der Systembiologie im Allgemeinen auf, indem ich die Eigenschaften eines Subsystems im Detail untersuche. Ob Modelica diese Anforderungen erfüllt, bewerte ich anhand von Modellen des menschlichen Baroreflexes, des Hodgkin-Huxley-Modells des Tintenfisch-Riesenaxons und eines eindimensionalen Modells des menschlichen atrioventrikulären Knotens. Da es auch andere Sprachen gibt, die darauf abzielen, ähnliche Probleme zu lösen, stelle ich dann ihre Fähigkeiten denen von Modelica gegenüber. Dies leitet über zu einer breiteren Untersuchung des Nutzens von Softwareentwicklungstechniken, wie z. B. Objektorientierung, strukturierte Dokumentation oder Komponententests, im Allgemeinen. Abschließend erarbeite und diskutiere ich einige Verbesserungen für die Verwendbarkeit von Modelica in einem biologischen Kontext.

Die Ergebnisse dieser Dissertation zeigen, dass in der Systembiologie verwendete Modellierungssprachen modular, deklarativ, menschenlesbar, offen, grafisch und hybrid sein sollten. Von allen untersuchten Sprachkandidaten erfüllt Modelica diese Anforderungen im größten

Maße. Die Verwendung anderer Sprachen ist möglich, bringt jedoch Nachteile entweder in der Modularität, Offenheit oder der grafischen Darstellung von Modellen mit sich. SBML und CellML, die in der Systembiologie empfohlene Standardsprachen sind, haben jedoch den klaren Vorteil, dass sie domänenspezifische Funktionen wie semantische Annotation mithilfe von Ontologien erlauben, und sie profitieren auch von einer hohen Akzeptanz und Interoperabilität mit anderen Tools in der Community. Unabhängig von der konkreten Sprache sollten Softwareentwicklungstechniken auf die mathematische Modellierung ähnlich wie bei anderen Softwarekomponenten angewendet werden. Damit könnte unter anderem die Reproduzierbarkeit der Methoden einer Simulationsstudie garantiert werden. Im Fall von Modelica bedeutet dies, dass sich die Sprache besser in einen typischen Software-Engineering-Workflow einfügen muss, was durch separate Tools für die Codebearbeitung, die Bearbeitung von Vektorgrafiken und die strukturierte Dokumentation erreicht werden kann, die im Rahmen dieser Dissertation bereitgestellt werden. Unter dem Strich ist Modelica nicht die perfekte Lösung für alle Probleme der Systembiologie, aber zumindest eine gute Inspiration, die entweder als Grundlage für zukünftige Sprachen verwendet oder teilweise in sie einfließen sollte.

Contents

1. Introduction

The subject of biological study is life itself, which poses immense opportunities and challenges for scientific research. The recent pandemic of the Coronavirus disease 2019 (COVID-19) caused by severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2) has shown that scientific progress in medical and biological questions is not easy to achieve, even when scientists across the whole world work together towards a common goal. It has also shown that there are biological questions that can only be answered by computer simulations.

In the case of the COVID-19 pandemic it was simply not possible to conduct experiments that would help to predict the spread of the disease because of the required scale, the urgency with which results were required, and of course the ethical aspect that one cannot simply infect a human being with a potentially deadly disease. Fortunately, epidemiologists could use mathematical modeling to calculate possible trajectories of the number of infections and hospitalizations given different intervention scenarios and to provide scientific evidence as basis for political decisions (Ndaïrou et al. 2020; Tuan, Mohammadi, and Rezapour 2020; Mandal et al. 2020).

These models can also be used in other areas of biology. At the smallest level, they can, for example, help explain the binding mechanism of the main active agent in the COVID-19 drug Paxlovid developed by Pfizer (Ahmad et al. 2021). Similar studies might inform drug discovery to reduce cost, speed up development, and improve success-rates, which are estimated at 4.3% for patent applications that actually lead to a product reaching the market (Pammolli, Magazzini, and Riccaboni 2011).

While there are already many successful modeling studies, the example of COVID-19 also shows that to understand and treat a disease like COVID-19, it is not enough to just look at the virus in a cell culture in a Petri dish or just at the population dynamics. Instead, the interplay between the virus, the cells that it infects, the function of the lung and respiratory tract, and the behavior of the human patient in the population has to be considered across all levels of biological organization. For example, at the organ level, SARS-CoV-2 can not only affect the lung, but also the heart and the kidney (Bader et al. 2021). To understand how and why the virus can enter cells of different organs, one has to look at the angiotensin-converting enzyme 2 (ACE2) receptor, a small protein in the cell membrane (Scialo et al. 2020). As a

second example, coronavirus variants are identified by mutations at the genetic level, but to determine whether they are variants of concern, the reproduction rates in the upper respiratory tract have to be investigated (Tao et al. 2021; Hou et al. 2020).

This calls for larger, more complex models, which incorporate effects at more than one level of organization and across all relevant areas of human physiology (Bardini et al. 2017). To build such models, it is imperative that tools are used that can manage the inherent complexity and that allow to combine existing models for all involved biological entities. In the engineering domain, the modeling language Modelica is an established solution precisely for integrating large, complex models across many engineering domains (Elmqvist 1978). Up until now, it is largely unknown in biology.

1.1. Systems Biology

Why is it that drug development takes 13.9 years on average with an average success rate of only 4.3% (Pammolli, Magazzini, and Riccaboni 2011)? Why is it not enough to sequence the full genome of a virus such as SARS-CoV-2 (Chiara et al. 2021) to figure out an effective treatment of the disease it causes? There seems to be a fundamental difference between engineering, where taking apart a machine or studying its blueprint immediately reveals its function, and biology, where analysis of living organisms proves to be more difficult.

The most important difference between biology and engineering is that the objects under study are the result of evolution—a random process guided by billions of years of ever-changing environmental influences (Darwin 1872; Palsson 2000). There is no designer's intent or small set of well-understood forces that would allow to easily identify the function of a component and to predict the results of a small change (Green 2015; Voit 2018). For example, the light does not directly fall on the photosensitive cells in the vertebrate eye, but must first pass through the optical nerve fibers, which is also the reason why we have a blind spot in our vision (Serb and Eernisse 2008). From a design standpoint this is an obvious flaw, but evolution does not find the global optimum (i.e. the "best" solution), but only performs small steps to improve the adaptation of an organism to the environment based on its existing traits, thus moving towards local optima but never taking "intermediate" steps that would lead to a temporary disadvantage (Dawkins 1996). Our nerve cells are another example, since they require dozens of different types of ion channels to fire an electrical signal that essentially serves a similar purpose as a switch being flipped on and off in a short period of time (Bean 2007). In general, evolved systems are therefore more complex than their designed counterparts.

In the past, researchers tackled this immense complexity mainly by a reductionist approach, investigating and cataloging the parts of organisms to understand the whole (D. Noble 2002). This has led to important discoveries such as deoxyribonucleic acid (DNA) as the basic building block of life (Franklin and Gosling 1953; Watson and F. H. C. Crick 1953) and a vast amount of knowledge about biochemical pathways that govern cell metabolism (Michal and Schomburg 2012). The hope that the focus on small parts of an organism can lead to an understanding of the behavior and traits of the whole organism is founded in the so-called "central dogma" of molecular biology (F. H. Crick 1958). In short, it states that there is a one-way flow of sequential information from DNA to proteins. Proteins, which are strands of amino acids that fold into macromolecules that govern cell functions, cannot transfer this information back to the genetic level.

This view is appealing, because it promises a simple way to understand living organisms in a bottom-up approach, but there are some processes that challenge it (here and in the following see A. D. Goldberg, Allis, and Bernstein 2007). The most important of these processes are methylation, which is the adding or removing of methyl groups to a DNA strand, and histone modification, which is an alteration in the proteins that are the "spools" around which DNA is wound inside the cell nucleus. Both methylation and histone modification are protein-mediated processes that regulate whether a gene is expressed, i.e. whether the information contained in it can be read by the cell. Gene expression can also be inherited along with the DNA itself. This constitutes a backwards flow of information from proteins to DNA. The flow is not symmetric, since the genetic code itself remains unchanged, but it does affect the resulting traits of the organism.

In general, there is no true one-way flow of information anywhere in biology due to an abundance of feedback loops across all organizational levels (Ferrell 2013). A feedback loop arises when changes in one physiological quantity like blood pressure, hormone levels, or gene expression levels, ultimately results in a further modification of the same quantity after an arbitrary number of intermediary steps. If raising the quantity leads to a further increase or lowering it leads to a further decrease, this is called a positive feedback loop. Positive feedback loops allow rapid responses to small changes in incoming signals, and they can act as irreversible switches (Ferrell 2013): For example, the voltage-gated sodium channels in a nerve cell start to open when the membrane voltage exceeds a certain threshold, resulting in an inflow of positively charged sodium ions, which increases the membrane voltage even further (Klabunde 2012). This positive feedback loop ensures that an action potential can be generated within less than a millisecond until it is stopped by a negative feedback loop.

In negative feedback loops, an increase of a quantity leads to a later decrease or vice versa. They ensure that the system stays within certain "healthy" boundaries by introducing oscillations as regulatory mechanism (Ferrell 2013). For example, a rise in blood pressure in the aorta of a human being leads to an activation of baroreceptors, which send a nerve signal to the autonomic nervous system, which in turn produces a chemical signal that decreases

the heart rate, thus decreasing the blood pressure in the aorta and closing the feedback loop (Klabunde 2012). If one blanks out the intermediate levels, an increase in blood pressure ultimately leads to a decrease at a later time—a self-regulating negative feedback loop. If the negative feedback is both strong and delayed, oscillations can also be more pronounced, which enables repetitive behavior such as the cell cycle (Ferrell 2013).

These feedback loops exist on all organizational levels from gene regulation to interactions between different organs or whole organisms, and they interact with each other (Ferrell 2013). As already mentioned, the positive feedback loop of the sodium channel is only stopped by another negative feedback loop that operates on the same signal with a small delay (Klabunde 2012). This occurs, for example, in nerve cells of the autonomic nervous system, which again play a part in the negative feedback loop of the aortic baroreceptors (Wehrwein and Joyner 2013).

These interconnected feedback loops can also introduce bifurcations, which are states where a small change in a single parameter can introduce qualitative changes in the observed behavior (Leite and Wang 2010). They act much like switches between different states (Ferrell 2013). Returning to the example of the autonomic nervous system, each nerve cell has a threshold potential (here and in the following see Martin et al. 2021). If the cell is stimulated by an external signal from another nerve cell, it is critically important if the stimulation potential is above or below the threshold. If it is lower, the cell will slowly return to its resting potential, but if it exceeds the threshold, the cell generates an action potential, which can then travel to other nerve cells and, for example, trigger a heart beat.

In general, any physiological quantity within an organism can influence almost any other quantity in that organism and even small quantitative errors in a prediction of one of these quantities may lead to a complete misjudgment of the overall behavior (Voit 2018). It is no surprise then, that the reductionist approach in its naïve form of studying isolated parts[1] has its limits in finding answers to biological questions (Kitano 2002): Whenever one studies a part of an organism in isolation, the results of such experiments may have only limited explanatory value of actual biological processes, as the perturbations introduced by the feedback loops in the live organism—*in vivo*—may lead to very different behavior (Voit 2018). This property of biological phenomena is called *emergence*. For example, the action potential is an emergent behavior of nerve cells that cannot be accurately predicted by only investigating individual ion channels or other exchange mechanisms (Martin et al. 2021). An example that can be observed in everyday life is the swarming behavior of fish, birds, or insects (Kelley

[1] Although the debate between molecular biology and systems biology is sometimes framed as a debate between reductionism and holism, the overloaded philosophical meaning of the terms can distract from the actual discussion (Gatherer 2010). I therefore only use the term reductionism and restrict it to this naïve form here, in order to explain the ideas behind systems biology without giving the impression that they constitute a rejection of reductionism per se.

and Ouellette 2013). Observing a single animal may yield some insight into its movement patterns, but the function of diverting and confusing predators only becomes apparent when observing the swarm as a whole.

To overcome this limitation of reductionism, one needs to shift the focus of experiments from single biological units like cells, proteins, or genes, to *networks* of these units including all major interactions between them (Voit 2018). In recent years, this has become tangible due to the advent of high-throughput technologies, allowing the accumulation of vast amounts of knowledge about the lowest levels of biological organization (Tyers and Mann 2003). The resulting branches of study are called genomics for information about genes in DNA, transcriptomics for knowledge about ribonucleic acid (RNA), proteomics for proteins, and so on, which has led to the use of the blanket term *omics* for all these data-intensive low-level disciplines (Karczewski and Snyder 2018). The eponymous suffix *omics* indicates that the object of study is the *totality* of the individual entities, highlighting the importance of the interplay between them (Yadav 2007).

The problem in the application of the knowledge generated by these omics studies is the size and complexity of the network between the individual units (Hammer et al. 2004). For example, describing the mechanisms involved in myocardial infarction purely by using omics data would be like describing the cause for a malfunctioning website in terms of electrical charges on transistors. In both cases, such an analysis is *possible* in theory, but not helpful in practice, if the goal is that a human should understand what is going on. In order to pose high-level biological questions, some or all parts of the biological networks therefore have to be replaced by higher-level abstractions. For the study of myocardial infraction this might be heart muscle cells and blood vessels, for which high-level rules can be formulated. These abstractions still have to be informed by the omics knowledge, but they can summarize that knowledge in a way that presents relevant information in a simplified way and neglects irrelevant information. Of course, the decision what is deemed relevant or irrelevant highly depends on the research question. Therefore, to represent all biology, descriptions of varying degrees of abstraction are required, leading to a hierarchical view of biology with several layers of organization from genes to proteins to cells and all the way up to whole ecosystems (Hammer et al. 2004). Borrowing from systems theory, such a hierarchical representation of a network of biological units is called a *biological system* (Wolkenhauer 2001).

The study of these biological systems is called *systems biology* (Kitano 2002). Due to the increased size and complexity of the objects of study in this field, it is often no longer possible to perform qualitative analysis and draw conclusions by human intuition (D. Noble 2002). Instead, quantitative methods like mathematical modeling are required to factor in all the feedback loops and bifurcations.

1.2. Mathematical modeling of biological systems

Making predictions about a biological system is challenging due to the complexity introduced by interwoven feedback loops. Omics-based knowledge such as biochemical pathways only allows a *post-hoc* explanation of observed behavior, but no predictions (Voit 2018). For example, it is known that an administration of insulin will lower the blood sugar of diabetes mellitus patients (here and in the following see De Meyts 2016). Using the data available about metabolic pathways, this behavior can be explained as insulin binding to a receptor protein in the cell membrane, changing its conformation, and thus activating an intracellular signal cascade, which after several intermediate steps leads to the transport of the insulin-sensitive glucose transporter GLUT4 from intracellular vesicles to the cell membrane, where it facilitates the diffusion of glucose into the cell. If, however, one were to develop an alternative to insulin, or try to target a different part of the insulin signal transduction pathway, a prediction of the actual effects of candidate substances would be much harder. To do this one would have to identify all possible interactions of the substance with any of the hundreds of individual molecules involved in the insulin signal transduction pathway and then track the resulting changes through the whole pathway. Due to interwoven feedback loops, some changes might both lead to an increase and a decrease in a specific molecule concentration. Using only the qualitative information of the pathway structure, it is not possible to know whether the sum of these influences will be a net positive or a net negative (Voit 2018).

This leaves only two possible options: Either one needs to perform a wet-lab experiment, recording how live cells or organisms react to the substance in question under controlled laboratory conditions; or quantitative information—such as specific reaction rate constants or substance concentrations—has to be collected and used to perform the same experiment *in silico* as a computer simulation (Palsson 2000). While wet-lab experiments and clinical trials are the only way to know for certain how the actual biological system behaves, they are also time-consuming, expensive, and can fail due to small perturbations. This is, for example, reflected in drug discovery, which is a process involving a considerable amount of trial and error with success rates of clinical trials as little as 3.4% for oncology and still only 20.9% for all other therapeutic groups (Wong, Siah, and Lo 2019). Drugs can also already fail in earlier stages such as preclinical trials, which have a success rate of 31.8% (Takebe, Imai, and Ono 2018). Additional to difficulty and price, some experiments cannot be performed by isolating cells or tissue *in vitro* in a test-tube or Petri dish, but must be executed *in vivo* in animal testing or clinical trials (Doke and Dhawale 2015). This raises ethic considerations, especially with regard to the low success rates of drug discovery (Festing and R. Wilkinson 2007). While there ultimately is no alternative that yields the same certainty as an *in vivo* experiment, it is clear that animal suffering should be reduced to a minimum.

This leads to the second option to make and test predictions of complex biological systems: mathematical modeling. If a biological system can be described in sufficient detail as a set of variables and equations, *in silico* experiments can be conducted by changing the parameters of these equations and calculating the resulting state of the system after a given time period (Di Ventura et al. 2006; Palsson 2000). It is usually very hard if not impossible to formulate a mathematical model in closed form, that is a set of functions $x_i(t)$ that can simply calculate the exact value of all variables x_i of the system at any time t without considering intermediate time steps (here and in the following see Borzì 2020). Instead, mathematical models are based on incremental numeric solutions, which only describe the change of the system from one small time step to the next. This is especially convenient since this is the exact kind of information that can be obtained from omics data: Is there an immediate positive or negative link between variables a and b? In order to perform simulations that allow accurate predictions, these qualitative links between variables have to be supplemented with quantitative parameters that define the strength of the link. In contrast to the *variables* of the system, *parameters* cannot change during the simulation and do not depend on any other variables or parameters. Instead, they have to be determined in advance either by directly measuring them in wet-lab experiments, or by parameter fitting, which is a mathematical optimization procedure that finds values that allow the model to reproduce the recordings of a wet-lab experiment as closely as possible (Voit 2018). Once parameter values have been found, evaluating the system step by step propagates changes through all feedback loops and, after a sufficiently long iteration, provides the desired prediction how the whole system changes after a given time period.

Mathematical models are used in many areas of biology across all levels of organization. The following examples are selected to show the diversity of the application area, but they do not necessarily constitute the most influential studies in that respective area:

Ecosystem Bologna, Chandía, and Flores (2016) estimate the impact of human population on the Lake Edward hippos in the Virunga National park between Congo and Uganda. In a more general approach, Gilad et al. (2004) used a mathematical model to find evidence that the spatial distribution of certain key species called "ecosystem engineers" can explain species loss events.

Organism Steppe et al. (2006) investigate the sap flow of a beech tree, comparing a hydraulic and an electrical analog model which both include a stem growth component that was not considered in previous models.

Organ Seidel (1997) created a mathematical model of the human baroreflex, which regulates the heart rate variability. The model is also used in this dissertation. As a second example, Gardiner et al. (2011) also developed an organ-level mathematical model to assess the quantitative importance of renal oxygen shunting, which is the diffusion of oxygen from arteries to veins thus bypassing the kidneys and leading to an oxygen deficiency (hypoxia).

Cellular Courtemanche, Ramirez, and Nattel (1998) developed an electrophysiological model to study action potential generation in human atrial cells. Mendoza-Juez et al. (2012) investigate tumor cell metabolism using a mathematical model that takes dynamical changes in the metabolism into account when a tumor cell switches from mainly using glucose to lactose and vice versa, with the ultimate goal to better inform metabolic therapies.

Molecular Perley et al. (2014) modeled T-cell activation in the human immune system as a complex signalling network involving four different signalling pathways, identifying several key regulation factors.

Genetic Woller et al. (2016) used a mathematical model involving gene expression levels to simulate the circadian clock of the liver responsible for synchronizing the metabolism to food timing, finding evidence that food and exercise timing might be key factors to avoid metabolic disorders.

Behind the scenes, these models rely on different mathematical formalisms, i.e. formal systems that allow to define the model in such a way that it can be simulated using generalized approaches developed for this kind of formal representation (here and in the following see Gershenfeld 2011; Banerjee 2014). To classify these formalisms, a few key distinctions can be made: First, a model can either use continuous or discrete time steps. Continuous time models represent the system state as a continuous trajectory of a set of variables that can be evaluated at any point in time from the start to the end of the simulation. These trajectories are calculated using differential equations, which define the gradient of the state variables at the current time depending on the current state.[2] Differential equation solvers then either assume that the system will follow this linear gradient between sufficiently small time steps or employ techniques like the Runge-Kutta method, to calculate a more accurate value for the next time step. The most common formalism used for differential equations are ordinary differential equation (ODE), which require all equations of the system to be given in gradient form. Alternatively, differential-algebraic equations (DAEs) also allow the inclusion of direct relations between variables without using derivatives to state conservation laws such as the conservation of mass. An example for a typical continuous model using differential equations are action potential models, which define the continuous trajectory of the membrane potential.

Leaving the continuous world behind, discrete time models define explicit time steps and their equations govern the state transition from step n to step $n + 1$. Formalisms for discrete time models include so-called difference equations, which are the discrete equivalent of differential equations, and discrete-event simulations (DESs), which define state transitions in the form of events that can be triggered by changes of variable values (Fishman 2013).

[2] This is true for first-order differential equations. In higher-order differential equations, a higher-order derivative is defined instead of the gradient, which can then also depend on the lower-order derivatives of variables.

For example, early models of the human baroreflex included many beat-to-beat models, which only measured variables such as blood pressure once per heart beat, allowing for very efficient long-time simulations (DeBoer, Karemaker, and Strackee 1987).

The second major distinction factor of modeling formalisms is the inclusion of spatial distribution (here and in the following see Gershenfeld 2011; Banerjee 2014). While ODE only consider time derivatives, partial differential equations (PDE) can also include derivatives across spatial dimensions, which enables two- or three-dimensional models of tissue. In PDE models, special care has to be taken to choose correct boundary conditions at the edges of the simulation space. The most common method for solving PDE is the finite element method (FEM), which discretizes the spatial dimensions with sufficiently high resolution so that the discretization error does not affect the overall behavior of the model. If not only the time but also the state of the system is discrete, it can be described as a cellular automaton, that is a grid of cells, which change their state in fixed time steps depending on the previous state of themselves and their neighbors (Wolfram 1983).

Third, modeling formalisms can also be classified in deterministic and stochastic representations (here and in the following see Gershenfeld 2011; Banerjee 2014). All previously discussed formalisms are deterministic, which means that they will always produce the same trajectories when the same initial conditions are used. Conversely, stochastic formalisms introduce a random element to simulations, which is governed by probability distribution functions. This can both be helpful at the micro scale, e.g. for collision probabilities of chemical molecules (Gillespie 1977), or at the macro scale, e.g. for the behavior of individuals in an ecological model (Black and McKane 2012). The stochastic extensions of already covered formalisms are called stochastic differential equations (SDEs) for ODEs, stochastic partial differential equations (SPDEs) for PDEs, and discrete-time Markov chain (DTMC) for discrete formalisms (Grimmett and Stirzaker 2020).

Finally, the state of the system can also depend on earlier values of variables (here and in the following see Banerjee 2014). For example, a neural signal from the baroreceptors that measure aortic blood pressure will take some time to travel to the autonomic nervous system (ANS) (Wehrwein and Joyner 2013). In the continuous case, time dependencies introduce delay differential equationss (DDEs), which introduce delay terms to specify the value of a variable at an earlier time in the simulation. In the discrete case, DESs can already represent this behavior by scheduling events for future time steps and jumping to this time step instead of just following a fixed time increment (Fishman 2013).

Some models require a mix of these formalisms and not every mix has its own name or theory (Bardini et al. 2017). However, there are many specialized mathematical frameworks, which allow to handle one or multiple formalisms very well for a specific set of models. This includes, for example, Petri nets for DES (Reisig 1985) and bond graphs for DAE or ODE (Thoma 1975). A framework that has gained popularity in systems biology in particular is agent-based

modeling (ABM), which can be seen as an alternative to equation-based modeling altogether (here and in the following see Macal and North 2005). In ABM, the main components are not equations but so-called agents, which are independent units with their own memory that exhibit goal-based behavior guided by rules. These rules can be implemented in any of the aforementioned formalisms, and agents can also have a set of meta-rules, which allows to change the rules based on information about the environment or their own internal state.

In order to move from a theoretical mathematical framework to an actual simulation, models must be implemented in one of the many available modeling languages: Most models are still built with general-purpose programming languages (Clerx et al. 2016) such as FORTRAN (Chivers and Sleightholme 2018), C (Kernighan and Ritchie 1988), C++ (Stroustrup 2013), and Java (Sierra and Bates 2005).[3] In these languages, the numerical method, such as the Runge-Kutta method for solving ordinary differential equation (ODE), must be implemented by the modeler, and the equations of the model must be brought in the specific form that is required by this specific method and implementation. The benefit of this approach is that it allows full control of the code that is being run, but this amount of control makes it also inflexible and error-prone (Clerx et al. 2016).

A step-up towards comfort and flexibility are specialized languages, which focus on numerical problems like MATLAB (The MathWorks 2022c) and the open-source equivalent Octave (Eaton et al. 2021). In these languages, numerical solvers are part of the standard library, allowing the modeler to focus solely on writing the equations of the system and configuring the parameters of the solver. An alternative to this approach is to use a specialized library in a general-purpose language, which includes numerical solvers and other utility functions for mathematical modeling. Examples of this are SimuPy (Margolis 2017) for Python (Van Rossum and Drake 2009) and DifferentialEquations.jl (Rackauckas and Nie 2017) for Julia (Bezanson et al. 2017).

Other languages completely detach the mathematical description of the model from the code that is required to solve the resulting equation system. For example, Modelica (Elmqvist 1978), Antimony (Smith et al. 2009), and the Simscape (The MathWorks 2022d) language within MATLAB fall in this category with Modelica translating models to C code, Antimony to Python, and Simscape to MATLAB. On a smaller scale, this can also be achieved by developing embedded domain-specific languages (DSLs) in other languages such as Python (e.g. PySB (Lopez et al. 2013) and Python Simulator for Cellular systems (PySCeS) (Olivier, Rohwer, and Hofmeyr 2005)) or Julia (e.g. Modia (Elmqvist, Henningsson, and Otter 2016)).

[3] Clerx et al. (2016) looked at 60 models of action potential generation that were cited in D. Noble, Garny, and P. J. Noble (2012). For other modeling areas, quantitative evaluations are hard to find, but it can be assumed that numbers are similar where similar mathematical foundations are used.

As a final step towards modeling convenience, graphical tools can be used to guide the user in building and analyzing their models (Hoops et al. 2006). These tools often use Extensible Markup Language (XML)-based exchange formats such as SBML (Hucka et al. 2003) and CellML (Clerx et al. 2020), which are not designed to be read and written directly by humans but still allow to save the mathematical definition of a model in a tool-independent way.

This multitude of languages, tools, and formalisms allows introducing *in silico* experiments wherever there is enough reliable data to inform the model design and parameter fitting processes. With this, mathematical modeling saves costs, time, and reduces animal suffering in *in vivo* studies.

1.3. Multi-scale modeling

As the basic idea of systems biology is to investigate biological systems at a larger scale, recent years have seen increasingly complex models that span multiple scales of time and space along with initiatives that strive to collect and integrate models for a specific biological system:

- Karr et al. (2012) achieved a breakthrough by building a whole-cell model of *Mycoplasma genitalium*, which includes the function of all 525 genes of the bacterium and is able to simulate the whole cell cycle up to cell division.
- Millard, Smallbone, and P. Mendes (2017) developed a model of similar scale for the cell metabolism of *Echerichia coli*, including 62 metabolites.
- In Germany, the successive projects HepatoSys, Virtual Liver Network, and Liver Systems Medicine (LiSyM) (Desmond 2022), collected models of the human liver and aim to inform medical practice with their findings (Jansen et al. 2019).
- Both the National Simulation Resource (NSR) at the University of Washington (J. B. Bassingthwaighte 2000; J. Bassingthwaighte and Jardine 2022) and the International Union of Physiological Sciences (IUPS) (P. Hunter, Robbins, and D. Noble 2002; International Union of Physiological Sciences 2022) have launched projects to explore the human physiome by collecting and combining physiological models.
- Plants in silico (X.-G. Zhu et al. 2016) follows a similar approach for plant physiology.
- The Blue Brain Project (Markram 2006; École polytechnique fédérale de Lausanne 2022a) aims to build digital reconstructions and perform simulations for the mouse brain.

All of these projects have one property in common: They face the challenge of incorporating model parts and performing simulations across multiple scales of space and time. For example, Karr et al. (2012) model metabolic interactions of molecules and genes in the cell at a timescale of seconds to ultimately observe the cell division after ten hours. On the spatial scale, the Blue Brain Project aims to build a model that starts with individual ion channels in the cell membrane of neurons at the nanometer scale and reaches to the centimeter scale for the whole rodent brain (École polytechnique fédérale de Lausanne 2022b).

In general, biological experiments often span multiple scales of time and space, because the independent variable[4], which is manipulated by the researcher, resides either several scales lower or higher than the dependent variables that are observed (Walpole, Papin, and Peirce 2013; Bardini et al. 2017). Beta-blockers block Norepinephrine-activated ion channels in heart muscle cells, thereby inhibiting the effect of sympathetic nervous activity on the action potential—an independent variable on the scale of milliseconds and micrometers (Wiysonge et al. 2017). The goal of treatment and therefore the dependent variable is the long-term treatment of increased blood pressure (hypertension) across the whole body, which increases the scales involved to meters and years (Wiysonge et al. 2017).

An example for the opposite directions are neurological studies, like the work of S. C. Miller et al. (2009), which measures oxytocin levels in humans before and after petting their dog. In this case, the independent variable is the interaction with the dog which takes place on a scale of meters and minutes, and the dependent variable is the oxytocin level which is observed across the same timescale but acts on a molecular scale of nanometers. The reason for this multi-scale entanglement is that all levels of organization in a biological system from genes to proteins to cells to tissue to organs to whole organisms, populations and ecosystems are linked by both upward and downward causations (Bardini et al. 2017).

Even when both the dependent and independent variable are on the same scale, the relevant causal link between them may still span multiple higher or lower levels of organization. For example, understanding of how two molecules interact with each other might require a quantum mechanical representation of both molecules as intermediary step (Warshel 2014). In all these cases of multi-scale phenomena, a mathematical model that wants to allow the observation of changes in the dependent variable based on changes to the independent variable has no other choice but to model the scales of time and space in between them in some detail.

These *multi-scale* models pose unique challenges due to their complexity. The simplest way to cover multiple scales is a *micro-level* model, which focuses on the lowest organizational level involved and finds equations describing components at this level (Bardini et al. 2017). In the

[4] Here I refer to the general terminology of scientific experiments, where an independent variable is modified in order to observe and measure its relation with another variable that depends on the independent variable by some law or rule.

next step, thousands of these homogeneous components are combined and connected to each other to reach the next hierarchical layer. While this is the simplest approach conceptually, the resulting models are computationally demanding, which limits the distance that can be spanned to one or two scales (Pitt-Francis, Garny, and Gavaghan 2006).

The alternative is a *multi-level* approach, which combines several descriptions of the system at different organizational levels (Bardini et al. 2017). In this case, the equations used for the different parts of the system are heterogeneous. For example, a cardiovascular model could include organ-level equations for the lung, the autonomic nervous system, the blood vessels, and the kidney, but at the same time include a more detailed description of the heart that reaches to the cellular level to simulate action potential propagation.

It is important to distinguish the terms *level* and *scale*: A *scale* is a measurable dimension, which can be expressed by the metric prefix (milli, kilo, etc.), whereas a *level* is a layer of abstraction that corresponds to a conceptual category (here and in the following see Bardini et al. 2017). Usually the *levels* of a model correspond to biological levels of organization (cell, tissue, organ, etc.) but there can also be models that mix several conceptual levels at the same organizational level. One example could be a model of a sodium channel that is composed of two sets of equations, one for the short-term behavior of action potential generation, and one for the long-term behavior of excitability regulation through slow inactivation. In this example, the whole model can be described purely on the organizational level of macromolecules, but still has the two conceptual levels of “short-term” and “long-term”.

The advantages of having multiple *levels* in a model are twofold: First, the computational complexity of *multi-level* models can be reduced by numerous approaches (Dada and P. Mendes 2011). Secondly, the hierarchical approach makes it easier to directly observe variables of interest, since they are explicitly modeled instead of being implicitly defined by the state variables of thousands of individual components (here and in the following see Bardini et al. 2017). The downside of *multi-level* models is that the model itself becomes more complex, because different equations are needed at each hierarchical level and interactions between levels have to be described explicitly. Often, these descriptions have to be taken from multiple preexisting models, which have to be fitted together. This raises the issue of reproducing models published by other researchers and reusing them in a different context than the one they were originally designed for.

1.4. Reproducibility of mathematical models

As the complexity and size of models grows, they become increasingly difficult to handle and existing modeling workflows and tools start to reach their limits (here and in the following see Porubsky et al. 2020). One area where this is especially apparent are reproduction attempts of the methods and results of a modeling study by a different lab. Reproducibility is one of the cornerstones of the scientific method itself, but it is especially important for multi-scale models. This is true regardless of the model structure: A micro-level model requires a model of the lowest-level unit, which is duplicated across spatial dimensions, while a multi-level model is composed of heterogeneous sub-models per definition (Bardini et al. 2017). In both cases, it is likely that these base components will be provided by a different study and possibly a different research team. In order to build a new multi-scale model, it is therefore necessary to obtain executable code of these base components that can produce the same results as the original study. Such a one-to-one reproduction of a study's methods provides a first indicator if a model is reliable in the sense that it actually behaves as described in the accompanying publication. However, in order to use a model as a component in a new multi-scale model, it must also still produce correct results if the model code is adjusted, and it must be usable with different simulation tools. Adjustments may include a change of input signals, an embedding of the code in another code base, a redesign of the model structure, or even a complete translation of the model equations into a different modeling language. It is therefore essential that the model's behavior and properties can be reproduced under varying experiment conditions. Without components that fulfill these reproducibility requirements, there is little hope to build complex multi-scale models.

One of the promises of *in silico* experiments is that they should have fewer issues with reproducibility than wet-lab experiments, which can fail easily due to a complex and intricate environment that is susceptible to small perturbations (Porubsky et al. 2020). For regular biological research, close to 80% of researchers state that they have failed to reproduce the results of an experiment by someone else (Baker 2016) and reproducibility rates as low as 11% have been observed for a set of landmark studies in preclinical cancer research (Begley and Ellis 2012). For *in silico* studies in systems biology, the rates are indeed better, but unfortunately they are also far from perfect. In Tiwari et al. (2021), the curators of the BioModels database—one of the largest databases for mathematical models of biological systems—state that only 51% of 544 model submitted to the database were directly reproducible, meaning that at least one figure of the original article could be reproduced from the submitted code without major adjustments. A smaller study of models in quantitative systems pharmacology found similar results with only 4 of 12 examined models being reproducible in the sense that figures from the original article could be generated via an existing "run" script (Kirouac, Cicali, and Schmidt 2019). As a single extreme example, Topalidou et al. (2015) required three months to reproduce a computational model of the basal ganglia. The reproducibility issues encountered in these articles are listed in table 1.1.

Reproducibility issue	reference	recoverable
missing executable simulation script	K, T	no
missing parameter values	T	no
missing initial values	T	no
missing data	K, T	no
structural errors in equations	T	no
incorrect parameter or initial values	T	no
sign errors	T	yes
typographical errors in parameter values (e.g. 0.1 vs. 0.01)	T	yes
unit errors (e.g. μs vs. ms)	T	yes
insufficient code documentation	T	yes
mismatch between variable names in article and code	T	yes

Table 1.1.: **Overview of common reproducibility issues for mathematical models in systems biology.** References are T: Tiwari et al. (2021), K: Kirouac, Cicali, and Schmidt (2019); "recoverable" means that reproduction was still possible with manual effort.

The main reasons for a reproduction attempt to fail were missing information of all kinds, structural errors in equations, or non-obvious errors in parameter or initial values (see table 1.1). Several other types of errors and obscurities could be overcome by researchers given enough time and expertise, but made reproductions cumbersome. Taken together, these issues constitute major hurdles for the development of multi-scale models and need to be addressed to allow scientific progress in systems biology (Porubsky et al. 2020).

Numerous researchers have picked up on this and suggested solutions at various levels of the academic system.[5] Modeling-specific suggestions generally fall into the following categories: suggestions for a) what to publish, b) where to publish it, c) which format to use for publications, and d) how to improve journal procedures.

Regarding what to publish, there is a consensus that, since missing code or data is one of the major reasons for failed reproduction attempts, all scripts containing simulation setup or plotting commands should be published along with the model specification (Sandve et al. 2013; Lewis et al. 2016; Medley, A. P. Goldberg, and Karr 2016; Waltemath and Wolkenhauer 2016; Stodden, Seiler, and Z. Ma 2018; Mulugeta et al. 2018; Hellerstein et al. 2019; Kirouac, Cicali, and Schmidt 2019; Papin et al. 2020; Porubsky et al. 2020). In particular, there should be an *executable* piece of code that can be used to run simulations from the article without any manual adjustments. Additionally, the simulation output of the model should be published

[5] A literature research produced 13 major publications between 2013 and 2020, which listed a total of 42 distinct recommendations. In this section I only present those suggestions that occurred in multiple publications or that are broadly applicable.

along with any other experimental data required to reproduce plots in the article (Sandve et al. 2013; Medley, A. P. Goldberg, and Karr 2016; Waltemath and Wolkenhauer 2016; Stodden, Seiler, and Z. Ma 2018; Kirouac, Cicali, and Schmidt 2019; Porubsky et al. 2020).

When deciding *where* to publish a model, specialized model databases such as BioModels (Malik-Sheriff et al. 2019), the IUPS Physiome Model Repository (Sarwar et al. 2019), or the ModelDB (McDougal et al. 2017), which allow model discovery via semantic information, should be favored (Medley, A. P. Goldberg, and Karr 2016; Waltemath and Wolkenhauer 2016; Mulugeta et al. 2018; Porubsky et al. 2020). In a broader sense, this is covered by the guiding principles that scientific data in general should be Findable, Accessible, Interoperable, and Reusable (FAIR) (M. D. Wilkinson et al. 2016). In systems biology, the FAIR initiative has lead to the development of SEEK, a research platform for sharing various research artifacts including models and simulations (Wolstencroft et al. 2015). FAIRDOMHub is a web-service built upon SEEK that allows systems biology researchers to upload and share their research assets in a FAIR manner (Wolstencroft et al. 2017).

A publication format that gained specific interest in the systems biology community is literate programming in the form of electronic notebooks that mix textual descriptions and code (Sandve et al. 2013; Topalidou et al. 2015; Lewis et al. 2016; Waltemath and Wolkenhauer 2016; Medley et al. 2018; Mulugeta et al. 2018; Hellerstein et al. 2019). However, electronic notebooks can be too rigid for the creation of large and complex models and pose some difficulties for version control (Medley et al. 2018). Alternatively, and especially for larger models, workflow systems such as Galaxy (Afgan et al. 2018) or the Konstanz Information Miner (KNIME) (Berthold et al. 2008) can be used to publish simulation procedures in a format that ensures methods reproducibility through the use of standardized components (Sandve et al. 2013; Waltemath and Wolkenhauer 2016; Medley et al. 2018). Regarding formats for model exchange and annotation, the COmputational Modeling in BIology NEtwork (COMBINE) also collects and promotes a set of standard suggestions for the field of systems biology (Waltemath et al. 2020).

Academic journals are also in part responsible for promoting and ensuring reproducibility. This can include the adoption of publication checklists such as Minimal Information Required In the Annotation of biochemical Models (MIRIAM) (Novère et al. 2005) and Minimal Information About a Simulation Experiments (MIASE) (Waltemath et al. 2011), which have to be completed by authors to ensure that they follow reproducibility guidelines (Lewis et al. 2016; Waltemath and Wolkenhauer 2016). Additionally, a "seal of approval" could provide missing incentives for researchers to put additional effort into assuring that their work is reproducible beyond the required minimum for publication (Lewis et al. 2016; Waltemath and Wolkenhauer 2016; Kirouac, Cicali, and Schmidt 2019; Papin et al. 2020; Porubsky et al. 2020). Ultimately, the only way to guarantee the reproducibility of the methods of an *in silico* experiment is for the reviewers to actually perform this reproduction. One of the most promising approaches is a pilot project that is a collaboration between the Center for

Reproducible Biomedical Modeling (Sauro et al. 2022) and the journal *PLOS Computational Biology*: Authors can opt in to an additional step in peer review where reviewers evaluate the reproducibility of their methods (Papin et al. 2020). An alternative approach is taken by the journal *Physiome*, which exclusively publishes articles that demonstrate the consistency and reproducibility of already published *in silico* studies and that are assessed by independent *Physiome* curators (Nickerson and P. J. Hunter 2017). This effectively generates a "seal of approval" that integrates with existing measures of academic success, since it increases publication and citation count.

If reproducibility is to be assessed objectively—be it by journals or individual researchers—it is important to clearly define what is meant by the term. Unfortunately, the word *reproducibility* has been used ambiguously in scientific literature (Plesser 2018). The Association for Computing Machinery (ACM), uses the term *reproducibility* when a different team can use a modified experimental setup to generate the same results as the original study. If the same experimental setup is used instead, this is called *replicability*, and if the team also is the same the term *repeatability* is used (Association for Computing Machinery 2022b). A competing terminology by Claerbout and Karrenbach (1992) also uses the terms *replicability* and *reproducibility*, but with interchanged meanings. Due to this confusion, Goodman, Fanelli, and Ioannidis (2016) proposed a different terminology that only uses the term *reproduction*, avoiding the linguistic similarity to *repetition* and *replication*, and instead focuses on the question of *what* should be reproduced—methods, results, or inferences. The ACM changed their terminology in August 2020 (Association for Computing Machinery 2022a) to fit the definition by Claerbout and Karrenbach (1992). However, I agree with the assessment of Plesser (2018) that an explicit distinction of the reproduction target is preferable over an implicit distinction using words that have ambiguous meaning in common language. In this dissertation, I therefore specify the subject of reproduction whenever my own work is concerned or I can be sure of the definition used by others. When I refer to work where I do not know which terminology was applied, I fall back to simply using the word "reproducibility" without further specification.

1.5. Model engineering

In addition to the modeling-specific solutions discussed in the previous section, researchers also advocate for general software engineering solutions that can facilitate reproducibility across multiple aspects. The key observation to make here is that a mathematical model can be seen as a piece of software. The Encyclopaedia Britannica defines software as "instructions that tell a computer what to do" (The Editors of Encyclopaedia Britannica 2021). Certainly, a formal model definition is an essential part of the instructions required to perform an *in silico* experiment on a computer. As such, it seems obvious that the rules for software

engineering should—at least to some extent—also apply to the creation of mathematical models (Hellerstein et al. 2019). Following Hellerstein et al. (2019), I use the term *model engineering* for this application of software engineering techniques to mathematical modeling.

In fact, numerous software engineering techniques are recommended in recent literature:

- Structured and thorough documentation of the model code can complement the methods section in a scientific article with additional details that facilitate reproduction attempts (Lewis et al. 2016; Waltemath and Wolkenhauer 2016; Stodden, Seiler, and Z. Ma 2018; Mulugeta et al. 2018).
- Version control helps to trace provenance of model parameters and documents which version of a model was used for individual experiments (Sandve et al. 2013; Lewis et al. 2016; Medley et al. 2018; Mulugeta et al. 2018; Hellerstein et al. 2019; Porubsky et al. 2020).
- Unit tests verify the correctness of model components and in a version controlled codebase they can identify changes that accidentally break methods reproducibility (Medley, A. P. Goldberg, and Karr 2016; Medley et al. 2018; Mulugeta et al. 2018; Hellerstein et al. 2019; Porubsky et al. 2020).
- Open standards can be used to make models available to a larger audience and increase interoperability between models (Medley, A. P. Goldberg, and Karr 2016; Waltemath and Wolkenhauer 2016; Kirouac, Cicali, and Schmidt 2019; Mulugeta et al. 2018; Porubsky et al. 2020).
- Appropriate file formats and style guides help to keep the code human-readable and facilitate results reproduction in a different language or with different tools (Medley et al. 2018; Hellerstein et al. 2019).
- Modularity of model code facilitates the reproduction of results in a different context than the original experiment setup (Lewis et al. 2016; Hellerstein et al. 2019; Porubsky et al. 2020).
- Object-orientation in particular is a technique for establishing modularity that is well-suited to represent biological structures (Mulugeta et al. 2018; Hellerstein et al. 2019; Porubsky et al. 2020).
- Virtual machine specifications go beyond the code itself to make the *environment* reproducible that the model code needs to be executed (Sandve et al. 2013; Lewis et al. 2016; Waltemath and Wolkenhauer 2016; Porubsky et al. 2020).
- Long-term archiving of code ensures results reproducibility for future generations (Medley, A. P. Goldberg, and Karr 2016).

As already indicated, these techniques do not only facilitate a one-to-one reproduction of methods or results, but also the reuse of models in a different context that may require their modification (Waltemath and Wolkenhauer 2016; Hellerstein et al. 2019). This is especially

important for multi-scale models, which often consist of multiple component models from different sources (X.-G. Zhu et al. 2016). Typically, these component models have to be adjusted to fit into the context of the larger multi-scale model by changing or adding interfaces between components and the next higher organizational level or by replacing a coarsely defined part of a model by a more detailed version (Dada and P. Mendes 2011; Neal et al. 2015). This is similar to the idea that software must be maintainable, because its requirements may change during its life cycle or because it contains undetected errors, which only become apparent in specific situations (Riaz, E. Mendes, and Tempero 2009). Many of the aforementioned techniques—such as modularity, version control, documentation, style guides, and unit tests—focus precisely on this problem.

To sum up all benefits of software engineering techniques for mathematical modeling, Hellerstein et al. (2019) coin the term *model engineering*. They argue that while models currently available in databases such as BioModels are largely still manageable without paying much attention to design aspects at the code level, this will change when modeling moves from whole-cell models to whole-tissue, whole-organ, or even whole-organism models. Software engineering has already made this transition from small single-file applications to huge interconnected systems comprising millions of lines of code decades ago (Ostrand, Weyuker, and Bell 2005). Similarly, mathematical modeling should also be considered an emerging engineering discipline that will require more structured solutions in the near future (Hellerstein et al. 2019).

1.6. Modelica

In the context of model engineering, it is interesting to look at other engineering disciplines that also use mathematical modeling for specification, optimization, and diagnosis. This includes, for example, aerospace, automobile or power plants (Briese, Klöckner, and Reiner 2017; Bouvy et al. 2012; Casella and Leva 2006). A language that has gained substantial interest in these areas is Modelica (Mattsson and Elmqvist 1997). It is described as a "non-proprietary, object-oriented, equation based language to conveniently model complex physical systems" (Modelica Association 2022d). As such, it could very well also be suited to model complex *biological* systems.

Indeed, the size of industrial Modelica models is often well beyond the size of state-of-the-art biological models. The largest open-source Modelica libraries such as Buildings (Wetter et al. 2014) and Integrated District Energy Assessment Simulations (IDEAS) (Jorissen et al. 2018) have well over 100,000 lines of code. One of the largest individual models is the model codenamed `RETE_G` in the work of Casella et al. (2016). It is a proprietary model of the Italian

electrical transmission grid without Sardinia and with a representation of European network connections. It was generated from a Python script parsing a list of network connections and has close to 600,000 equations.

Modelica is also widespread in other areas such as aerospace (Briese, Klöckner, and Reiner 2017) and automobile (Bouvy et al. 2012), but it is currently largely unknown in systems biology. A notable exception is the Physiolibrary (Mateják et al. 2014) and the corresponding Physiomodel (Mateják and Kofránek 2015), an integrative model of human physiology. It is based on HumMod (Hester et al. 2011), a model with over 5,000 variables, whose predecessor has been used in the Digital Astronaut Project by NASA (Summers, T. Coleman, and Meck 2008). In the Physiomodel project, Modelica is used precisely because the previous way of defining the HumMod model as a collection of thousands of XML files turned out to be a major barrier for its use in other projects, where it would have to be adjusted (Kofránek et al. 2017). With over 80,000 lines of code, the Modelica version approaches the size of aforementioned industrial Modelica applications and shows that biological models of this size are both required and can be realized with Modelica.

Both the ability to manage large and complex models and the influence of ideas from the engineering domain make Modelica an interesting tool for the development of multi-scale models and the application of model engineering techniques. The main goal of this dissertation is the assessment of the usefulness of Modelica and the general concepts employed in the Modelica ecosystem for the modeling of biological systems.

1.7. Cardiovascular modeling

In a single dissertation it is not possible to explore all areas of systems biology where Modelica could be of use. This work is therefore only focused on models of the cardiovascular system. One reason for this is their undeniable relevance: In 2017, ischemic heart disease was the leading global cause of death measured in years of live lost (Roth et al. 2018). Additionally, multi-scale cardiovascular models are particularly interesting as an accurate description of cardiovascular phenomena often involves not only the heart but also the lung, the ANS and kidney and relevant phenomena reach from changes in ion concentrations at the biochemical level to overall patient health at the organism level (Klabunde 2012). Regarding time scales, the action potential that triggers a heartbeat occurs over few milliseconds, but clinical heart rate variability (HRV) analysis uses ultra-short-term (60–240 seconds), short-term (approximately 5 minutes), and 24-hour recordings (Shaffer and Ginsberg 2017).

The history of cardiovascular models starts with the first model of a single heart cell by D. Noble (1962). This model was an adaptation of the Hodgkin-Huxley model for a neuron in the squid giant axon (Hodgkin and Huxley 1952). Much like a neuron, a heart cell can generate an electrical signal called an action potential, but this action potential has a different shape and some heart cells are capable of spontaneous action potential generation, which does not require an external stimulus (Klabunde 2012). The Hodgkin-Huxley-model showed that the action potential generation is mainly governed by two ion channels in the cell membrane, which are selective for sodium and potassium ions respectively and change their conformation with the electric potential between the inside and outside the cell membrane. D. Noble adjusted this model to show that the same mechanism could explain the behavior of cells in the Purkinje fibers—a network spanning the ventricles, which propagates electrical signals from the atrioventricular node (AVN), which itself receives signals from the sinoatrial node (SAN), the primary pacemaker of the heart (Klabunde 2012).

Since then, models of the heart have evolved both in spatial dimensions and in detail. Two-dimensional micro-level models span from the cellular level to the tissue level (Moe, Rheinboldt, and Abildskov 1964; Barr and Plonsey 1984; Winslow et al. 1993) and allow to investigate the propagation of an action potential through the heart muscle. In recent years, three-dimensional models have put the organ level in reach (Seemann et al. 2006; Bin Im et al. 2008) and grow ever more detailed. Strocchi et al. (2020) provided full four-chamber geometries of human hearts, but models based on these anatomical structures remain confined to simplified conduction and membrane mechanics.

Since large two- and three-dimensional models are computationally demanding, they often use less detailed formulations for the individual cells. One prominent choice is the FitzHugh-Nagumo model, which simplifies the Hodgkin-Huxley model to just two differential equations (FitzHugh 1961; Nagumo, Arimoto, and Yoshizawa 1962). At the same time, the Purkinje fiber model by D. Noble has been extended by additional ion channels and also active transport of ions through the membrane by pumps. For example, Di Francesco and D. Noble (1985) include several additional ion channels as well as the sodium-potassium-pump, which is responsible for return the membrane potential to its resting state, and the uptake and release of calcium ions by the sarcoplasmic reticulum, which is both relevant for the membrane potential and muscle contraction. Detailed models using Hogdkin-Huxley-style equations now exist for atrial (Courtemanche, Ramirez, and Nattel 1998), SAN (Zhang et al. 2000; Kurata et al. 2002), AVN (Inada et al. 2009), Purkinje fiber (Sampson et al. 2010), and ventricular cells (O'Hara and Rudy 2012). In some cases, however, this level of detail is not sufficient to replicate *in vitro* experiments on single cells, which is why even more detailed Markov formulations are becoming popular, which take into account more subtle state changes of ion channels and pumps beyond the dichotomies of open/closed and inactivated/not inactivated (Iyer, Mazhari, and Winslow 2004; Greenstein et al. 2000; Mazhari et al. 2001).

Similar to heart cells, the heart itself is also typically represented with simplified equations if it is just one part of a larger model. Such models that reduce the spatial complexities involved in a system to a few key points in the spatial dimension are called zero-dimensional (Kamoi et al. 2014) or lumped parameter models (Díaz-Zuccarini and LeFèvre 2007). These representations of the heart generally fall into one of three categories: beat-to-beat, averaged continuous, or electrical analog models. Beat-to-beat models also discretize the timescale and calculate relevant variables for the current heart beat based on their value at the last beat (DeBoer, Karemaker, and Strackee 1987). Conversely, averaged models do not use discrete beat events but instead use the heart rate as a continuous variable in the system and derive the average blood pressure during the heart cycle from this variable (Saul et al. 1991). The third category are electrical analog models, which define the heart in terms of an oscillating circuit that produces a continuous curve for the rise and fall of the blood pressure during systole and diastole (Kamoi et al. 2014). Like averaged models, electrical analog models typically do not introduce the notion of a beat as a discrete event, but rather only allow to set the heart rate as a parameter (Shimizu et al. 2018). Multiple approaches can be combined to obtain hybrid models that are both able to represent blood pressure with sub-beat precision and identify beats as discrete events and include beat-to-beat relationships between variables (Seidel 1997).

The circulation of blood through the body forms a loop, which is why most models that include the heart also include the vasculature in some way. One of the first models of integrated physiology was Guyton, T. G. Coleman, and Granger (1972). It also includes the baroreceptor, which generates a neural signal that is used by the ANS to control the heart rate through the secretion of norepinephrine and acetylcholine; the lung, which is responsible for oxygenating the blood, and the kidneys, which regulate the salt and water content in the blood. In general, models of the vasculature can, similar to models of the heart, be categorized by the amount of spatial information that is taken into account. Guyton, T. G. Coleman, and Granger (1972) represent blood pressure and other variables at a few individual points in the body, which is why this model can be categorized as zero-dimensional model. Later approaches model fluid dynamics in the vasculature in one to three dimensions (here and in the following see Morris et al. 2016). This is difficult and computationally demanding, since it involves solving the Navier-Stokes equations for fluid motion, which are nonlinear PDE. The benefit, however, is that subtle turbulences and disease conditions like aneurysms can be accurately simulated.

In the light of all these different approaches, one of the current and upcoming challenges in cardiovascular modeling is to combine the information gained from different models to obtain experimental results with high precision at the organ or organism level. Both the IUPS Physiome Project (P. Hunter, Robbins, and D. Noble 2002) and the NPR Physiome Project (J. B. Bassingthwaighte 2000) have dedicated their efforts towards this goal. My dissertation

aims to support this work, which is why I do not focus on spatial micro-level models, but start my investigation with a high-level lumped parameter model and then investigate possible integration points for other models, which represent parts of the model in greater detail.

1.8. The Seidel-Herzel model as multi-scale modeling platform

The model that stands at the center of this dissertation is the Seidel-Herzel model (SHM) of the human baroreflex, which was developed by H. Seidel in his PhD thesis under the supervision of H. Herzel (here and in the following see Seidel 1997). It came to the attention of my working group, because it is a relatively small and thus manageable model that nevertheless produces realistic heart rate variability (HRV) curves through the careful introduction of noise terms.

HRV describes the variation in the time interval between heartbeats (Ernst 2014). It is a promising application area for physiological models since HRV is strongly connected to the activity of the autonomic nervous system (ANS) and therefore allows to non-invasively assess its function. It is a valuable risk indicator for several disease conditions in which ANS function plays a role including diabetes, obesity, atherosclerosis, coronary artery disease, and ischemic sudden death (Xhyheri et al. 2012). However, the causal link between diseases and increased or decreased HRV parameters is often still only partly understood (Fairchild et al. 2009; Thio et al. 2018). Mathematical models such as the SHM might be one way to shed further light on these mechanisms.

The SHM is a hybrid lumped-parameter model, which only uses a single variable for blood pressure without any spatial distribution, but includes the function of the baroreceptors, the ANS including the sympathetic and parasympathetic system, the lung, and the signal transduction at the heart including the SAN and AVN (Seidel 1997). With this setup, it is able to simulate first and second degree atrioventricular block (Seidel 1997), carotid sinus hypersensitivity (Seidel and Herzel 1998), congestive heart failure (Kotani et al. 2005), and primary autonomic failure (Kotani et al. 2005) as well as treatment options such as the administration of atropine or metoprolol (Kotani et al. 2005). Additionally, it exhibits interesting dynamical properties including Mayer waves, which are fluctuations in the heart rate on the order of 10 seconds, (Seidel 1997); bifurcations (Seidel and Herzel 1998); and cardiorespiratory synchronization (Kotani et al. 2002). From a technical standpoint it also poses interesting challenges as it includes delay terms, and it mixes discrete beat events with continuous variables such as systemic arterial blood pressure. In some ways it can already be seen as a multi-scale and multi-level model since most equations describe physiological phenomena

on the organ level, but the ANS is modeled on the biochemical level through norepinephrine and acetylcholine concentrations (Duggento, Toschi, and Guerrisi 2012). In terms of time, the systolic blood pressure changes significantly over a few milliseconds, but the slowest observed phenomenon are the aforementioned Mayer waves with a frequency of roughly ten seconds (Seidel and Herzel 1998). More than that, however, the SHM is a great target for extension to move to even lower scales by replacing a component of the model with a more detailed version.

In particular, an interesting target for extension is the AVN, which controls the conduction of beat signals from the atria to the ventricles. A more detailed representation would allow to simulate the effects of premature ventricular contraction (PVC) (Ip and Lerman 2018; Walters et al. 2018) and atrial fibrillation (Marrouche et al. 2018; Pereira et al. 2020), which are both active fields of medical research. Here, the first (D. Noble, Garny, and P. J. Noble 2012) and only[6] detailed model of the AVN was developed by Inada et al. (2009) as a micro-level model consisting of a one-dimensional string of a few hundred individual cells featuring multiple ion channels and pumps. It operates at a lower scale than the SHM, but is considerably larger with over 100 heterogeneous equations per cell (Inada et al. 2009). Incidentally, the model suffers from all the reproducibility issues summarized in table 1.1, which also makes it a great target to investigate possible countermeasures for these issues.

1.9. Research questions

To sum up, some aspects of biology can only be understood by investigating biological systems as a whole, but their inherent complexity due to an interplay of multiple feedback loops requires a quantitative approach. Mathematical modeling can fulfill this requirement, but traditional approaches of structuring and communicating these models are reaching their limits due to an increase in model size and complexity and the need for models that span multiple scales of time and space. This leads to issues in reusability and reproducibility, which persist in many models. In search for a robust solution, the software engineering domain seems to be a promising source of inspiration, since it needs to handle source code at much larger scales. The modeling language Modelica in particular already incorporates many of these ideas, such as a declarative, object-oriented syntax. Due to the vast amount of biological models, it is necessary to restrict the focus of the analysis to cardiovascular models and in particular the SHM, which can serve as an example for many of the typical challenges for multi-scale models in systems biology.

[6] According to a search among over 600 models in the Physiome Model Repository (Yu et al. 2011).

With this dissertation I therefore want to investigate the potential benefits of using Modelica in a model engineering context to build reproducible and reusable models that can serve as a robust basis for multi-scale—and in particular multi-level—cardiovascular models. This task is split up systematically into the following research questions.

RQ01–RQ03 are a systematic deconstruction of the original research question "What are the benefits of the modeling language Modelica for systems biology?".

RQ04 and RQ05 broaden the view and reflect my own insight during the dissertation that it is not so much about Modelica but about a general software engineering mindset that is present in Modelica and required in systems biology.

1.9.1. RQ1: What are the requirements for a modeling language in systems biology?

Before I can assess the usefulness of Modelica, it is critical to get a detailed understanding of the specific challenges posed by biological systems in general and multi-scale modeling in particular. What—if anything—distinguishes biological models from physical ones? Where do classical approaches fail, and what can and has to be done to improve upon them?

1.9.2. RQ2: Does Modelica fulfill these requirements?

Once I have established requirements, I want to critically assess the language Modelica in light of these criteria. I want to examine the properties that distinguish Modelica from other modeling languages, examine existing biological projects in Modelica in more detail and also have a look at the Modelica ecosystem to identify potential clashes with the needs of the systems biology community with regard to openness versus an industrial and proprietary focus.

1.9.3. RQ3: How does Modelica compare to other existing languages?

Of course, Modelica is not the only language that has promising potential for multi-scale modeling. Other candidates such as MATLAB/Simulink, systems biology markup language (SBML), CellML, Python- and Julia-based DSLs have to be examined with similar scrutiny in order to assess both whether Modelica's benefits are unique to it and whether other languages have benefits in areas where Modelica is lacking.

1.9.4. RQ4: Can software engineering techniques in general address the challenges of systems biology?

This question broadens the view away from a single language and more towards a general model engineering mindset. I want to examine which software engineering techniques were particularly helpful during my own modeling tasks. This includes an assessment of object-oriented software design, structured documentation, version control, unit testing, virtualization and continuous integration, and the long term archiving of code.

1.9.5. RQ5: What are the tools and language improvements required to increase the usability of Modelica for systems biologists?

Assuming that RQ2 has a positive answer, the lack of adoption of Modelica in the systems biology community might be due to other factors such as impaired usability or openness of tools. With this question I want to address possible improvements or additional tools that could remedy this situation and bridge the gap between Modelica and the systems biology community.

1.10. Document structure

In the following chapters, I will first give a summary of the publications that are a part of this cumulative dissertation followed by additional contributions presented at the International Modelica Conference. Chapter three then consists of an in-depth discussion of the aforementioned overreaching research questions of this dissertation combining insights from all the work presented in chapter two. My final conclusion and open questions can be found in chapter four, followed by the full text of the publications presented in chapter two.

2. Results

This section presents a short summary of the scientific publications that are a part of this cumulative dissertation or were otherwise created in conjunction with it. Section 2.1 presents articles published in scientific journals, while section 2.2 contains contributions presented at conferences and section 2.3 lists other publications that are not immediately relevant for the main research questions. Copies of the full published articles from sections 2.1 and 2.2 can be found in section 8. For a full explanation of the methods employed to reach the main findings, the reader is therefore referred there. However, the rest of this document does not assume detailed knowledge beyond the summaries presented here, which also provide context to locate the work in the overall dissertation project.

2.1. Journal articles

2.1.1. Characteristics of mathematical modeling languages that facilitate model reuse in systems biology: A software engineering perspective

> C. Schölzel et al. (2021a). "Characteristics of Mathematical Modeling Languages That Facilitate Model Reuse in Systems Biology: A Software Engineering Perspective." In: *npj Systems Biology and Applications* 7.1, art. no. 27. DOI: 10.1038/s41540-021-00182-w.

This publication is the centerpiece of this dissertation. It was conceived after I had already performed a one-to-one translation of Seidel's original implementation of the SHM in C to Modelica (which is detailed in section 2.2.1). This first implementation already showed improvements in reusability, because the code followed the biological structure of the modeled system instead of the imperative logic of the loop used for solving the differential equations. However, it was only a first proof of concept and a thorough investigation of the characteristics of Modelica that were responsible for this improvement was warranted.

In order to do this I specifically wanted to have a closer look at areas of the code that were most challenging and might still be improved in terms of reusability. The most obvious example was the cardiac conduction system that regulates if and when a beat signal generated by the sinus node actually triggers a contraction of the ventricles: In the original implementation of the SHM in C, this part of the model was scattered across three conditional statements involving 8 variables with names such as `ts`, `tse`, and `tvs` in a file with 560 lines of code. My first Modelica implementation gathered these conditional statements in a Modelica class called `Contraction`, which only contained code relevant for the cardiac conduction system, used speaking variable names, and measured only 65 lines of code.

This separation was only possible, because Modelica is a modular language that allows to write both continuous and discrete parts of a model in a declarative style independent of the surrounding logic of the integration loop introduced by the differential equation solving algorithm. This is a clear indication that language choice influences the understandability and thus reusability of a mathematical model. However, it turned out that just having a Modelica implementation of the cardiac conduction system was not enough to guarantee reusability.

My colleague, V. Blesius, wanted to extend the SHM with a trigger for premature ventricular contractions (PVCs). When I tried to help her with the implementation, I quickly noticed that the `Contraction` model did not provide any guidance to decide which variables and conditions would have to change to incorporate the additional external signal. It was still a monolithic part within an otherwise modular model, and in this part every variable somehow depended on almost every other variable. I decided to completely re-implement the whole cardiac conduction system by following the same modular modeling style that previously was applied to the SHM as a whole. In the resulting modular version, the PVC extension was straightforward.

This made it clear to me that there are two parts of the puzzle: Choosing a language that offers the right tools and features, and knowing how to apply them effectively and consistently. I therefore went on to investigate language characteristics that can facilitate model reuse on a larger scale. I re-examined both the whole SHM and the cardiac conduction system and performed a literature research for best practices facilitating reusability and identified six individual language characteristics: In my opinion, modeling languages should ...

- ...be *modular* to support structuring the code according to the biological structure of the modeled system
- ...be *declarative* so that model code can express what is modeled instead of having to define how to simulate it
- focus on *human-readability* over machine-readability to facilitate version control, enable reproduction in different languages, and minimize tool dependence

- ...be *open* in the sense that the compiler, the language itself, and associated tools should be published under an open-source license in order to remove barriers for reuse and reproduction
- ...support a *graphical* representation of models to increase its understandability at a higher level and to communicate the model to domain experts
- ...and be *hybrid* in the sense that it supports multiple modeling formalisms, the most common being ODEs or preferably DAEs along with discrete events.

This is summarized in the acronym MOdular, Declarative, human-Readable, Open, Graphical, and Hybrid (MoDROGH). While I started my investigation with Modelica, a number of languages exhibit the MoDROGH criteria to some extent including but not limited to MATLAB (with the Simulink environment and the Simscape language), the SBML, CellML, Python-based DSLs (such as PySB, the PySCeS, SimuPy or PyDSTool), Antimony, and Julia-based DSLs (such as Modia, DifferentialEquations.jl, or ModelingToolkit.jl).

Indeed, there is no single "best" language with regard to these characteristics, but there exist trade-offs between each pair of languages, which makes it all the more important that modelers are aware of these differences when choosing a language. Additionally, the example of the PVC extension of the cardiac conduction system of the SHM shows that it is not enough to simply use a suitable language. Instead, modelers must be aware of the MoDROGH characteristics to utilize them consistently and effectively.

2.1.2. An understandable, extensible and reusable implementation of the Hodgkin-Huxley equations using Modelica

> C. Schölzel et al. (2020). "An Understandable, Extensible, and Reusable Implementation of the Hodgkin-Huxley Equations Using Modelica." In: *Frontiers in Physiology* 11, art. no. 583203. DOI: 10.3389/fphys.2020.583203.

The next logical step after establishing the MoDROGH characteristics was to test their benefits using a different model. As mentioned in section 1.7, the next model of interest after the implementation of the SHM was the one-dimensional model of the rabbit atrioventricular node by Inada et al. (2009). However, this model is too large to examine the full code of an implementation in detail.

Since the Inada model is based on the Hodgkin-Huxley equations, the original Hodgkin-Huxley model (Hodgkin and Huxley 1952) seemed to be a perfect intermediate step. It is usually communicated either directly through the differential equations or through circuit

diagrams, but both are not ideal entry points for novices since they only indirectly represent the modeled system and therefore require previous knowledge about another formalism (either differential equations or electrical systems). For experts, additional limitations of these representations become apparent when considering large extensions like the Inada model, which includes additional ion channels and pumps up to a total of over 100 equations that interact with each other. It is clear that such models would benefit from some kind of modularization to reduce their complexity. I therefore implemented a modular version of the Hodgkin-Huxley model in Modelica, applying the guidelines established along with the MoDROGH characteristics.

In order to objectively assess whether I reached my goal to increase the understandability of the model, I used cognitive load theory (CLT)—a proven theory in cognitive psychology. In short CLT states that understandability is inversely related to the amount of items that have to be kept in working memory at the same time in order to process a piece of information. Understandability is also lower, if there is a high element interactivity between these items.

The analysis showed that the cognitive load of the Modelica implementation of the Hodgkin-Huxley model was indeed decreased by a factor of six. For extensions of the model, the picture was similar: In the monolithic version, cognitive load grows quadratically with each additional equation due to high element interactivity. In contrast, extensions of the new implementation did not increase cognitive load significantly due to the modular structure and reuse of common code structures. This suggests that the modular implementation is a promising basis for the implementation of the Inada model.

The only major limitation of this approach is the fact that the abstractions in a modular model only allow to examine one component at a time in full detail. An expert who is already familiar with typical equation structures of the domain might indeed prefer to directly look at a list of equations to get an overview of the overall model structure. Fortunately, a modular and declarative modeling style also allows to present a model in different views. As a proof of concept, I created ModelicaScriptingTools.jl (MoST.jl)—a Julia library that both facilitates running and testing Modelica models in Julia and is able to generate a HyperText Markup Language (HTML) documentation listing all equations, parameters, variables, and functions used in a Modelica model (Schölzel 2021b). The equations are grouped by the modular structure of the model to combine the benefits of modularity with the precision of mathematical formulas. A future version of this library could even allow the automatic generation of supplements for scientific articles, ensuring the completeness and correctness of the equations therein.

2.1.3. Countering reproducibility issues in mathematical models with software engineering techniques: A case study using a one-dimensional mathematical model of the atrioventricular node

> C. Schölzel et al. (2021b). "Countering Reproducibility Issues in Mathematical Models with Software Engineering Techniques: A Case Study Using a One-Dimensional Mathematical Model of the Atrioventricular Node." In: *PLOS ONE* 16.7, art. no. e0254749. DOI: 10.1371/journal.pone.0254749.

While I was indeed able to reuse most of the code of the modular version of the Hodgkin-Huxley model and easily fit the additional ion channels into the existing code structure, the re-implementation of the Inada model proved to be challenging due to a host of other reasons:

- Some equations and parameters were missing in Inada et al. (2009). Most of them could be recovered from other sources.
- There were small errors in equations and parameters including sign errors, shifted floating points and missing unit conversions.
- Although the article states that the code is available for download from the journal, it was not. Contacting the authors was unsuccessful, but I could obtain a copy after contacting the journal.
- Both the original implementation in C and the CellML version published in the Physiome Model Repository (PMR) were not executable, i.e. they did not contain enough information to reproduce plots from the article.
- For some parts of the model, no reference plots were available, and for existing reference plots the experiment protocol was not given.
- Some parts of the model contained cryptic equations, whose semantics could only be recovered after following a long chain of references.

When I encountered these issues, I noticed that I already had unconsciously ensured that most of them could not occur for my own models by following common best practices in software engineering. I simply had used these techniques for my own convenience to avoid time-consuming bug fixing, but I noticed that a systematic re-evaluation of their benefit for mathematical modeling in general might be of interest.

This investigation showed that along with the MoDROGH characteristics, continuous integration, continuous delivery, version control, unit testing, object orientation, long-term archiving systems, structured documentation, and published reference outputs are beneficial with

respect to reproducibility. Used together they can guarantee exact methods reproducibility, i.e. that running the same software on a different machine yields the same results, and facilitate results reproducibility, i.e. the reuse of models in a different environment.

2.2. Conference papers

2.2.1. Modeling biology in Modelica: The human baroreflex

> C. Schölzel et al. (2015). "Modeling Biology in Modelica: The Human Baroreflex." In: *Proceedings of the 11th International Modelica Conference*. Versailles, France, pp. 367–376. DOI: 10.3384/ecp15118367.

This and the following papers constitute early work that was conducted before the first journal article described in section 2.1.1. They were presented at the International Modelica conference in order to receive input from the Modelica community and to get a first feel for the strengths and weaknesses of the language. In particular, I wanted to know if Modelica can *at all* truthfully represent a biological model.

There is some reason for doubt, since there is a major difference between biological and physical models even though both use the same mathematical structures: Especially in an engineering context, physical systems can be broken down to well-understood components and laws, forming a unifying theory. Such a unifying theory does not yet exist for biological systems, where descriptions at higher organizational levels are often very rough approximations of the underlying mechanisms, which are often only partially known. As a result, model components become rather heterogeneous. For example, while even two fundamentally different electrical circuits will still consist of the same basic components like resistors and capacitors, there might be no overlap at all between equations used for a high-level description of the pumping behavior of the heart and the low-level electrophysiological behavior of an ion channel within a single heart cell.

As a first proof of concept, I wanted to reproduce the simulation output of the original C implementation of the SHM of the human baroreflex—which I thankfully could obtain from the author—as exactly as possible[7] while still utilizing the object-oriented features of Modelica to structure the code according to the actual biological structure of the modeled system. Indeed, it turned out that Modelica fit quite well to this task. Dividing the model

[7] This first implementation did not contain the noise terms that Seidel added to the base period of the heart and lung, because they would have hindered an exact one-to-one comparison. Following the request of another researcher, I later added the noise terms in version 1.7.0 of the model (Schölzel 2021a).

into components yielded a natural and intuitive representation of the baroreflex which hides implementation complexity and avoids code repetition through encapsulation, inheritance, and object instantiation. Regarding the simulation output, visual differences in blood pressure and heart rate seem only to occur due to the fact that Modelica uses a more complex and precise method of finding the exact point in time when a discrete event occurs than the simple conditional statements introduced to the manual implementation of the fourth order Runge-Kutta method by Seidel.

Only two areas of the model posed difficulties: One was a complicated formula for the "broadening" of the baroreceptor response, which was quite computationally expensive in Modelica.[8] The other were the complex rhythms in the cardiac conduction system involving refractory periods and spontaneous depolarizations of the AVN. Here, the Modelica implementation that seemed straightforward was rejected by the OpenModelica compiler and I needed to add additional continuous variables to emulate the discrete events. This highlights some small areas for possible improvement of the Modelica language and its open-source compilers. However, even without these improvements, the translation of the SHM to Modelica was successful and yielded a fully functional model that is equivalent to the original C implementation.

Comparing the two versions, Modelica was able to adapt well to the biological structure of the modeled system, while the C implementation required to strictly follow the imperative logic of the main Runge-Kutta loop and the C language. Considering the fact that neither the many heterogeneous components nor the different mathematical formalisms of the SHM model could hinder this process, this paints a very promising picture for future projects implementing biological systems in Modelica.

2.2.2. Mo|E — A communication service between Modelica compilers and text editors

> N. Justus et al. (2017). "Mo|E – A Communication Service between Modelica Compilers and Text Editors." In: *Proceedings of the 12th International Modelica Conference*. Prague, Czech Republic, pp. 815–822. DOI: 10.3384/ecp17132815.

During the implementation of the SHM it turned out that the OpenModelica integrated development environment (IDE) OMEdit (Fritzson et al. 2005) did not fit well in a typical software engineering workflow. For example, it did not allow saving intermediate versions of files that still contained syntactical errors, and comment lines and formatting changes

[8] This could probably be replaced by a simple low-pass filter, if the constraint is dropped that the resulting model must be a one-to-one representation of the formulas used in the original C implementation.

introduced to increase code readability could be lost when a model was saved to disk and reopened.[9] To an extent, the Modelica Development Tooling (MDT) (A. D. I. Pop et al. 2006)—a plugin for the Eclipse IDE (des Rivières and Wiegand 2004)—can address this, but like OMEdit, Eclipse is a fully-fledged IDE, which means that it can be slow and difficult to use.

Especially for smaller projects, a simple solution based on a structured text editor would be preferable and ideally could allow switching between different compilers on the fly, thus increasing the interoperability of models. I therefore advertised and supervised a bachelor's thesis that aimed to communicate with the OpenModelica compiler from one of the structured text editors commonly used by software engineers. Inspired by a similar project called ENSIME (ENSIME contributors 2022) for the language Scala, my student N. Justus, chose to implement a server process that communicates with text editor plugins via a series of simple Hypertext Transfer Protocol (HTTP) requests (Justus 2016a). The major advantage of this architecture is that it simplifies the implementations of text editor plugins and therefore allows to easily support multiple text editors.

The resulting project Modelica-pipe-editor (Mo|E) (Justus 2022a; Justus 2022b) gives the modeler full control of the Modelica source code including features such as error highlighting, code completion, navigation based on code structure, documentation rendering, and displaying the type of variables and parameters. Mo|E is the first part of a larger suite of open source Modelica tools called Modelica Tool Ensemble (MoTE) (Justus et al. 2022).

2.2.3. MoVE — A standalone Modelica vector graphics editor

> N. Justus, C. Schölzel, and A. Dominik (2017). "MoVE – A Standalone Modelica Vector Graphics Editor." In: *Proceedings of the 12th International Modelica Conference*. Prague, Czech Republic, pp. 809–814. DOI: 10.3384/ecp17132809.

Modelica allows the annotation of model components with vector graphics icons, which can then be used to compose larger models by drag and drop. This is especially interesting for biological systems, which are often explained using diagrams featuring drawings of the systems components. Unfortunately, this is another area where the only available open source solution, namely OMEdit (Fritzson et al. 2005), lacks many convenience features compared to a standalone vector graphics editor like Inkscape (Inkscape developers 2022). Additionally, it is not possible to import vector graphics created with other tools into Modelica models since Modelica uses its own annotation language, which is incompatible with standards

[9] This critique pertains to OpenModelica version < 1.11.0. OpenModelica has since received updates that address loss of information, but currently in version 1.17.0 still does not, for example, preserve custom formatting of `annotate()` statements.

such as the Scalable Vector Graphics (SVG) format (Quint 2003). I therefore advertised and supervised another student project, also by N. Justus, to implement a standalone editor for the Modelica vector graphics format (Justus 2016b).

The resulting Modelica Vector graphics Editor (MoVE) (Justus 2022c) is a platform-independent Java application that features, for example, selection through transparent components, manual adjustment of stacking order, rotation handles, a snap-to-grid function, and alternate drawing modes for straight lines, perfect circles and perfect squares. As remaining limitations, MoVE cannot be used to place connector icons, which are required to graphically connect components arranged by drag and drop, or handle icon inheritance. For these applications, the Modelica Diagram Editor (MoDE) (Hoppe 2017; Hoppe 2022) was planned, but unfortunately never fully realized.

Since August 2019, MoVE is superseded by the Inkscape plugin Modelica iNKscape plugin (MoNK), which works around the incompatibility between SVG and Modelica by only supporting a subset of SVG. MoNK was only published on Zenodo (Schölzel 2020) and GitHub (Schölzel 2022a), but does not have an accompanying publication in an academic journal or conference proceedings.

2.3. Other scientific work

During this dissertation I also published other scientific work that is not directly related to the main findings of the dissertation. I very briefly summarize these publications here in chronological order for the sole purpose of giving a full account of my scientific work.

Silicon Heart: An easy to use interactive real-time baroreflex simulator

> M. Menzel et al. (2015). "Silicon Heart: An Easy to Use Interactive Real-Time Baroreflex Simulator." In: *Computing in Cardiology* 42, pp. 973–976. DOI: 10.1109/CIC.2015.7411075.

Silicon Heart was an experiment in modularizing a Java implementation of the Kotani et al. (2005) model to the extreme of physically separating of the computing units across several Raspberry Pi (Richardson and Wallace 2013) units. The aim of this project, which I supervised along with my own supervisor A. Dominik, was to create a learning tool for both medical and computer science students to get first insights into the world of mathematical modeling

by pushing some actual buttons and observing the resulting change in the clicking sound and blinking light of the Silicon Heart. It won the best poster award at the 2015 Computing in Cardiology conference.

Can electrocardiogram classification be applied to phonocardiogram data?—An analysis using recurrent neural networks

> C. Schölzel and A. Dominik (2016). "Can Electrocardiogram Classification Be Applied to Phonocardiogram Data? – An Analysis Using Recurrent Neural Networks." In: *Computing in Cardiology* 43, pp. 581–584. DOI: 10.22489/CinC.2016.167-215.

The *Computing in Cardiology* conference was an opportunity to get a glimpse at current medical topics including other modeling attempts. This poster contribution, however, was a side project applying machine-learning techniques—which were my previous research focus—to electrocardiogram (ECG) and phonocardiogram (PCG) data.

Brutus der Orkschamane erklärt die Brute-Force-Methode: Gamification und E-Learning in der Veranstaltung "Algorithmen und Datenstrukturen"

> C. Schölzel (2018). "Brutus der Orkschamane erklärt die Brute-Force-Methode: Gamification und E-Learning in der Veranstaltung ‚Algorithmen und Datenstrukturen'." In: *Proceedings der Pre-Conference-Workshops der 16. E-Learning Fachtagung Informatik*. Frankfurt, Germany.

In parallel to my dissertation, I also developed tools and concepts to improve the learning experience of my students. This included an extensive gamification project for an introductory course to algorithms and data structures, which featured the development of two web-platforms for tracking points and skills and for automatic evaluation of student submissions for programming exercises as well as an entertaining story introducing characters such as Brutus, the orc shaman, who freshened up the dry theoretical lecture content.

Nonlinear measures for dynamical systems

> C. Schölzel (2019). *Nonlinear Measures for Dynamical Systems*. Version 0.5.2. Zenodo. DOI: 10.5281/ZENODO.3814723.

An additional thesis aim that was discarded later on was the improvement of the SHM with regard to the physiological plausibility of variables other than the blood pressure and heart rate. For this task I implemented all nonlinear measures for HRV suggested in Task

Force of the European Society of Cardiology and The North American Society of Pacing and Elecrophysiology (1996) in Python to ensure that modifications of the SHM would not shift any of these measures out of the prescribed range for a healthy patient. The resulting Python package nolds has since been used in many research projects in different fields (e.g. Gilpin 2021; Hamzi and Owhadi 2021; Santana et al. 2021).

Webmodelica: a Web-Based Editing and Simulation Environment for Modelica

> N. Justus and C. Ifland (2022). *THM-MoTE/Webmodelica: A Web-Based Modelica-toolbox*. URL: https://github.com/THM-MoTE/webmodelica (visited on Mar. 7, 2022).

The Master's thesis of N. Justus (Justus 2019) continued his work on the MoTE and was also supervised by me in addition to my own supervisor A. Dominik. This project provided a web-service that reduces the hurdle for students and researchers to perform simulations in Modelica. It featured a web-based structured code editor and simulation and plotting environment based on Mo|E. Unfortunately, the project was discontinued before it could be opened to the public.

HRT assessment reviewed: A systematic review of Heart Rate Turbulence methodology

> V. Blesius et al. (2020). "HRT Assessment Reviewed: A Systematic Review of Heart Rate Turbulence Methodology." In: *Physiological Measurement* 41.8, art. no. 08TR01. DOI: 10.1088/1361-6579/ab98b3.

As already mentioned in section 2.1.3, I worked together with my colleague V. Blesius to investigate PVCs in the SHM since she wanted to investigate heart rate turbulence (HRT). This paper constitutes her overview of the assessment of HRT in literature, finding several discrepancies that severely hinder the comparability of studies. As coauthor, I assisted with the interpretation of results and reviewed the manuscript.

NeuroKit2: A Python toolbox for neurophysiological signal processing

> D. Makowski et al. (2021). "NeuroKit2: A Python Toolbox for Neurophysiological Signal Processing." In: *Behavior Research Methods* 53.4, pp. 1689–1696. DOI: 10.3758/s13428-020-01516-y.

One of the major projects using my Python package nolds was NeuroKit2. The first author D. Markowski consulted me several times to resolve discrepancies between my implementation and other existing implementations of the algorithms in nolds. Together we found several points for improvement both in his and my code, which is why he invited me as a coauthor for the publication of NeuroKit2. The package itself is intended as a high-level, low-barrier entry point for neurological researchers, who want to analyze ECG, electroencephalogram (EEG), and other types of data.

Comparability of Heart Rate Turbulence Methodology: 15 Intervals suffice to calculate Turbulence Slope A methodological analysis using PhysioNet data of 1074 patients

> V. Blesius et al. (2022). "Comparability of Heart Rate Turbulence Methodology: 15 Intervals Suffice to Calculate Turbulence Slope A Methodological Analysis Using PhysioNet Data of 1074 Patients." In: *Frontiers in Cardiovascular Medicine* 9, art. no. 793535. DOI: 10.3389/fcvm.2022.793535.

The second paper by V. Blesius continues the investigation of HRT methodology by providing a clear answer which of the prevalent variants of one parameter should be used for further analysis. The parameter in question was the number of intervals after a PVC that are used to find the turbulence slope, which is the steepest incline across five consecutive data points after the PVC. Again, I assisted with the interpretation of results and reviewed the manuscript.

3. Discussion

The results presented in the previous section now allow to answer the research questions formulated in section 1.9. This will be done by devoting a section to each question and discussing how the findings of my individual publications contribute to an overall answer of this question. For the sake of brevity, I only briefly explain the relevant findings without deriving them in full detail. The interested reader is thus always referred to the full articles in section 8.

3.1. Requirements for modeling languages in systems biology (RQ1)

In order to discuss requirements, it is always worthwhile to investigate the current state of the art and to identify possible areas of improvement. This will be done in section 3.1.1 followed by a detailed introduction of the MoDROGH characteristics established in Schölzel et al. (2021a) and an investigation where they proved useful in my own work in section 3.1.2.

3.1.1. Classical approaches fail for multi-scale models

The classical approach to mathematical modeling in systems biology has a narrow focus on the biological findings with only peripheral interest in mathematical structure and model code. Some authors do discuss the choice of equation structures and the rationale behind seemingly arbitrary fitting functions. For example, at one point Lindblad et al. (1996) report that "the action potential fits were improved by small reductions in the slope factor for inactivation (obtained from Boltzmann fits to the data in Fig. 2A)", which is quite a detailed rationale for a single number in an equation. Other authors only choose to state what was changed and which equations were used without discussion—often to the point that the actual equations only occur as an uncommented appendix block at the end of the article or even just in a supplement. This is, for example, the case for Kurata et al. (2002), who show

two equations in the main article and list the rest only in the appendix; and Inada et al. (2009), whose equations are only given as a supplement. Model code is even less perceived as an important part of a simulation study. Code is usually not shown at all, not part of the review process, and often not even available for download along with the research article (Kirouac, Cicali, and Schmidt 2019; Stodden, Seiler, and Z. Ma 2018). Sometimes, the programming language is named, but without version number or any other details such as the type of differential equation solver that was used (Stodden, Seiler, and Z. Ma 2018). An example is Hulsmans et al. (2017), which combine the Inada AVN model with a model for macrophages and adjust it to the physiology of a mouse. I could have used this work to gain some insight how to integrate the Inada model into the SHM, but it turned out that the mathematical model was only briefly described in a single paragraph and although the paper has a data availability section, the model code is not even mentioned there.

In consequence, if code *is* available, one can then clearly see that it was written for the purpose of performing a single *in silico* experiment without regard to future reuse by other researchers. Both the original versions of the SHM and the Inada model contained no README or any other information that explained where to find which part of the model. They also contained code from preliminary versions that was deactivated and while Seidel manually implemented a Runge-Kutta method, Inada et al. did not provide the main file required to execute the simulation at all. They were both written in the imperative language C with no modularization with respect to the biological structure of the model. These examples seem to be representative of a large portion of today's models: While declarative languages like SBML and CellML are on the rise and are recommended by the COMBINE, most models are still built with imperative and general-purpose languages like MATLAB, C, C++, and FORTRAN (Clerx et al. 2016; Mulugeta et al. 2018), and most SBML models in the BioModels database are monolithic (Cooling et al. 2010).

I have shown that even a model as simple as the Hodgkin-Huxley model of the squid giant axon can benefit from a declarative and modular model structure that pays more attention to understandability and extensibility. However, one can also argue that such a complex code structure would be simply overkill for experienced systems biologists, who are already familiar with all kinds of mathematical representations but not with software engineering concepts. This is certainly true for the Hodgkin-Huxley equations and probably also for many other small- to medium-sized models. Researchers would not use the classical approach to mathematical modeling so often, if it was not working for their use cases.

However, this argument looses ground with increasing model size and complexity. Multi-scale and especially multi-level models consist of multiple heterogeneous parts that are typically reused from existing models, which can be the small- and medium-sized models described before. Even the AVN cells in the Inada model, which mostly only contain equations at the cell level, already are composed of individual currents that are reused from three different preexisting models (Inada et al. 2009; Lindblad et al. 1996; Zhang et al. 2000; Kurata et al.

2002). Figure 1 in section 8.3 shows that when one follows the full chain of references required to understand the model, it turns out that it contains reused parts of at least nine other articles. If these individual models are not built with reuse in mind and researchers have to sift through appendices with dozens of possibly erroneous equations or large files of undocumented imperative code, this is a considerable hurdle to building a multi-level model based on them.

To facilitate the development of large multi-scale models, the central focus of model design therefore has to shift towards reusability. In order to reuse a model, researchers from different working groups must be able to download the model code and run simulations, thus reproducing the methods of the original study. However, reuse typically goes beyond a one-to-one reproduction of published methods and instead also involves different tools in different contexts and with different goals. For example, Inada et al. reused the equations for the fast sodium channel and the inward rectifier channel from the atrial cell model of Lindblad et al. (1996) for their model of an AVN cell and Seidel used a model by Warner (1958), which describe the baroreceptor response in dogs, for a model of the human heart. In both cases this is not a direct reproduction of methods, but an indirect reproduction of the results of the underlying article.

Reuse and reproduction of this kind requires a detailed understanding of the model equations and the assumptions that they carry explicitly or implicitly. For example, Lindblad et al. (1996) have variable intracellular sodium and potassium concentrations, while Inada et al. (2009) assume that these concentrations remain constant. Other examples include biochemical models where either simplified Michaelis-Menten kinetics can be assumed or more detailed kinetic laws can be used instead (Chou and Voit 2009; Hill, Waightm, and Bardsley 1977). As a model is by definition a simplification of reality, no model is free from such assumptions and for each biological system exist different sets of assumptions that are valid for different simulation scenarios. It is therefore unlikely that reusing a model in a different scenario is possible by just copying and pasting model code and renaming variables. Some understanding of the structure and semantics of the equations will be required, even if just to identify which parts of an equation belong to the part that is being reused and which ones do not.

Even the simulation results of an otherwise understandable model might not be easily reproducible in another context if the model structure is not somewhat tailored towards extensibility. A monolithic code structure, in which all model parts are firmly entangled with each other, may be made understandable with appropriate documentation. However, it will still require a lot of effort to identify which parts of the model need to change if it is to be extended or to be broken apart for reuse in a different context. One example for this are my two Modelica versions of the cardiac conduction system within the SHM: Each parameter and variable of the first, monolithic implementation was documented as precisely as possible, yet still the PVC extension was only possible with the second, modular implementation that was built with the goal to facilitate such extensions.

In summary, multi-scale modeling requires model reuse, which in turn requires results reproducibility, understandability and extensibility. Classical approaches using imperative languages to build monolithic structures tailored to a single experiment are not fit to provide these requirements.

3.1.2. MoDROGH characteristics

Some shortcomings of the classical approach to mathematical modeling are closely tied to language and design aspects. This includes the imperative nature of MATLAB, C, C++ and FORTRAN and the lack of modularity in published model code. The MoDROGH characteristics, which are explained in detail in section 8.1, summarize language features that, if applied consistently, can fulfill the reusability requirements established in the previous section. Here, I will only briefly introduce the characteristics and discuss their importance with respect to my own modeling tasks, assessing where they were most effective.

Modular

A language that is modular provides support for defining models as composition of small self-contained components with clearly defined minimal interfaces. There is a broad consensus in software engineering that large monolithic structures tend to hinder reusability, understandability, and extensibility (Sarkar et al. 2009; Clark and Baldwin 2000). If they are carefully designed and well documented, they can work fine for small- to medium-sized projects, but generally speaking it is preferable when code is split into clearly defined sections that each only address one specific concern.

My first Modelica implementation of the cardiac conduction system of the SHM utilized all MoDROGH characteristics apart from being graphical and modular. The missing modularity alone was the main reason that hindered its reuse and extension with a trigger for PVC. The modular version of the model was not only more extensible but also more understandable to the point that it revealed small inconsistencies in the structure of the SHM. In fact, I would argue that one of the main advantages of modularity in the context of systems biology is the fact that modularizing a model both requires and facilitates a deeper understanding of the modeled system.

An extreme case of this is the handling of the intracellular calcium concentration by the sarcoplasmic reticulum in the Inada model. Inada et al. (2009) only present this part of the model in monolithic form, not further discussing any of the involved parts. This part could be split into two diffusion reactions, the calcium uptake by the SERCA pump and calcium release by ryanodine receptors, but it was neither possible to modularize the system without

understanding it, nor was it possible to understand it without separating the entangled equations into their modular form. The reason for this is that in a monolith, only the top level needs to mean anything to the reader. In contrast, a modular structure describes the system at a lower organizational level and thus also must assign meaning to entities on this organizational level—if only by naming the individual modules.[10]

Declarative

While imperative languages describe sequences of actions, declarative languages just define properties of the desired result without specifying how this result is to be reached (J. W. Lloyd 1994). Essentially they add a layer of abstraction that allows—in the context of mathematical modeling—to focus on *what* is modeled instead of *how* to solve equations. This adds a lot of flexibility for reuse in a different context that is otherwise missing with imperative implementations.

For example, the original C implementation of the SHM was entirely imperative in nature with a hard-coded main loop following a fourth order Runge-Kutta algorithm. The description of the cardiac conduction system was scattered throughout this large loop and entangled with the imperative solver logic. This made it quite hard to extract this description and to reproduce it in Modelica. Additionally, it masked mathematical and biological inconsistencies inside the model, such as the fact that the trigger for the refractory period of the sinus node was actually not checked when a sinus signal was produced but only after the signal had already reached the ventricles. The benefit of declarativeness is therefore twofold: First, it allows for flexibility in choosing solvers and changing other aspects of the simulation protocol. Secondly, more focus on the mathematical structures means that they can be checked more strictly by the compiler, which gives additional guidance for understandable, analyzable, and robust model design.

Human-readable

Every modeling language is human-readable to some degree, but while languages like SBML and CellML are designed as intermediate languages, which are used for model exchange and must be transformed into some other representation for convenient editing (Dräger et al.

[10] This part of the InaMo implementation also showcases the importance of choosing the interfaces between modules deliberately, which is an important part of modularization: A first version of the modular calcium handling used concentrations and concentration gradients as main interface variables. This was consistent with the equation structure presented in Inada et al. (2009), and it simplified some equations in monolithic form, but in the modular version it required additional conversion terms and, most importantly, interfered with the model semantics. Some equations in the model were implicitly based on the conservation of mass between two or more connected compartments containing calcium ions. However, this conservation of mass can only be explicitly expressed using substance amounts, because there is no conservation law governing concentrations. After changing the interface to substance amounts accordingly, the model became both simpler and more intuitive.

2009; A. K. Miller et al. 2010), other languages like Antimony and Modelica are top-level languages directly designed to be read and written by humans in text form (Choi et al. 2018; Mattsson and Elmqvist 1997).

The PMR contained an implementation of Inada et al. (2009) in CellML (C. M. Lloyd 2009). For the first few months, this was the only reference code available for my implementation of InaMo. For a researcher without previous experience with CellML tools, this was difficult, because it required both learning to use a new tool and getting familiar with an unknown model. The raw XML code also offered no alternative route of understanding the model directly, because the equations in MathML (ISO/IEC 2016) were too convoluted to be of use. What was needed was a textual representation of the model that could be opened in a text editor to quickly find the relevant pieces with a keyword search.

Fortunately, this was possible with the CellML Text representation provided by the tool OpenCOR (Garny and P. J. Hunter 2015). In other tools, learning the code representation and search features of the tool would be required and one could never be fully sure if every detail was displayed or if the tool abstracted away or could not interpret some information contained within the model file. Since the CellML model did not reproduce any plots from Inada et al. (2009) but was advertised to do so in the documentation, I wanted to look at the version history to see if anything might have changed since the time when the reproduction was performed. Again, this was difficult, because large but meaningless structural changes due to automatic processing of XML code can obscure the relevant semantic edits made by a human researcher.

In short, one can say that for maximum understandability of model code, there should be as little automatic processing steps between the writer and reader of a model file as possible. Ideally, the language is designed for direct code editing in a text editor, which also facilitates literate coding practices proposed by other researchers (Lewis et al. 2016; Waltemath and Wolkenhauer 2016; Medley et al. 2018; Hellerstein et al. 2019). Languages with more focus on machine-readability like XML can still be used for model distribution, but they should not be the single medium for making the source code of a model available to other researchers. The example of CellML Text shows that it is possible to use a human-readable source language for viewing and editing, which is then translated into a more machine-readable distribution language for simulation with a range of compatible tools.

Even within a language designed to be directly read and written by humans, however, code can be confusing if it is not documented. Virtually all languages provide some form of comments for code documentation, but some additionally allow structured documentation in the form of human-readable labels that are part of the syntax and can be assigned to individual variables, subcomponents, or equations (see for example Kramer 1999). Following the argument made in the discussion of the modular characteristic, such structured documentation can facilitate understandability by assigning meaning to lower-level components. Additionally, these kinds

of documentation labels persist across multiple representations of the model and allow, for example, automatic generation of HTML or Portable Document Format (PDF) documentation as with MoST.jl.

Open

As in any scientific discipline, proprietary licenses can be a major hurdle to the reproduction of mathematical models and can also stand in the way of other best practices. For example, MATLAB licenses only allow the use of free continuous integration (CI) services since version R2020a and there is no guarantee that future licensing changes will not return to disallowing such use of the software (The MathWorks 2022a). Additionally, research projects frequently require special solutions that are not readily available in state-of-the-art software. Developing dedicated extensions of tools is only possible if this is allowed and supported by the manufacturer or if the tool is published as open source software. For multi-scale models this is especially important since the span over multiple scales can require additional techniques to manage the computational complexity of simulations which do not yet have widespread support in simulation tools (Dada and P. Mendes 2011).

Graphical

As already established, biological systems are complex and full of feedback loops. A purely textual description of a biochemical pathway with dozens or even hundreds of species would probably be confusing to say the least. Unsurprisingly, diagrams that visualize such feedback loops are an integral part of any biology textbook and when the discussion moves to higher organizational layers, such as the cell or organ layer, the individual components in the diagram also tend to be illustrated with drawings rather than just text (Gerstner et al. 2014; Voit 2018). While these diagrams could also be provided as an image file that is completely separate from the code and thus independent of the modeling language, they become more useful the more directly they are coupled to the model code as this facilitates transfer of information between the two representations.

For each of the models I implemented—be it the Hodgkin-Huxley model, the SHM, or the Inada model—I frequently switched between textual and graphical representations of the model in order to understand and analyze its behavior and to find bugs in my implementation. For this task, it was especially important that the diagram was accurate. This can be ensured in a modeling language, when the diagram representation is defined in the form of annotations in the model code itself that define the appearance and placement of model components. Additionally, this type of support for graphical annotations also allows composing models in a graphical fashion by drag and drop. This was especially useful to quickly create small test examples akin to unit tests.

Hybrid

As explained in section 1.2, there are multiple mathematical formalisms that can be used to create a model of a biological system including, for example, ordinary differential equations (ODEs), differential-algebraic equations (DAEs), discrete-event simulation (DES), or frameworks like Petri nets and bond graphs. However, as more and more complex systems are modeled, it is increasingly unlikely that a single formalism can describe the system. As a simple example, consider an organ-level model of the human heart, like the SHM, which requires both a continuous blood pressure curve (an ODE), but also has a discrete switch between systole and diastole (an event in a DES). In the case of the SHM, there is also a stochastic element in the base period of the heart and lung (similar to a SDE[11]), and a delay between the baroreceptors and the ANS (a DDE).

Such a model that uses more than one mathematical formalism is called *hybrid*, and consequently a modeling language is also called *hybrid*, if it supports multiple mathematical modeling formalisms. The aforementioned mix of discrete and continuous model parts is important for many multi-level models as a shift between levels of organizations may require a shift in discreteness (Walpole, Papin, and Peirce 2013). Very low (genetic, molecular), and very high levels (organ, organism, population) may feature discrete state changes and individual entities, while intermediate levels (cell, tissue) tend to be more adequately described by continuous processes.

Another, more indirect requirement for hybridity, is again due to the abundance of feedback loops in biological systems. If a purely ODE model is modularized in form of blocks that have an input and an output, these feedback loops become algebraic loops where the output of a component is fed back as its input, either directly or through some other components (Lee et al. 2015). There are techniques to handle such algebraic loops, but they can pose issues both with the accuracy and the speed of the simulation (Shampine, Reichelt, and Kierzenka 1999; The MathWorks 2022b). A more elegant solution is to support DAEs, especially in implicit form, which can be used to define acausal connections between components via conservation laws that do not require defining whether a variable is an input or an output of a component (Ascher and Petzold 1998). For example, by using Kirchhoff's current law, electrophysiological models like the Hodgkin-Huxley model and the Inada model can be realized as electrical components of a circuit diagram that are even fully compatible with the predefined electrical components in the `Modelica.Electrical` standard library (Salam and Rahman 2018).

[11] Strictly speaking, the stochasticity in the SHM is not explicitly modeled as a continuous stochastic differential equation (SDE) but as random fluctuations introduced on a beat-by-beat basis through a discrete event.

3.2. Modelica as modeling language for systems biology (RQ2)

After establishing the requirements for modeling language in systems biology, the next step is to assess to what extent the language Modelica can actually fulfill them. Since this investigation requires some technical detail, the core language concepts are first introduced in section 3.2.1. Section 3.2.2 follows with a discussion of each of the MoDROGH criteria. To understand the success of Modelica in the engineering domain and thus the benefits it might bring for systems biology, it is also necessary to investigate the Modelica ecosystem and to look at existing biological projects that already have been performed by other researchers. This will be done in sections 3.2.3 and 3.2.4 respectively.

3.2.1. A very brief introduction to object-oriented modeling with Modelica

In order to talk about the features of Modelica, it is necessary to introduce the language on the code level. However, due to the complexity of Modelica I will only be able to scratch the surface in this section. For a far more comprehensive introduction, the interested reader is therefore referred to Michael Tillers excellent e-Book "Modelica by Example" (Tiller 2020). Here, I will only try to give a rough first impression of the coding style and features of Modelica.

A good example to use for this task is the simple predator-prey model that was independently developed by Alfred J. Lotka (Lotka 1910) and Vito Volterra (Volterra 1926b; Volterra 1926a). In this model, two differential equations describe the evolution of the populations of a predator and a prey species on the timescale of years:

$$\frac{\mathrm{d}N_1}{\mathrm{d}t} = \beta_1 N_1 - \delta_1 N_2 N_1 \tag{3.1}$$

$$\frac{\mathrm{d}N_2}{\mathrm{d}t} = \beta_2 N_1 N_2 - \delta_2 N_2 \tag{3.2}$$

The variables denoted with N are population sizes, the β parameters are birth rates, and the δ parameters are death rates. Variables and parameters with the subscript 1 refer to the prey population, while those with subscript 2 refer to the predator population. The β and δ parameters have to be set to concrete numbers that stay fixed during the simulation. The

population variables N_1 and N_2 will change during the simulation, but their initial value must also be given beforehand. A first Modelica implementation of this model might look as follows:

```
model LotkaVolterra "simple predator-prey model"
  Real N1 "size of predator population";
  Real N2 "size of prey population";
  parameter Real beta1 = 0.1 "birth rate of prey";
  parameter Real beta2 = 0.02 "birth rate of predator";
  parameter Real delta1 = 0.02 "death rate of prey";
  parameter Real delta2 = 0.4 "death rate of predator";
equation
  der(N1) = beta1 * N1 - delta1 * N2 * N1;
  der(N2) = beta2 * N1 * N2 - delta2 * N2;
initial equation
  N1 = 10;
  N2 = 10;
end LotkaVolterra;
```

[3.1]

The translation from equations to code is straightforward in this example: Both variables and parameters are prefixed with the data type `Real` and parameters are denoted as such using the **`parameter`** keyword. Equations that are true all the time are placed in a block starting with the **`equation`** keyword and initial conditions, which only hold at $t = 0$, are placed in an **`initial equation`** block. The **`der`**`()` function denotes the time derivative of the variables `N1` and `N2`.

While this model is already fully functional, there are a lot of possible improvements that can, and should, be made:

- Variables and parameters should be renamed with respect to their biological function rather than their mathematical equivalents in equations 3.1 and 3.2, since this would facilitate the goal of the model to explain biological processes.
- The growth rates `beta1` and `delta2` measured in individuals per second are different from the *fractional* growth rates `beta2` and `delta1` measured in individuals per individuals of the other population per second. This should be clarified by adding unit and data type information.
- The unit of N_1 and N_2 is not defined properly. These variables should be given in numbers of individuals, but with the current parameter settings, the minimum for N_2 is 0.7, suggesting that a value of 1 actually means, for example, one thousand individuals.
- To ensure interoperability with other models, time should be measured in seconds and not in years and the parameter values should be changed accordingly while still keeping a secondary time variable in years for plotting purposes.

These details become all the more important, if the model is extended and becomes more complex. Say a third species should be added to the model: a larger predator N_3, which preys on both populations N_1 and N_2. The mathematical representation of the model then becomes:

$$\frac{\mathrm{d}N_1}{\mathrm{d}t} = \beta_1 N_1 - \delta_{1,2} N_2 N_1 - \delta_{1,3} N_3 N_1 \tag{3.3}$$

$$\frac{\mathrm{d}N_2}{\mathrm{d}t} = \beta_{2,1} N_1 N_2 - \delta_{2,3} N_2 N_3 \tag{3.4}$$

$$\frac{\mathrm{d}N_3}{\mathrm{d}t} = \beta_{3,1} N_1 N_3 + \beta_{3,2} N_2 N_3 - \delta_3 N_3 \tag{3.5}$$

While it is possible to extend the model in this fashion, increasing numbers of species will make it more likely that a slip occurs: a N_4 that should have been an N_3, or a missing or duplicated item in a sum. At the very least, it would be easier to do this if one could operate at a higher level of abstraction and just specify species relations like "Wolves eat lynxes" or "It is assumed that hares always have enough food to reproduce at a fixed rate". This becomes possible when taking a closer look at the equations and their composition. Any Lotka-Volterra-like model with an arbitrary number of species will only be composed of three template terms:

- A "birth" template for prey species x that do not themselves act as predators of the form $\beta_x N_x$;
- a "predation" template for interactions between predator y and prey x, which introduces the addend $\beta_{y,x} N_x N_y$ to the predator population and the addend $-\delta_{x,y} N_y N_x$ to the prey population
- and a "death" template of the form $-\delta_y N_y$ for apex predators y, which are not eaten by any other species in the model.

These templates constitute the higher level relations that are needed to facilitate the extension of the model. The remaining core requirement is a way to automate the summation of instances of these templates to form the equations of the system. In Modelica, this can be done by using connectors like the following:

```
connector Species
  LVUnits.PopulationSize amount "number of individuals";
  flow LVUnits.PopulationRate rate "local change to population";
end Species;
```
[3.2]

This code introduces two variables `amount` and `rate` and combines them to a so-called *connector*. Instead of the generic `Real` in the previous example, the variables now have the custom types `PopulationSize` and `PopulationRate` that specify their semantics. These types are defined in a separate package called `LVUnits`:

```
package LVUnits
  type PopulationSize = Real(unit="1", quantity="PopulationSize", min=0)
    "population size in numbers of individuals";

  type PopulationRate = Real(unit="1/s", quantity="PopulationRate")
    "change in population size as individuals per second";

  type GrowthRate = Real(unit="1/s", quantity="GrowthRate", min=0)
    "exponential growth rate of population";

  type FractionalGrowthRate = Real(
      unit="1/s", quantity="FractionalGrowthRate", min=0
    ) "fraction of growth rate contributed per individual of second population";

  constant Real secondsInYear(unit="s/y") = 60 * 60 * 24 * 365;

  function from_perYear "unit transformation from 1/y to 1/s"
    input Real py(unit="1/y");
    output Real ps(unit="1/s");
  algorithm
    ps := py / secondsInYear;
  end from_perYear;
end LVUnits;
```

[3.3]

The types in this class are effectively just normal `Real`s with additional information: the physical unit; a unique string identifying the physical quantity; an explanatory documentation string; and a constraint that some of them must be non-negative (`min=0`), which generates errors when it is violated during a simulation. The function `from_perYear` and the constant `secondsInYear` will be used to handle the timescale conversion in the following. Note that, unlike the previously shown **`equation`** block, the **`algorithm`** block in Modelica functions uses imperative assignment statements akin to general-purpose programming languages (denoted by `:=`) instead of declarative equations (denoted by `=`).

Back to the `Species` connector, the only code that remains unexplained in listing 3.2 is the **`flow`** keyword assigned to the variable `rate`, which is exactly the mechanism that automates the summation of template instances. In a three species Lotka-Volterra model where the species N_1, N_2, and N_3 are hares, lynxes, and wolves, the rate equation for hares may be composed as follows:

```
connect(hares.species, hare_birth.born);
connect(hares.species, lynxes_eat_hares.prey);
connect(hares.species, wolves_eat_hares.prey);
```

[3.4]

The three **`connect`**`()` equations connect four different instances of the `Species` connector across four model components. In general, each of these connectors can itself contain an arbitrary number of variables, and a single **`connect`**`()` equation is translated into individual

equations for all these variables. In this case, we have the two variables `amount` and `rate`, where `rate` is a flow variable, which requires special treatment. After the **`connect()`** equations have been resolved, the model will have the following actual equations:

$$\texttt{hares.species.amount} = \texttt{hare\_birth.born.amount} \tag{3.6}$$

$$\texttt{hares.species.amount} = \texttt{lynxes\_eat\_hares.prey.amount} \tag{3.7}$$

$$\texttt{hares.species.amount} = \texttt{wolves\_eat\_hares.prey.amount} \tag{3.8}$$

$$\begin{aligned} 0 = {} & \texttt{hares.species.rate} \\ & + \texttt{hare\_birth.born.rate} \\ & + \texttt{lynxes\_eat\_hares.prey.rate} \\ & + \texttt{wolves\_eat\_hares.prey.rate} \end{aligned} \tag{3.9}$$

Normal variables in connectors lead to simple equalities as in 3.6–3.8. However, the **`flow`** keyword in listing 3.2 instead introduces a conservation law by collecting all connected variables in a group and generating the single equation 3.9, which ensures that the sum of these variables balances out to zero—similar to Kirchhoff's current law or the conservation of mass and energy in a closed system (Salam and Rahman 2018; Hesthaven 2018). By treating the change in population size as a "flow" of individuals, one can thus define and combine an arbitrary number of components that *produce* individuals like `hare_birth` and components that *consume* individuals like `lynxes_eat_hares` and `wolves_eat_hares`. To make the resulting system of equations solvable, exactly one *reservoir* component `hares` is required, which does not add anything to the flow, but is instead used to capture the net flow. The compiler achieves this by solving 3.9 for `hares.species.rate` and thus collecting the negative sum of local flows introduced at all other components in this variable (hence the term *reservoir*).

This process of introducing zero sums with the **`flow`** keyword is exactly what is needed to build the differential equation 3.3 as a sum of template terms. Currently, equation 3.9 is an algebraic equation that does not yet include any derivative. The derivative comes into play inside the `hares` component, which is an instance of the model `Population`:

```
model Population "population of a species"
  Species species(start=initial_size, fixed=true);
  parameter LVUnits.PopulationSize initial_size = 10000 "initial population size";
equation
  der(species.amount) = -species.rate;
end Population;
```
[3.5]

This component serves as a reservoir for individuals of a species. It establishes the functional relationship between the `rate` and `amount` variables in the `species` connector and allows to set an initial population size with the parameter `initial_size`. The negative sign is needed to keep the intuitive convention that positive local rates in other components will increase the population and negative rates will decrease it. By default, Modelica treats the `start` parameter as a fallback value that can and must be overwritten by an explicit initial equation.[12] The parameter `fixed` changes this behavior so that the value of `start` actually serves as that initial condition (which thus "fixes" the initial value in the sense that it can no longer be overwritten by initial equations). Combining equation 3.9 with the equation introduced by this component finally yields the sum of template terms for equation 3.3:

$$\begin{aligned}\frac{\texttt{dhares.amount}}{\mathrm{d}t} = {} & \texttt{hare\_birth.born.rate} \\ & + \texttt{lynxes\_eat\_hares.prey.rate} \\ & + \texttt{wolves\_eat\_hares.prey.rate}\end{aligned} \tag{3.10}$$

The template addends are defined similarly in the producing and consuming components:

```
model FixedBirth
  Species born;
  parameter LVUnits.GrowthRate birth_rate = LVUnits.from_perYear(0.1);
equation
  born.rate = birth_rate * born.amount;
end FixedBirth;
```
[3.6]

```
model FixedDeath
  Species dying;
  parameter LVUnits.GrowthRate death_rate = LVUnits.from_perYear(0.1);
equation
  dying.rate = -death_rate * dying.amount;
end FixedDeath;
```
[3.7]

```
model Predation
  Species predator;
  Species prey;
  parameter LVUnits.FractionalGrowthRate birth_rate = LVUnits.from_perYear(0.0002);
  parameter LVUnits.FractionalGrowthRate death_rate = LVUnits.from_perYear(0.0002);
equation
  predator.rate = birth_rate * prey.amount * predator.amount;
  prey.rate = -death_rate * predator.amount * prey.amount;
end Predation;
```
[3.8]

These directly correspond to the three birth, death, and predation templates that we identified earlier. The `Species` connectors are named after the role that the species takes within the component. For `FixedBirth` and `FixedDeath` this is not very relevant, but in `Predation` it allows to intuitively distinguish between the `predator` and the `prey` population.

[12] An example of an **`initial equation`** block in Modelica can be seen in listing 3.1 on lines 11–13.

With the template components and the `connect()` equations to combine them, it is now possible to build the full model entirely on a higher level of abstraction:

```
1  model LV3
2    Population hares;
3    Population lynxes;
4    Population wolves;
5    FixedBirth hare_birth;
6    FixedDeath wolf_death;
7    Predation lynxes_eat_hares;
8    Predation wolves_eat_hares;
9    Predation wolves_eat_lynxes;
10   Real time_years(unit="y") = time / LVUnits.seconsInYear "time in years";
11 equation
12   // Hares have always enough food to reproduce at fixed rate
13   connect(hare_birth.born, hares.species);
14   // Lynxes eat hares
15   connect(lynxes_eat_hares.prey, hares.species);
16   connect(lynxes_eat_hares.predator, lynxes.species);
17   // Wolves eat hares
18   connect(wolves_eat_hares.prey, hares.species);
19   connect(wolves_eat_hares.predator, wolves.species);
20   // Wolves eat lynxes
21   connect(wolves_eat_lynxes.prey, lynxes.species);
22   connect(wolves_eat_lynxes.predator, wolves.species);
23   // Wolves die at a fixed rate independent of other populations
24   connect(wolves.species, wolf_death.dying);
25 end LV3;
```

[3.9]

For example, the connect-equation in line 13 corresponds to the aforementioned statement "It is assumed that hares always have enough food to reproduce at a fixed rate.", and lines 15 and 16 constitute the statement "Lynxes eat hares.". This can be made even more intuitive by adding graphical annotations to the model, which define icons for components, place these components on a diagram coordinate system, and represent the connect-equations with lines between components. For the sake of brevity, I will not go into detail about these graphical annotations here, but only show an example how a connection line between the components `hares` and `hares_birth` might look in the code.

```
1  connect(hares.species, hare_birth.born)
2    annotation(Line(points = {{80, -16}, {80, 12}}));
```

[3.10]

These annotations do not have to be written manually, but can be generated via an interactive diagram view with a drag and drop interface in the OpenModelica connection editor OMEdit (Fritzson et al. 2005). An example of a resulting diagram can be seen in figure 3.2.1. Because they are tied to the code constructs they represent, they automatically stay up to date when the code changes. For example, when the above connect-equation is removed without removing the annotation, this results in a syntax error. If an editor like OMEdit is used, it can also be assured that, for example, the positional values of connection lines are updated when the component at one end of the connection is moved.

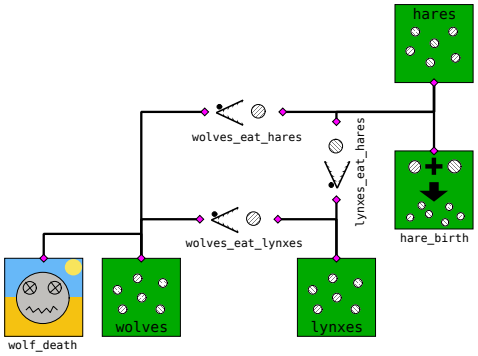


Figure 3.1.: **Graphical representation of the three-species Lotka-Volterra model shown in listing 3.9**. The whole diagram is defined within Modelica code using the Modelica annotation syntax. Icons for components are defined in the component class and include the special template string `%name`, which is replaced with the name of the component upon instantiation. Lines between components are defined by annotating connect equations as shown in listing 3.10.

With this modular and graphical design, the extension of the model by yet another species becomes trivial as one can directly translate the high-level model assumptions one would write down on paper into code. With the graphical annotations it is not even required to write any code. One just selects the appropriate components from a tree view of available models, drags them into the diagram and connects them with lines much like a circuit diagram.

When such a modular model is compiled, the modular structure will be stripped by the Modelica compiler and the connect-equations are transformed to regular mathematical equations leaving an equation system that is identical to the mathematical representation in equations 3.3–3.5 with some additional trivial alias equations of the form $x = y$. In fact, OpenModelica models compile to C code that, albeit containing a lot of boilerplate code, is not that different in structure from the C code written by Seidel or Inada et al. One could say that in this regard Modelica is to C what C is to Assembly: It does not add any more capabilities *per se*, but it facilitates the creation of more complex structures, because the user can operate on a higher level of abstraction and has to care less about the technical details of lower levels.

The main caveat is that while the language does facilitate design patterns that make models reusable and extensible, no modeler is *required* to use these design patterns. In fact, most novices will probably shun the complex features as there is a high initial barrier to understanding them and to applying them effectively. After all, the first monolithic version of

the Lotka-Volterra model presented in this Section was also perfectly valid Modelica code that "gets the job done". For the language Python there exists a neologism that code can be more or less "pythonic", meaning that it is designed with the same goals and ideals in mind that are associated with the language and thus exploits its strengths and avoids its weaknesses (Alexandru et al. 2018). In the same sense, Modelica code can be written more or less "modelica-esque" and one has to understand and accept the mindset behind the design choices of the language in order to utilize its benefits.

3.2.2. Fulfillment of MoDROGH characteristics

Modular

Modelica was designed to support modularity via object orientation. At the core, everything in Modelica is a **`class`**. The special keywords like **`connector`**, **`model`**, **`package`**, or **`function`**, only impose certain restrictions on what can be contained within the class definition. These classes are templates that can be used to compose complex models via *instantiation* and *inheritance* (E. Freeman et al. 2004). Instantiation assigns a name to a copy of the abstract class template to form a concrete object. In the previous example, this would correspond to adding the object `hares` of the `Population` class to a model. While the different `Population` instances `hares`, `lynxes`, and `wolves` share the same code, they represent distinct entities in the equation system.

Inheritance includes the code of one class in the definition of another, more specialized class. In contrast to instantiation, the included code is not assigned a separate name, but instead resides on the same hierarchical level as the rest of the code in the inheriting class. For example, my Modelica implementation of the Hodgkin-Huxley model contained a base class for ion channels in general and two specialized versions for the sodium and the potassium channel respectively. Since Modelica allows *multiple* inheritance, one class can inherit code from multiple other classes, which allows, for example, to inherit both graphical code from an icon class and equations from another class.

Both instantiation and inheritance can be further customized by changing parameter values, redefining parts of the class labeled as **`replaceable`**, or overwriting certain types of equations. With this, almost all conceivable hierarchical structures can be represented with Modelica code, allowing, in theory, to build models of arbitrary complexity.

However, Modelica still has one more mechanism for modularity that adds another level of abstraction around Modelica code. The Functional Mock-up Interface (FMI) allows to encapsulate any Modelica model as a so-called Functional Mock-up Unit (FMU), which acts as an opaque box that defines interface variables and parameters and contains executable C

code (Blochwitz et al. 2012). FMUs can be used in two different ways: In *model exchange*, the FMU acts as a function that computes a single time step and that can be used from within an existing solver. FMUs for *co-simulation* already integrate the numeric solver into the FMU, which just receives inputs and then runs its internal simulation for a specified amount of time. Importing tools can use this functionality to couple multiple FMUs together and thus form another hierarchical layer of abstraction above the capabilities of Modelica itself.

The main selling point of the FMI is that the technology is essentially independent of the modeling language used to define the individual FMUs. One might couple a Modelica FMU created with OpenModelica with one that was created with Dymola and combine those with a third FMU generated from MATLAB code that has nothing to do with Modelica altogether. Implementing tool support just requires handling an XML-based metadata format and plain C code, which is why there are already over 150 tools implementing the FMI standard (Modelica Association 2022g). On the downside, the choice of defining FMUs as opaque boxes means that they have to be used as a whole and do not allow manipulation of internal details.

Declarative

Even with all the modular structure, the basic building blocks of Modelica models still are mathematical variables and equations. This is illustrated by the fact that equations do not have to be written in explicit form. For example, Ohm's law could be written as `i = v / r` but also as `i * r = v` (Salam and Rahman 2018). The order in which equations have to be solved and the variables for which they have to be solved are decided by the Modelica compiler and the differential equation solver, so the modeler does not have to think about such low level issues. As a welcome side effect, Modelica compilers are quite strict with regard to the mathematical correctness of models and are able to flag many design errors that an imperative implementation in C might have readily accepted. The **`flow`** keyword introduced earlier also imposes such additional mathematical constraints in order to make conservation laws explicit and thus avoid breaking them due to numerical errors. Such a feature only makes sense in a declarative mathematical context, and it would be difficult to implement in an imperative language (Ascher and Petzold 1998).

Another declarative aspect of Modelica are unit definitions, which are realized by the two parameters `unit` and `quantity` that can be specified for any variable. While `unit` contains a parsable unit string that can be used for automated unit consistency checks, `quantity` is used to further distinguish between different quantities that have the same unit and also to group together different units that can be used for the same quantity. For models of biological systems, however, more fine-grained semantic identification of variables is required in the form of referencing terms in a standardized ontology like the Systems Biology Ontology (SBO) (Courtot et al. 2011). Since Modelica has an industrial and physical focus, such a construct does not exist in the language. With the present version of the language, it

could be emulated in two ways: Either so-called "vendor-specific" annotations can be used to add information of arbitrary structure to a variable or class, or one could define ontological terms within a dedicated ontology library and then associate classes and variables with these library objects via multiple inheritance. These approaches will be discussed in more detail in section 3.5.4.

Human-readable

Modelica fulfills both major aspects of the human-readable characteristic: It is designed to be directly written by humans in text form without intermediate processing steps, and it has extensive documentation features: Variables, classes, and equations can all have an explanatory documentations string and carry any additional tool-dependent information in structured form via vendor-specific annotations. Classes can additionally include a full HTML document for more detailed explanations. The human-readability of source code is only limited by the fact that the format for graphical annotations is quite verbose and can therefore obscure the actual variables and equations of the model. This can be alleviated by editors that allow to collapse these annotations for better visibility and by outsourcing icon definitions into separate classes that can then be included via inheritance.

As discussed in section 3.1.2, an expressive language also has downsides, the main point being that its complexity makes it harder for tools to directly support the language. In the Modelica ecosystem, this is alleviated by the FMI, which defines a model exchange format based on XML and C code, which is much easier to implement. Modelica models can therefore be distributed both as human-readable source code and as executable FMU, which allows interoperability with other tools and languages.

Open

The language Modelica itself and most widely used libraries are open source. There also exists an open source Modelica compiler named OpenModelica (Fritzson et al. 2005), which also offers a fully fledged IDE called OpenModelica Connection Editor (OMEdit). A second open-source option called JModelica unfortunately recently moved to a proprietary for-profit model (Åkesson, Gäfvert, and Tummescheit 2009; Modelon AB 2022). In principle, this fulfills all requirements for the open characteristic. In practice, however, one has to consider that OMEdit does not offer the same convenience as commercial alternatives like Dymola (Dassault Systèmes 2022). The open source tools are continuously improved and actively developed, but they lag behind in certain less used but nevertheless important features like automatic conversion scripts for backwards compatibility of libraries.

Additionally, most tools—commercial as well as open—do not implement the full extent of the formal Modelica specification, or there are certain corner cases where the specification can be interpreted in different ways. This leads to small inconsistencies between tools that are nevertheless serious enough that it cannot be guaranteed that a model developed in tool A can also be loaded in tool B. For maximum interoperability and openness, the research community therefore has to commit to using open source alternatives or at least ensure that published models are compatible with these open source tools.

Graphical

The Modelica language is built with two equally important model representations in mind. Models can be created, viewed, and edited both as textual code and as graphical diagrams. The lowest hierarchical layer of components must be defined as code, but from there on upwards, both views can be used interchangeably depending on the nature of the task at hand. This is possible, because the graphical annotations are stored directly in the code and tied syntactically to the individual components and equations that they represent. This facilitates a split workflow: Experienced Modelica developers work directly on the code level to create generalized libraries for application domains such as `Modelica.Electrical` for circuit diagrams (Modelica Association 2022e) or `Chemical` for chemical reaction networks (Mateják et al. 2015).

Users of these libraries do not have to concern themselves with the code containing the mathematical details, but only need higher-level domain knowledge to graphically combine the right library components in the diagram view so that the resulting model represents their use case. In a certain way, my implementation of the Hodgkin-Huxley and Inada models followed a similar workflow where I first identified and extracted general "library" components and then used their graphical representation to recreate the high-level biological structure of the modeled system.

Hybrid

Modelica models are differential-algebraic equation (DAE) systems, but they also can contain complex discrete logic like in a discrete-event simulation (DES). In industrial settings, this is important for "cyberphysical" systems that introduce control logic for machines that manipulate processes in the physical world. For biological systems, this can be interesting when crossing from one organizational level to another. For example, the tissue level blood pressure trajectory might best be characterized with ordinary differential equations (ODEs), but at the organ level one might want to study HRV—an entirely discrete phenomenon that requires a discrete beat signal (Seidel 1997).

By supporting DAEs and not just ODEs, Modelica allows an acausal modeling style that is especially suited to represent feedback loops. The support for discrete models is not limited to the re-initialization of continuous variables due to discrete events, but instead variables can themselves be labeled as discrete, allowing to explicitly declare discrete structures of arbitrary complexity. Additionally, DDEs and stochastic terms in events[13] are also supported, the latter through noise generating components in the Modelica Standard Library (Beutlich et al. 2020). This flexibility of Modelica is perhaps best illustrated by the fact that frameworks like Petri nets, bond graphs, finite state automata (FSA), and agent-based modeling can and have been implemented as Modelica library (Proß et al. 2012; Cellier and Nebot 2005b; Modelica Association 2022b; Sanz, Bergero, and Urquia 2018).

Summary

Modelica has an extensive feature set that fulfills all MoDROGH criteria. However, the size of this feature set itself can be a drawback, since it constitutes a high initial barrier for novices that have to understand all these features before they can write or extend Modelica models. This can be alleviated by a strong reliance on standardized libraries, but still these libraries have to be programmed by modelers versed both in Modelica and the application domain, and the application domain needs to be understood well enough to allow such a standardized representation. Additionally, open source Modelica tools lack some convenience features and there is no preexisting support for annotating Models with ontology terms.

3.2.3. The Modelica ecosystem

To understand what distinguishes Modelica from other modeling languages it is also worthwhile to take a look at the Modelica ecosystem. Modelica is characterized by a large and diverse user base with an equal mix of academic and industrial applications centered around a core commitment to open source standards. Regarding the sheer size, the Modelica community is about three times as large as the community of SBML and CellML combined.[14] Its dual focus on research with industrial application is already reflected in the conception of the language based on a PhD thesis of H. Elmqvist (Elmqvist 1978), who then founded the Company DynaSim AB to commercialize and further develop the Modelica precursor Dymola. Version 1.0 of Modelica was designed by a group of 14 people, of which five were affiliated with a university, five with a company, and the remaining four with industry-related research

[13] Note that SDE, which introduce continuous-time stochasticity, are not supported by Modelica.

[14] The International Modelica Conference reported over 400 attendants in 2019 (Modelica Association 2022a) while the COMBINE meeting reported 101 attendants in the same year (Waltemath et al. 2020). A search for the topic keywords “Modelica” vs “SBML” OR “CellML” on Web of Science performed on the 4th of May 2021 reveals a similar ratio of 296 to 126 articles since 2017.

institutes (Mattsson and Elmqvist 1997). Today, the language development is guided by the nonprofit Modelica Association. Two thirds of its institutional members are regular companies, while the other third consists of universities, research institutes and university-related spin-off companies (Modelica Association 2022f).

Perhaps, the best indicator how Modelica benefits from contributions of both sides is the *International Modelica Conference*, which is an academic event that also hosts vendor booths and tutorials (Modelica Association 2022c). Articles from attendants with an industrial background tend to focus towards scalability and special features and libraries that are required in certain industrial applications. Two examples from the domain of electrical power system are a model by Casella et al. (2016), which is among the largest Modelica models to date, and the PowerSystems library, which provides basic components for such models and is one of the largest Modelica libraries (Franke and Wiesmann 2014).

In contrast, researchers with an academic background usually do not have use cases of this size, but contribute more creative and flexible approaches investigating cutting edge experimental features in open source solutions. Examples include PDEModelica (Saldamli 2006) and its spiritual successor PDEModelica1 (Šilar, Ježek, and Kofránek 2018), which define language and compiler extensions to handle PDEs, or parallelization approaches (Braun et al. 2020; Walther et al. 2014). The combination of industrial and academic interests within such a large user base yield a stable environment for continuous advancement of the language and ensure that Modelica will probably remain relevant in the foreseeable future.

3.2.4. Existing biological projects in Modelica

Other researchers have already used Modelica for biological projects in the past. These projects mainly fall into two categories. The first class of articles are large integrated models that have exactly the requirements outlined in section 3.1:

- The lab of Jiří Kofránek works on large, integrated models of human physiology. They translated the diagram-based Guyton model of circulatory regulation into a graphical model using MATLAB/Simulink (Kofránek, Rusz, and Matoušek 2007). However, after comparing Simulink to Modelica, they decided to use Modelica for future projects, because Modelica's acausal connections allow building models whose structure follows the physical structure of the modeled system rather than the calculation process (Kofránek et al. 2008). One of their largest recent projects is the Physiomodel (Mateják and Kofránek 2015), a reimplementation and extension of the HumMod model (Hester et al. 2011), which is a large 5,000 variable model developed in the lab of Thomas Coleman, one of the authors of the Guyton model. Kofránek et al. argue that while

HumMod was the most complete and up-to-date integrated model of human physiology, other researchers still preferred to use older models, because configuring and extending HumMod on the basis of thousands of individual XML files seemed too cumbersome. In contrast, the Physiomodel is based on the Physiolibrary (Mateják et al. 2014), a robust library of standardized and well-documented physiological model components, which can be used to extend and adapt the Physiomodel as well as to build own large- or small-scale models of human physiology. The group continues to build physiological models in Modelica for use in medical and educational scenarios, including, for example, a kidney simulator for e-learning (Šilar et al. 2019).

- Another, similarly ambitious, biological Modelica model is SteatoNet, a 9,000 variable model of the hepatic metabolism including 159 metabolites (Naik, Rozman, and Belič 2014). Unfortunately, the authors do not go into detail about their reasons to choose Modelica over other language candidates. Like the Physiomodel, SteatoNet is also based on a component library called SysBio that is available as a supplement to the article.
- A third, more recent example is the work of Ploch et al. (2019), who present a framework for modeling a biorefinery. Their use case is a pretreatment process called OrganoCat, which makes native biomolecules in plant-derived biomass accessible for transformation. The preprocessed feedstock is then subjected to microbial conversion by Corynebacterium glutamicum, which involves 50 metabolites. They do still use MATLAB to run the compiled simulation and optimization code, but the actual modeling task was performed using Modelica and the visualization tool Omix. Like the Physiomodel and SteatoNet, this model is also based on a custom Modelica library of biochemical unit operations.

The second class of existing projects using Modelica for biological problems is foundational work that aims to enable a workflow for a whole class of modeling tasks with Modelica:

- To some extent, Ploch et al. (2019) can also be considered to be part of this category.
- One of the earlier approaches in this area is the BondLib library for formulating bond graphs in Modelica (Cellier and Nebot 2005b). Bond graphs are a very low-level, graphical formalism for physical models, which are especially suited for Modelica, because they allow to quickly leave the complicated code level and then only operate on the diagram level composing more complex components via drag and drop of the atomic elements. The authors advocated for the use of this formalism in hierarchical modeling of human hemodynamics (Cellier and Nebot 2005a).
- At the same time, Larsdotter Nilsson and Fritzson developed the Modelica libraries BioChem and Metabolic, which provide components for modeling biochemical reactions and metabolic processes (Larsdotter Nilsson and Fritzson 2005a; Larsdotter Nilsson and Fritzson 2005b). More recently, researchers from the Bielefeld University

used Modelica to develop a library for extended hybrid Petri nets, called PNLib (Proß et al. 2012), in order to model biological systems with this formalism. This lead to the development of VANESA (Brinkrolf et al. 2014; Brinkrolf et al. 2018), an IDE for developing and simulating models based either on the Petri net formalism or on biological networks. For Petri net models, VANESA uses the PNLib in conjunction with the OpenModelica compiler.
- Last but not least, Maggioli, Mancini, and Tronci (2020) recently developed SBML2Modelica, a tool that can transform almost any SBML Level 3 version 2 model into human-readable Modelica code.

As a common theme, it can be seen that Modelica is powerful enough to represent other formalisms and even other languages. However, it is seemingly only used for very large projects—maybe, because its expressiveness also makes it quite complicated at the code level.

3.3. Comparing Modelica to other modeling languages (RQ3)

While my investigation started with Modelica, it is by far not the only language that exhibits the MoDROGH characteristics. Section 1.2 already presented a general classification of alternative approaches including MATLAB, SBML and CellML, as well as Python- or Julia-based DSLs. In the following, the most widely used language candidates from each of these categories are compared to Modelica in terms of their fulfillment of the MoDROGH criteria. These sections only highlight key differences. For a more detailed characterization of these languages, the reader is referred to section 8.1 including the article supplement. An additional element that was not part of the investigation in Schölzel et al. (2021a) is the support for DDE and SDE, which is summarized in section 3.3.5. Finally, the individual comparisons lead to a summary in section 3.3.6.

3.3.1. MATLAB/Simulink+Simscape

For a fair comparison between Modelica and MATLAB (The MathWorks 2022c), the Simulink environment (The MathWorks 2022e) and the Simscape language (The MathWorks 2022d) have to be considered. Simulink adds a graphical layer to MATLAB models, allowing drag and drop composition of models, and Simscape allows defining acausal Simulink components that extend the otherwise block-based system to the same level of expressiveness as Modelica.

Recently, full support for export and import of FMUs has been implemented in Simulink (Rouleau 2019). With this setup, the feature sets of MATLAB and Modelica are almost identical. In contrast to MATLAB, Modelica adds more flexibility for modular model design by supporting multiple inheritance and overwriting of parameter values and explicit equations during instantiation and inheritance. In the declarative characteristic, only Modelica offers opportunities for adding ontology support via multiple inheritance or custom structured annotations and in the graphical characteristic, MATLAB only supports bitmaps and not vector graphics.

However, far more important than these differences is the fact that MATLAB is not open in any way. Neither the language, nor the compiler, nor tools and editors are available as open source software, which has severe implications for the accessibility of models and the extensibility of the environment. For example, a project like PDEModelica for MATLAB would only be possible with direct support from the company Mathworks. This lack of openness also affects the human-readable characteristic, since, while Simscape is designed to be read and written by humans, it can only be used within Simulink models, which are stored in a proprietary binary format that is not guaranteed to be backwards compatible. Without a licensed version of the exact same MATLAB and Simulink version that were used to create the model, there is no guarantee that another researcher would even be able to view its code. It is, in fact, quite likely, that models stored in Simulink format will become unusable binary blobs within a few decades, regardless of how well they have been archived.

3.3.2. SBML and CellML

SBML (Keating et al. 2020) and CellML (Clerx et al. 2020) are both state-of-the-art solutions, which are recommended by the COMBINE (Waltemath et al. 2020). While SBML seems to be tailored to the biochemical level on the surface, it supports arbitrary ODE and even DAE models, and the SBML level 3 specification introduces so-called packages, which provide hierarchical composition, and a code-based graphical representation much like Modelica. CellML follows a more general approach with no special constructs for any organizational level. Its feature set is very similar to SBML, but only supports linking external image files as graphical representation without further tying those diagrams to the model structure.

Compared to Modelica, the first obvious difference is that both SBML and CellML have built-in support for the semantic annotation of model parts with ontology terms. Because they originate in the systems biology community, they are much more advanced than Modelica in capturing the biological semantics of a model. On the downside, both SBML and CellML lack in human-readability, because they are not designed as top-level languages, which are directly read and written by humans. Instead, they are designed as exchange formats for a diverse ecosystem of tools that provides graphical interfaces or higher-level textual languages

to edit the model content. The raw model files are XML files with MathML equations. Experienced modelers may understand and even write such files by hand, but for novices they provide a considerable barrier. This is not much of an issue most of the time, because these languages are widely used and researchers need only one of the over 100 tools that support SBML to read an SBML file (Bergmann, Shapiro, and Hucka 2021). However, despite this interoperability, the transfer of a model from one tool to another may hide information of optional or tool-specific SBML constructs that are not understood by the tool used for reading.

Maybe even more important is the fact that while modularity is one of the core principles of Modelica, both SBML and CellML were conceived without modularity in mind and only added capabilities for model composition in later versions. The SBML level 3 package comp, which defines features for the hierarchical composition of models, is neither widely supported by SBML tools[15], nor is it often used in published models[16]. Even with the package, there is no support for full object orientation or any similarly powerful method for modular design. CellML has added support for modular composition in version 1.1 (Cooling, P. Hunter, and Crampin 2008). The implementation of this feature is mandatory for tools, but the capabilities are very basic, only allowing the import of one model file in another and the definition of basic interfaces but not any kind of modification or more complex structural properties. In contrast to SBML, the user adoption rate of modular composition features is higher.[17] Despite their shortcomings in terms of some MoDROGH characteristics, both SBML and CellML currently offer a much more seamless modeling experience than Modelica in the systems biology community, simply because of their wide acceptance in tools and databases.

3.3.3. Python-based solutions

Python is a general-purpose programming language that is widely used in systems biology (Python Software Foundation 2022b; Van Rossum and Drake 2009). Python itself does not fulfill many of the MoDROGH criteria, but there are two general approaches to use Python for mathematical modeling that are worth discussing: General libraries for solving ODE or DAE systems, and packages that define embedded DSLs for specific types of mathematical models.

[15] For example, the SBML Test Suite Database (California Institute of Technology 2022) lists 13 tools that report success on test 000001, which tests a basic SBML model without any optional features, but only 5 of these also report success on test 01127, which includes basic code using the SBML comp package.

[16] Of 812 models published in the BioModels database from 2014 onwards (which is after the publication of the SBML comp package in November 2013), only one actually uses SBML comp in the model code.

[17] The PMR lists 92 projects using CellML version 1.1 and 72 of them include at least one import directive. However, the majority of the 649 models in the database was written in CellML version 1.0, which does not support this feature.

Libraries for solving differential equations typically only provide low-level functions for solving equation systems without much regard to model design, which is why they mostly lack support for modularity, structural documentation, and graphical representation. They are mostly designed as a reliable lower level upon which more user-friendly solutions can be built. Examples for this are SimuPy (Margolis 2017) and PyDSTool (Clewley 2012).

Building on this basic functionality there are several Python packages that define DSLs to facilitate model creation and maintenance. These embedded DSLs are simple human-readable languages that can be used to supply model definitions as a large string in a Python program. They are easier to read and write than the technical definitions required by the basic solver packages, but in order to achieve this simplicity they have to restrict both their expressiveness and the set of models that can be described to a rather narrow focus on a specific domain. Therefore, they lack several features which would be required for large multi-scale models such as, again, support for modularity, structural documentation, and graphical representation. Despite these shortcomings, embedded DSLs have an important advantage with regard to hybrid models: Since they are embedded in a general-purpose language, it becomes easier to couple them with other packages or DSLs that cover other modeling formalisms or domains. Typical examples for Python-based DSLs include PySB (Lopez et al. 2013) which focuses on rule-based reaction models and PySCeS (Olivier, Rohwer, and Hofmeyr 2005), which focuses on cellular systems.

One notable exception to the narrow focus of DSLs is the Tellurium project with its embedded DSL Antimony (Choi et al. 2018; Smith et al. 2009). Antimony retains the simplicity of PySB and PySCeS, but adds sophisticated features for modularity, being one of the two major tools that supported the SBML comp package since its creation in 2013 (SBML.org 2022). It still supports neither a graphical model representation nor structural documentation and the modularity features are similarly limited as those of SBML, but it can serve as a more human-readable top-level language to define large hierarchical SBML models in an understandable way. It also has the advantage that it includes modularity as a core language concept, so that one can be sure that each tool that supports Antimony also supports hierarchical model composition.

3.3.4. Julia-based solutions

Besides Python, Julia is another general-purpose language that has very promising features with regard to mathematical modeling (Bezanson et al. 2017). On the lower level, the package DifferentialEquations.jl (Rackauckas and Nie 2017) provides full support for ODE and DAE including physical units and implicit equations forms. Like SimuPy and PyDSTool, it is too low-level to be used for large multi-scale models, but it can power other solutions such as DSLs.

Julia facilitates the creation of embedded DSLs by providing powerful meta-programming features that allow to extend the syntax of the language. DSL code therefore does not need to be formatted as a string literal, but can be directly written as modified Julia code that seamlessly integrates with other parts of the language. There are two similar projects that use these capabilities to define DSLs for MoDROGH-style mathematical modeling:

The first is Modia (Elmqvist, Henningsson, and Otter 2016), which is a re-implementation of the Modelica syntax within Julia. The Modia project is still in an experimental stage and lacks, for example, graphical capabilities, but it can serve as a proof of concept what could be possible with embedded DSLs in Julia.

The second Julia-based DSL of interest is ModelingToolkit.jl, which takes a very similar approach and whose syntax is also inspired by Modelica (Y. Ma et al. 2021). Unlike Modia, the project is already mature and has a growing user base. Modularity is achieved through a similar object-oriented approach as in Modelica, but additionally all features of the language Julia can also be used for model composition, including multiple dispatch, which can be seen as a more flexible version of the classical object-oriented paradigm (Bezanson et al. 2017). This also has implications for the declarative characteristic, since imperative Julia code can be mixed with the declarative syntax for defining model components. On the one hand, this can lead to less code duplication, but on the other hand extracting a purely declarative description of a ModelingToolkit.jl model without removing any information about the model structure may prove challenging. Regarding openness, ModelingToolkit.jl has the clear advantage that the code for compiling and simulating models is also part of the same environment in which models are defined, which makes it accessible to implement extensions. In consequence, ModelingToolkit.jl already supports more modeling formalisms than Modelica, including, for example, PDEs and SDEs. As an interesting additional feature, it also allows to automatically convert code written for DifferentialEquations.jl, SBML and CellML models into this high-level human-readable syntax. Like Modia, however, it currently does not feature any graphical capabilities.

Moving from the language to the ecosystem, the company Julia Computing also recently started the project JuliaSim (Rackauckas et al. 2021), which aims to enable "industrial scale modeling and simulation" (Julia Computing 2022) in Julia. JuliaSim is not a single package but a full modeling environment more akin to Tellurium. Models are defined using ModelingToolkit.jl or imported from Modelica or other languages using the FMI. The project also includes graphical composition of biochemical pathway models using the Pumas platform (Rackauckas et al. 2020). Unfortunately, while the programming language Julia is open source, the JuliaSim environment is not.

3.3.5. Support for DDE and SDE

The original investigation in Schölzel et al. (2021a) only considered DAE and DES support for the hybrid category, but since the SHM features both DDE and stochastic elements akin to SDE, a satisfactory answer to RQ3 also includes an investigation of these formalisms. Table 3.1 shows an overview of how well these features are supported across the MoDROGH languages presented in this dissertation. As already mentioned, Modelica lacks in support for SDE, since it only supports to add stochastic noise at discrete events. Here, three categories of languages seem to have an advantage over Modelica: PySB and PySCeS both specialize in models at the biochemical level where stochasticity is common; MATLAB covers an even broader range of application areas than Modelica and thus also provides functions for simulating SDE; and both DifferentialEquations.jl and ModelingToolkit.jl can leverage the fact that they are DSLs embedded in a general-purpose programming language with packages that support a wide range of algorithms.

	Modelica	MATLAB	SBML	CellML	PySB	PySCeS	SimuPy	PyDSTool	Antimony	DiffEq.jl	Modia	ModTK.jl
DDE	✓	✓	✓	✗	✗	(✓)	✗	(✓)	(✓)	✓	✓	✗
SDE	(✓)	✓	✓	✗	✓	✓	✗	✗	✗	✓	(✓)	✓

Table 3.1.: **Overview of support for DDE and SDE in MoDROGH languages.** DifferentialEquations.jl and ModelingToolkit.jl are abbreviated to DiffEq.jl and ModTK.jl respectively. A check mark means full support, while a check mark in parentheses means that the feature is only supported for discrete events and not as continuous part of differential equations.

Conversely, rather few languages have the same level of DDE support as Modelica: Again, MATLAB and DifferentialEquations.jl stand out, because they are embedded in a larger environment with diverse application areas. In particular, it is interesting that of the systems biology-specific solutions only SBML fully supports this formalism, although it is clearly required for models like the SHM. As is often the case with SBML, the general support in the language does not mean that all tools also support this feature: PySCeS and Antimony explicitly state that delay expressions in imported SBML models will be discarded.

3.3.6. Summary

It is apparent that there is a major trade-off between feature-rich general-purpose solutions like Modelica, MATLAB, and ModelingToolkit.jl on the one side and systems biology-specific solutions that have lower complexity and barrier of entry on the other side. The advanced features for modular and hierarchical model design of Modelica, MATLAB, and ModelingToolkit.jl currently cannot be matched in any of the other solutions. They provide robust solutions for arbitrary model complexity, demonstrated by the size of already existing models. Of these three candidates, MATLAB has a clear disadvantage with regard to openness. When choosing between Modelica and ModelingToolkit.jl, the question is whether the increased hybridity and extensibility of ModelingToolkit.jl or the comprehensive capabilities for graphical representation and composition of Modelica are more important for the modeling project.

However, leveraging all of these advanced features is only possible with education and experience, and the exact features that allow to handle model complexity might make models *less* understandable for novices. This is especially true when considering that there is currently no standard way to tie biological semantics to Modelica models. While SBML and CellML are less feature-rich, they fit much more seamlessly into existing systems biology workflows and environments. In this conflict, Antimony might be seen as a step in the right direction, as it alleviates some shortcomings of SBML by providing another layer of abstraction *above* the already accepted standards. Maybe a future Antimony-like language could provide the same advanced modularity features as Modelica or ModelingToolkit.jl and still compile to valid SBML for interoperability with other tools.

3.4. Software engineering approaches for challenges in systems biology (RQ4)

My work with Modelica has not only shown benefits of the language in particular, but also benefits associated with general best practices of software engineering, which is in line with the ideas formulated by model engineering (Hellerstein et al. 2019). This section discusses the use of these techniques in my own work and their promise for mathematical modeling in systems biology in general.

3.4.1. Object-oriented software design

In a way, object-orientation can be seen as the idea to find a natural representation of physical and biological structure within a computer program (Kay 2003). It was conceived to promote and simplify code reuse and maintainability of large software projects. Instead of having only a set of procedures passing data around in ways that become increasingly hard to trace with increased program size, the main idea is to couple data and procedures to so-called *objects*, which are independent of each other and only communicate via clearly defined messages (for an introduction to object-oriented programming see Phillips 2018; McLaughlin, Pollice, and West 2007). Both the goals and the biological/physical analogy fit perfectly to the task of multi-scale modeling in systems biology. In my modeling tasks, I could mostly just translate the structure of the biological system directly into the object-oriented structure of the model. At the top level, the SHM consists of objects such as `Baroreceptors`, `Lung`, and `Heart`. Similarly, InaMo includes top-level objects like `ANCell`, or `CurrentClamp`, and one hierarchical step below there are `LipidBilayer`, `InwardRectifier`, or `SodiumPotassiumPump`. An interesting exception is the modular version of the cardiac conduction system of the SHM. Here, the model structure does not separate by physical entities, but rather by different effects of the same biological structure (i.e. tissue in the atrial ventricular node), which leads to object names such as `RefractoryGate`, `Pacemaker`, and `AVConductionDelay`. However, even in this case the object-oriented paradigm was a natural fit to the way I thought and talked about the model with my colleagues. The InaMo example shows that this can be done on multiple levels of organization. It is possible to combine multiple objects to a new higher-level object or to split up the code for one object into several lower-level objects.

Perpendicular to this hierarchical composition, object orientation also allows different levels of abstractions. Each concrete object is an instance of an abstract *class*, which serves as a template for creating objects of a specific type. For example, the SHM contains two objects of the type/class `NeurotransmitterAmount` for the concentrations of Norepinephrine and Acetylcholine. These objects share the exact same code and only differ in their parameter settings and their connections to other objects. Taking this idea one step further, there can be classes that are even more abstract, capturing similarities between different entities in the system. In InaMo, `TransientOutwardChannel`, `InwardRectifier`, and `RapidDelayedRectifierChannel` are all more specific versions of the general class `IonChannelElectric`. The equations and variables that occur in each of these classes are only defined once in `IonChannelElectric` while the other classes import these components via *inheritance*. In contrast to composition, inheritance does not involve another layer but allows combining and reusing code at the same organizational level. This is particularly interesting, if a language allows *multiple inheritance* from different parent classes. In my projects I mostly used this feature to inherit both functionality from a general class and a graphical representation from an icon class, but it would also be possible to combine different types of functionality like defining a `RefractoryPacemaker` that is both a `Pacemaker` and a `RefractoryGate`.

Regardless of the kind of inheritance used, classes in an object-oriented piece of software have to be self-contained: To serve as interchangeable modules, objects cannot access arbitrary internal details of other objects. For example, the `LipidBilayer` in InaMo must be connectable to any kind of current components, be it an ion channel, an ion pump or something else. Therefore, it would be too limiting, if the code in `LipidBilayer` assumed that there was, for example, an activation gate or a sodium concentration at the other side of the connection. Instead, components should only be connected through a minimum number of clearly designated interface variables, forming a well-defined interface. This is required both for maintenance, as a limited number of shared variables allows an easier localization and tracking of errors in the system, and for reuse, as it becomes clear which kind of connections to the outside are supported and required by the class.

There is a debate in the systems biology community that this *encapsulation* principle can be detrimental, because model reuse may happen in unforeseen ways and unusual connections to internal details of a component may be required (Clerx et al. 2016; Neal et al. 2014). For example, a fixed parameter for the intracellular sodium concentration might actually become a variable in the context of the reuse, which requires adding this variable to the interface, since there might be multiple components that need to access this shared concentration value. However, I would argue that it is actually beneficial if such a major change to a model component cannot be done without accessing its code. When properly designed, interfaces capture the assumptions that a model is based on. For example, if the temperature of the cell medium is a fixed parameter, this is a clear indicator that the model may be lacking equations that would be required to capture dynamical changes to that temperature and this is indeed the case in large parts of InaMo. At the very least, such a switch should not be done without carefully examining the internal equations of the model, which is enforced by a clear, and possibly rigid, interface definition.

Taken together, hierarchical composition, inheritance, and interfaces make it possible to handle programs and models of arbitrary size and complexity. This is apparent by huge software projects powered by object-oriented programming languages, such as the Firefox browser with 21 million lines of code (Abadie and Ledru 2020) or Google, who famously uses a single source code repository of 2 billion lines of code for all of its services (Potvin 2015). With Modelica, I have made the same observations for mathematical models: In principle, object-orientation allows to split a model of arbitrary size into small, maintainable components that each have only a few dozen lines of code. However, it also became apparent that mathematical models require some additional features to these core concepts. Examples include the grouping of several interface variables to a single *connector* to reduce the complexity of connections, and the overwriting and deletion of variables and equations during inheritance.

While object orientation is a natural fit to facilitate elegant multi-scale modeling, modern general-purpose programming languages also use other paradigms to handle complexity. Functional programming, which allows using functions themselves as values that can be

passed as arguments to another function like numbers or strings, can complement object orientation, because it "modularizes" computations and data flow. In fact, Modelica does support creating new functions by fixing parameters of an existing function to a default value, and it also allows using functions as values. I used this both in HHmodelica and in InaMo to compose fitting functions used inside ion channel models and to avoid code duplication. With regard to ModelingToolkit.jl, the multiple dispatch mechanism of Julia, which decides which version of a function to use based on the data type of the arguments that are passed to it, might also act as a replacement for the traditional view of object-orientation presented above.

3.4.2. Documentation

In software engineering, there is a consensus that in order to be maintainable and reusable, code must be well-documented (Martraire 2019). Virtually all programming languages allow comments in the form of text that can be added to a source code document and that is ignored by the compiler. This can be used to explain complex and unintuitive lines of code, but it is only useful when looking directly at the relevant line in the source code where an item of interest is defined. However, when exploring an unknown code base or writing own code, one often wants to know the detailed definition of a code item that is not defined but just mentioned in the code one is currently analyzing or writing. Here, structured documentation is required—i.e. documentation that is tied to the code item it explains and that can therefore be automatically retrieved whenever that item is mentioned.

In mathematical modeling, such documentation can, for example, come in the form of short labels for models, parameters, variables, or components. Additionally, a model or component may require some more in-depth discussion in form of a rich-text document, for example in HTML format, that is embedded in the source code. The benefit of such structured documentation over simple comments is that it can enrich the presentation of the model in many formats. Editors and other tools can parse the documentation and display it alongside the item that it documents, export formats like FMI can use it to create summaries of the model content, and it can even be used to automatically create a documentation website (e.g. using a tool like MoST.jl). This is especially important since it can enrich model discovery in databases and web services like BioModels or the PMR. While it is possible to copy parts of normal comments or external documents to achieve these goals, this quickly becomes cumbersome once changes to the code require a change in the documentation. It is better to only have to change a single label in proximity to the changed line of code than to have to remember to update several websites and other external documents, which might lead to oversights.

In my own modeling experience, Modelica's structural documentation features were invaluable in the re-implementation of the Inada model. Here, I constantly had to remember why equations looked as they do, why parameter values in tests differed from the parameter values for the main model, or how my own base classes and interfaces were structured. The documentation of InaMo now contains the result of my research through 9 articles and two other code bases and aims to explain every single parameter setting so that future researchers will not have the same overhead when reusing my code. This thorough explanation can be found in the code itself, in the exported FMU file and in the online documentation of the model, which are all created from a single source.

3.4.3. Version control

Research is always ongoing work that is subject to frequent changes. Mathematical models are no exception. Seidel's and Inada's C code, and the CellML version of the Inada model all showed signs of being edited after their initial publication. For my own models, the version number of each of them changed several times: 10 for SHM, 3 for SHMConduction, 7 for HHmodelica, and 8 for InaMo. Each of these changes included substantial modifications to the models and their tests. This makes it apparent that it is important to track these changes and to document which exact version of a model or script was used in a publication in order to achieve an exact reproduction of its methods.

The software engineering solution for this problem are Version Control Systems (VCSs) like Git (Chacon, Long, and the Git community 2022). These systems can handle arbitrary plain text documents by identifying changed lines and allowing the user to add a meaningful comment to each small change performed on a document. While each of these so-called *commits* already carry a unique identifier, programmers can also use human-readable labels to identify when these small-scale changes constitute a shift from one version of the code to another. Ideally, this should be accompanied by a separate document called a *changelog*, which summarizes the small-scale commit messages to explain which meaningful top-level changes were performed since the last version (Lacan and Fortune 2017). With these instruments, it is easy to keep track of what changed when and why and, in particular, to identify why some test might work with one version of the software but not with the following version.

Especially for the Inada model, I would have welcomed both a coarse-grained changelog and fine-grained commit messages in order to know when and why the Acetylcholine-sensitive channel was added, which experiments were performed using this code, and whether this branch of development was continued and used for the experiments in the article or not. Such information could have saved a lot of time spent on research and testing during the reimplementation.

There is one major caveat with regard to VCSs: They assume that each change in a document is due to a meaningful and conscious decision by a human being. This assumption is violated when documents are not edited directly in a text editor or IDE, but through a graphical user interface, or a definition in a different, higher-level language that is automatically translated. Such automatic translation can introduce structural changes such as a reordering of elements that is ultimately meaningless but will show up as numerous changed lines in the VCS. This is true for SBML and CellML models and, to some extent, also for graphical annotations in Modelica models. When one is not careful, such clutter can obscure actually meaningful changes.

Additionally, the value of the VCS is diminished if it can only show changes on a lower level that the programmer is not actually familiar with. For example, a change in an equation in a SBML model will result in a block of changed MathML lines, which may be quite hard to decipher, because SBML tools usually do not show the raw MathML code, but a human-readable representation. To some extent, this can be alleviated by making the VCS aware of the structure of these documents and providing alternative, high-level views for these changes, but such solutions are not implemented in standard VCS solutions like Git.

3.4.4. Automated testing

The abundance of positive and negative feedback loops in biological systems makes it extremely hard to pinpoint the source of an error in a model. Taking the example of the SHM, if one observes that the blood pressure is too high, the reason for this may be an error in the equations for the blood pressure curve in the heart, or it might be that the equations of the sympathetic system produce an abnormally large activity, or the sensitivity of the baroreceptors to absolute blood pressure might be too low. Without a way to test each of these model parts individually, there is no way to tell which part of the loop is the source for erroneous behavior.

One of the first things that I did when implementing the SHM in Modelica was to just focus on the baroreceptors themselves and then build a small test model that simulated the reaction of the baroreceptors to a predefined blood-pressure curve. This effectively breaks the feedback loop and allows the modeler to observe whether the behavior of this single component is plausible when it is isolated from the rest of the model.

This general concept is called a *unit test* in software engineering (here and in the following see S. Freeman and Pryce 2010). The idea is that in order to narrow down the search for errors, the smallest parts of the system that can still be considered to be self-contained *units* should be tested for a range of different inputs, comparing the actual output to the expected output. By doing this, it is ensured that the failure of such a *unit test* can only mean that the error is

within a single unit, which should only span a few dozen lines of code. For mathematical modeling, unit tests have the additional value that they make it easier to reuse only parts of a model. When these parts are subjected to unit tests, a researcher that wants to reuse them both knows their expected behavior in isolation and has an example of how to extract them from the code base.

A prime example is the `CaHandling` component in InaMo, which I transformed from a monolithic to a modular representation in order to make it more understandable. This part of InaMo had to undergo several major changes, because I had some misconceptions about the model and equation structure at first. To localize errors, I implemented small tests for each of the components such as the SERCA pump, the Ryanodine receptors, and diffusion components. These tests highlighted several accidental errors that I introduced during the modification of the overall `CaHandling` system.

The `CaHandling` system also shows some peculiarities of unit tests in mathematical modeling. Most importantly, defining the expected output of a model is not as easy as determining the expected output of, for example, a function that should sum up its arguments. The output of any simulation is usually a time course of some variables. In order to write a unit test that does more than ensuring that the model compiles successfully, it is therefore preferable to store a reference simulation output along with the model. The test can then be as simple as requiring an exact match within some tolerance between the actual simulation output and the reference.

This principle of comparing the output of a program with the output of a previous version is called a "diversifying" test and if it is used to ensure that modifications do not lead to unforeseen errors (i.e. "regressions"), it is further classified as a *regression test* (Liggesmeyer 2009). It would also be possible to use qualitative descriptions of the expected output like "Variable `a` has a peak of height 12 at time t = 0.5s.", but this would require special testing libraries that support the translation of such coarse concepts to actual mathematical tests on the output data. In fact, storing reference data in the repository even has the added benefit that researchers who want to reproduce the model in a different language can use the very same reference output to test and even quantify the difference between their implementation and the original one. On the downside, one must be very careful when there is a *desired* change in behavior that requires to update the reference data, because it must be ensured by manual inspection that no additional, unwanted changes slip under the radar.

There are, of course, also errors that do not occur in isolation, but only when components are connected to each other. Therefore, the same structures and libraries used for unit tests are also used for so-called *integration tests*, which test the behavior of the whole system in some clearly defined cases (S. Freeman and Pryce 2010). For mathematical modeling, this principle

can be used to test the very same simulations that are performed in the accompanying research article. These tests can even serve as a documentation of the exact experiment protocol used.

3.4.5. Virtualization and continuous integration

Unit, regression, and integration tests are more useful if they are performed often, preferably for each small change to the code, because this allows pinpointing errors to just one such small change. Additionally, even if the tests run on one machine, they may fail on another system because of a different behavior of the operating system, missing dependencies, or missing configuration files. These issues can be addressed by continuous integration (CI) (Shahin, Ali Babar, and L. Zhu 2017). The basic idea of this technique is to increase software reliability by running tests—especially integration tests, hence the name—frequently in an automated pipeline. CI is usually performed in a virtual machine or a container, in order to have a minimalist, clearly defined environment, which also shields the server from errors in the software that is being tested. This means that a CI script has to include the full information that is required to set up the software on a machine that is completely separated from the development environment. It is even possible to simultaneously run tests on different operating systems with different versions of dependencies in order to detect errors that might only occur in a specific setup. This guarantees methods reproducibility and facilitates results reproducibility of any software-based study.

For mathematical modeling, it has the additional benefit that a CI script needs to contain all the information that is required to set up the environment and to run simulations. If Inada et al. published their model code publicly with an accompanying CI script, or if the CellML Model Repository performed automated tests on published models, most if not all errors that I encountered in my re-implementation could probably have been fixed by the authors before publication.

Even when one does not consider reusability, CI also helps during the development process. I noticed that, due to the abundance of feedback loops in models of biological systems, changing one part of the model could make virtually any other part fail. To make matters worse, I would often only notice these errors much later in the development after I had made several other changes, making it quite hard to pinpoint the defective lines of code. To manage these unforeseen interactions, I needed to run the full test suite for all model parts after each change to the code. Doing this manually was cumbersome and slowed down my workflow, so I implemented an automated CI pipeline. Looking back, I would estimate that even creating the pipeline from scratch with limited prior knowledge about CI tools still resulted in an overall reduction of the development time, because it cut down on debugging time.

Regarding reusability, the CI scripts did not only guarantee that my methods were reproducible with the OpenModelica versions, which I was using at the time of publication, but they also helped me realize that there were new issues with OpenModelica version 1.15 and onward due to changes in the compiler logic. By fixing this issues I could allow other researchers to reuse my models with newer OpenModelica versions up to version 1.17.

The same workflow systems used for CI can also be used to automate the release of software artifacts, which is called continuous deployment (CD). For example, each version of my implementation of the SHM, HHmodelica, and InaMo, also includes a FMU of the main model of the respective project in order to keep the barrier for other researchers to run their own simulations with the model as low as possible. This FMU is automatically generated whenever a tag with a new version is pushed to the Git repository. Similarly, HTML documentation generated by MoST.jl is also updated automatically for each new model version.

3.4.6. Long-term archiving of code

Even if the methods of software-based research projects are reproducible at the time of publication, this is not a guarantee that it will stay reproducible in the following decades. If the software is published on a private or institute website by the authors, limited funding and short-term contracts may mean that the maintaining researchers leave the institution after a few years and take up other projects, abandoning the website maintenance for time reasons. A more sustainable approach is to upload software artifacts to a journal website along with the corresponding research article. However, still, journal websites and databases are subject to changes that might, as in the case of the Inada model, bury these artifacts, rendering them inaccessible for other researchers over time. It becomes clear that the long-term archiving of code is a nontrivial task that requires dedicated solutions. There are already many interesting approaches of which I want to highlight a few:

Dryad is a general research data repository for any field and any data format based on open-source software (Cruse et al. 2022; White et al. 2008). It is manually curated to ensure data quality and archived through the Merritt Repository of the University of California Curation Center (UC3) (The Regents of the University of California 2022b). At the top level, Dryad provides citable Document Object Identifiers (DOIs) for entries and works together with publishers such as Wiley, The Royal Society, and PLOS to cross-link published data to the corresponding article and check compliance with journal requirements.

BioModels was originally designed as a database for SBML models at the European Bioinformatics Institute of the European Molecular Biology Laboratory (EMBL-EBI), but now accepts all model formats (Malik-Sheriff et al. 2019). Like Dryad, it is in part

manually curated with strict guidelines for SBML models, although curation is not available for non-SBML models. Both curated and non-curated models are always linked to a publication. BioModels is an ELIXIR Deposition Database, which means that it meets certain technical quality and governance criteria and aligns with the FAIR principles (Durinx et al. 2017). However, I am not aware of a long-term archiving strategy employed by BioModels. On the surface-level, BioModels does not use DOIs but their own unique model identifiers. The BioModels website offers a rich set of features for model exploration using semantic metadata.

Zenodo is a service for storing and publishing all kinds of research data, and is hosted at the CERN Data Centre (Cern Data Centre 2022). Like Dryad, it is based on open-source software, and it provides citable DOIs for entries. Uploads are not curated and therefore integration with journals is only possible in a more loosely fashion. As part of the CERN Data Centre, Zenodo follows high standards for long-term storage.

The GitHub Archive Program is a long-term archiving service for software hosted on GitHub (GitHub 2022b). It employs a pace-layers strategy (Brand 2018) with a hierarchy of backup services from low-latency short-term solutions to slower long-term solutions. On the extreme end, the GitHub Arctic Code Vault is designed to store software artifacts for at least 1000 years. The GitHub Archive Program does not use manual curation or separate identifiers like DOIs, but it ensures that source code repositories hosted on GitHub will be available for future uses for as long as possible without any need for manual intervention by the repository owner. On the downside, this system is only designed for source code and other text documents and not large volumes of binary data.

From this selection it becomes apparent that there are many approaches and that there are trade-offs between them. Zenodo and Dryad are well-suited for recordings of *wet lab* or *in silico* experiments, while GitHub is a good choice for model source code, and finally BioModels is especially interesting for SBML models. For my own models, I chose to combine multiple approaches to achieve optimal reliability and accessibility. The GitHub Archive Program ensures the long-term storage of my models, while Zenodo makes individual versions of the models citable using a DOI, and BioModels makes them discoverable through the FAIR principles.

3.5. Required tools and language improvements for mathematical models of biological systems (RQ5)

During my dissertation I encountered many areas where the modeling process with open-source Modelica tools could be improved either in general or with specific regard to multi-scale models in systems biology. For most of these issues, I drafted solutions that I either realized myself or by supervising a bachelor's or master's thesis. This section will give an overview both of the issues and the suggested solutions.

3.5.1. Mo|E

When I first started working with Modelica, the OpenModelica IDE OMEdit (version 1.9) was still missing a lot of convenience features and did not fit very well in my software engineering workflow. One of the major concerns was that it focused very much on graphical model composition over model design on the code level. Among other inconveniences, comments in the code could be lost or moved around the document, and purely structural changes like indentation could be discarded or rearranged when the model was saved. As mentioned in section 3.4.3, this creates major issues with VCSs, because it can obscure meaningful changes in a large chunk of meaningless artifacts due to restructuring. The focus on the graphical composition also meant that model files could only be saved when they were fully syntactically correct, because otherwise they could not be loaded for editing again.

Additionally, OMEdit did not support advanced code editing features like, for example, code completions, automatic detection of type errors or references to non-existing variables, multiple cursors, VCS integration, or a search engine for editor commands that is accessible from a keyboard shortcut. Structured text editors like Atom (GitHub 2022a), Visual Studio Code (Microsoft 2022b), and Sublime Text (Sublime HQ 2022), provide most of these features out of the box, but for error reporting and code completion they require language-aware plugins. Mo|E provides such a plugin for Atom and allows the easy definition of plugins for other editors. It was the Bachelor's thesis of Nicola Justus, which I supervised.

Mr Justus drew inspiration from the ENSIME project (ENSIME contributors 2022), which provides a language server for the language Scala that decouples language-aware code from editor plugins, which only have to implement the simple HTTP-based protocol to communicate with the server process. Mo|E takes the same approach, using a custom error-tolerant Modelica parser in the server to also provide code completions in a document that is not fully syntactically correct. In general, it removes the extra automatic processing step between writer and reader of model code that is introduced in OMEdit, ensuring that the

code stays exactly as it was written and allowing fine-grained manual structuring to make the code itself as understandable as possible. It also allows applying a multitude of software engineering tools like VCS, multi-cursor editing or static code analysis, which allows the design of Modelica models in a modern software engineering workflow.

3.5.2. MoVE and MoNK

Even though OMEdit was and is focused on graphical composition of models, the vector graphics editor is another weak point of the software. In version 1.9, there was, for example, no way to undo changes, and it was impossible to select an element that was situated "behind" a transparent element. In version 1.18, there is currently still no support for bringing an element to the front or moving it to the back of the graphical stack. This situation is unfortunate since vector graphics have clear advantages in a scientific setup where model icons will not only be viewed inside the IDE, but also in varying sizes in PDF documents, on websites, and on posters.

In order to facilitate and promote the use of Modelica's vector graphics feature, I wanted to allow Modelica developers to edit Modelica vector graphics with the same convenience that is provided by professional tools such as Inkscape (Inkscape developers 2022) or Adobe Illustrator (Adobe 2022). In general, there are two approaches for this: Developing a standalone vector-graphics editor for Modelica models; or translating an SVG document created by another tool into Modelica syntax. With MoVE, I first pursued the first approach, because the Modelica vector graphics format is incompatible with SVG. Like Mo|E, this project was performed by Nicola Justus and supervised by me as preparation for Mr Justus' Bachelor's thesis. MoVE did provide a rudimentary implementation of the features that were missing in OMEdit and improved on usability and convenience. However, it turned out that, even with the limited amount of primitives provided by the Modelica syntax, making the features stable enough to provide the same convenience as professional tools required a great amount of effort, because there were a lot of corner cases to consider.

Since my own time for maintaining the project was quite limited, I decided to no longer develop MoVE, but instead try the second approach, translating SVG documents to Modelica syntax. The resulting Inkscape plugin MoNK does neither support all SVG primitives nor can it create every possible Modelica primitive, but there is a sufficiently large common ground between the two formats that allows for satisfactory results in almost all cases, requiring only minimal manual adjustment of the resulting Modelica code. The most noticeable downsides include difficulties in handling smoothed paths and polygons, text placement, Modelica fill patterns, and SVG elements that have no direct equivalent in Modelica. On the upside, however, MoNK allows to shift the major part of Modelica icon design to professional tools offering a lot of design features that would be impossible to re-implement in an IDE like

OMEdit or a standalone editor like MoVE without a large development team dedicated only to this feature. Although it is designed as Inkscape plugin, the MoNK script can also be used to translate SVGs from other sources like Adobe Illustrator, or preexisting icons from online databases.

3.5.3. MoST.jl

Defining unit tests for mathematical models in general and Modelica models in particular is not straightforward with existing tools. General-purpose unit testing frameworks in programming languages like Python or Julia are mostly focused on testing whether the output of a single function call matches an expected result and not on comparing trajectories of continuous variables through time (Python Software Foundation 2022a; Bezanson et al. 2022). Specific Modelica libraries for defining unit tests exist, but they are either part of proprietary tools, in an early stage like XogenyTest (Elsheikh et al. 2022), or discontinued like MoUnit (Samlaus et al. 2014). Some large Modelica projects like OpenModelica (Sjölund, A. Pop, and perost 2022) and the BuildingsPy library (The Regents of the University of California 2022a) include custom libraries for regression tests that do not require changing the code of Modelica models themselves. These libraries are tailored to the specific projects and cannot easily be extracted from their environment. Additionally, they do not provide concise top-level summaries of test success or failure like comparable libraries for general-purpose programming languages.

Due to these shortcomings, I decided to implement my own test script using the libraries OMJulia.jl (Kumar et al. 2022), which allows connecting to the OpenModelica Compiler (OMC) from a Julia script, and the built-in unit testing features of the Julia standard library. During the development of my Modelica models, this script grew into the library MoST.jl, which aims to make thorough testing of Modelica models as easy as possible for the end user. Consider the following example code:

```julia
using ModelicaScriptingTools
using Test

withOMC("test/out", "test/res") do omc
    installAndLoad(omc, "Modelica"; version="3.2.3")
    @testset "Example" begin
        testmodel(omc, "Example"; refdir="test/regRefData")
    end
end
```

[3.11]

This script installs and loads the required version of the Modelica Standard Library (MSL) if required and possible, loads the model `Example` from the library folder `test/res` checking it for any kind of errors and reporting them in a human-readable way. Then it reads the simulation settings from the model annotation—including a custom vendor-specific annota-

tion, which can be used to filter relevant variables—and performs the simulation, writing the output to the folder `test/out`. Additionally, if a file with corresponding name exists in the directory `test/regRefData`, a regression test is performed, checking for missing variables, varying output lengths, and differences above the default relative tolerance of 10^{-6} between the output and reference data after both have been resampled to the same size. From the user's point of view all this work just requires a few lines of code, which can be configured to their liking.

To facilitate running unit tests in a CI workflow, I also implemented the GitHub action `setup-openmodelica` (Schölzel 2022b). With this action, the user only has to specify which version of OpenModelica they would like to install instead of going through the slightly complicated process of adding a package repository and its authentication keys to the package manager `apt`.

Apart from providing an easy-to-use unit testing library, MoST.jl also attempts to solve a different problem. While hierarchical composition facilitates model development and reuse, it can make it hard to get an overview of the actual equations that are used in a model, because the user has to traverse through all intermediary levels to reach the actual mathematical representation at the lowest code level. This can be solved in two ways: First, one can facilitate quick transitions through the model hierarchy, for example by providing an HTML-based documentation where the user can jump from model to model using hyperlinks.

Secondly, it would be helpful to be able to generate a mathematical summary of the whole model that is independent of the modeling language, much like the lists of parameters and equations encountered in journal articles. If this second option can be automated, it could also help to reduce human error in published model equations. MoST.jl attempts to provide both solutions by extending the Julia library Documenter.jl (Piibeleht, Hatherly, and Ekre 2022). Documenter.jl allows writing documentation for Julia packages using an extended version of Markdown that can, among other things, contain statements that import documentation strings from Julia functions. The extensions provided by MoST.jl allow the same functionality for Modelica models. For example, a documentation of the main model of the HHmodelica project might look as follows:

````
## Full modular model

```@modelica
%outdir=../out
HHmodelica.CompleteModels.HHmodular
```
````

[3.12]

The special language code `@modelica` tells Documenter.jl to not format the code block as such, but instead output a model summary defined by MoST.jl. This summary contains...

- the rendered HTML documentation string found in the annotation of `HHmodular`,
- the full model code,
- all equations of the DAE system defined by the model, grouped by the model hierarchy and cleaned of alias variables, introduced due to connect equations,
- a list of all functions that are called in the equation list, avoiding duplications due to parameter changes,
- and a list of all variables and parameters involved in the DAE system, including units, parameter values, and labels.

The procedure to create this summary is quite involved and error-prone, due to the amount of cleanup and simplification that is required to make the raw compiler output human-readable, but the first experimental results are promising. It might turn out that in order to solve this robustly, the simplification features have to be integrated into the OMC itself, but since this makes models much more approachable both for novices and experts, it might be worth to further pursue this route. An example can be seen at `https://cschoel.github.io/hh-modelica/dev/`.

3.5.4. Ontology support

As mentioned in section 3.2.2, one of the main shortcomings of Modelica with regard to the use in systems biology is the lack of support for annotating models and their components with ontology terms. There are two possible ways to remedy this that I want to outline here.

The first approach uses so-called vendor-specific annotations. These can be added to any model or model component and take the form `__VendorName(key1=value1, key2=value2, ...)`. Values for keys can be arbitrary Modelica data including Arrays, Objects, and primitive numbers, Booleans, or strings. With this feature, it would be easy to just add something like the following to a model of the human heart:

```
model Heart
  ...
  annotation(
    __ModelicaOntologyExtension(
      file="HUPSON.owl",
      id="http://sig.uw.edu/fma#Heart",
      label="Heart"
    )
  );
end Heart;
```

[3.13]

The benefit of this route is that vendor-specific annotations are a very powerful tool to add data of almost any shape to almost any part of a model. On the downside, tools need to support this kind of information in order to make it valuable for a user of the model who does not want to look directly into the source code to uniquely identify model components. Systems biology researchers would either have to implement their own Modelica-based tools, or work together with open source tools like OpenModelica to add support for such a feature into the existing tools.

The second approach therefore attempts to leverage existing Modelica features as much as possible. Since Modelica supports multiple inheritance, the **extends** keyword can be exploited to import ontology information into a model as part of the type system. This requires that the Ontology is transformed into a Modelica library that does not provide any functionality, but only the desired unique labeling. To illustrate this idea, consider the following example code using the human physiology simulation ontology (HuPSON) (Gündel et al. 2013):

```
package HUPSON
  model heart "heart"
    parameter HUPSONInfo hupson(
      definition= "A hollow organ located slightly to the left ..."
      id= "http://sig.uw.edu/fma#Heart",
      synonyms= {"hearts", "cardiac"},
      subClassOf= HUPSON.anatomical_part,
      ...
    );
  end heart;
end HUPSON;

model Heart
  extends HUPSON.heart;
  ...
end Heart;
```

[3.14]

Using such a structure, existing Modelica tools will display the information that `HUPSON.heart` is a base class of the `Heart` model. Moreover, this also means that `Heart` objects have a regular Modelica parameter called `hupson`, which contains a record with all ontology-related information attached to the *heart* term in HuPSON, including the unique identifier of the term. The content of this parameter can be inspected with the same tools used to inspect other Modelica parameters. Links to other ontology terms can be implemented by actually referencing them in the `hupson` parameter. It is likely that such a translation of ontologies to Modelica libraries can even be automated. The only remaining downsides in comparison with the approach using vendor-specific annotations are that it is not easily possible to annotate *equations* in this way and that by adding `hupson` objects as actual parts of the model, unnecessary clutter is added to the model that might slow down compilation and simulation.

3.5.5. Modelica features in COMBINE languages

There is one remaining downside of Modelica that is hard to overcome: The systems biology community has already decided on using SBML and CellML as standard languages. Both are suitable in general, but lack some of the features of Modelica, most notably its support for object orientation. Another approach, which may even be undertaken in parallel to a gradual adoption of Modelica for very large models, could be to extend these languages with the necessary features. This would have to cover three major areas:

1. Graphical model representation must be promoted and supported more widely. As far as I can tell, SBML's support seems sufficient on the language level, although the separation between the layout and rendering package might be confusing for users if it is not abstracted away properly by editing and viewing tools. To avoid the Modelica situation where vector graphics are possible but cumbersome to use, an alternative approach might also be to just embed a SVG image in a SBML/CellML model (which should be easy since all these formats are XML files) and require that each top-level element in the SVG is connected semantically to a component or connection term of the model.
2. Human-readability and better version control could be achieved by adding another layer of human-readable languages on top of SBML/CellML. This process has already started with Antimony and CellML Text, but both formats are currently mostly used as editing help within other tools and environments. Instead, they should be used as primary, standalone source files, which can be kept under version control and from which SBML and CellML code can be generated on the fly.
   In order to do this, both languages need to be extended with additional mechanisms that represent the full feature set of SBML/CellML. Here, Modelica might serve as inspiration how complicated modeling features can be implemented in a human-readable syntax.
3. Advanced modularity support might be the highest barrier to build large-scale models in SBML/CellML, because in both languages this would require substantial extensions to the core language constructs. As outlined in section 3.4.1, I believe that the object-oriented paradigm—either in its classical form or as the multiple dispatch variant used by Julia—is perfectly suited to solve the scaling- and reusability-related problems of large multi-scale models. There are other concepts with similar power, such as functional programming, but I am not aware of any projects that use those to implement models of a similar size as Modelica.
   Therefore, I think that both SBML and CellML should strive towards supporting full object orientation, including (multiple) inheritance, hierarchical composition, and interfaces with variable grouping. Future versions of SBML should also incorporate this feature as a core language concept, not just as an optional add-on.

There are probably other, more specialized areas where language design in Modelica could inspire language design in SBML and CellML and maybe also vice versa. In any way I believe that a collaboration between language developers and communities of both sides would be beneficial for scientific progress in mathematical modeling as a whole.

4. Conclusion

Summing up the answers to the initial research questions, it can be said that modeling languages for systems biology should be MOdular, Declarative, human-Readable, Open, Graphical, and Hybrid (MoDROGH) (RQ1) to facilitate reusability and that Modelica does fulfill all of these criteria with small concessions regarding openness (RQ2). Other MoDROGH languages either lack in modularity (SBML, CellML, most DSLs), openness (MATLAB), or graphical model representation (ModelingToolkit.jl, Antimony), but the COMBINE languages SBML and CellML currently beat Modelica in terms of ontology support as well as interoperability and acceptance within the systems biology community (RQ3). Additional to the language choice, the application of software engineering techniques to mathematical modeling seems a natural fit that is warranted for larger projects and benefits go as far as guaranteeing methods reproducibility through continuous integration (RQ4). Tool support should concentrate on embedding model design into a typical software engineering workflow with structured editors (Mo|E) and automated testing and documentation (MoST.jl), as well as enabling other domain specific features such as vector graphics editing (MoVE/MoNK) or ontology support (RQ5). Finally, the perfect language for future systems biology models might not exist yet, but could be developed as a combination of the features of Modelica and existing standards, such as an extension of SBML and CellML or a Modelica-like language that uses SBML or CellML as exchange format.

In consequence, these findings reveal the need for changes on multiple levels: Individual modelers have to choose suitable approaches, institutions have to reconsider their infrastructure, and the systems biology community has to provide structures and incentives. In the following I will therefore go into detail what can be done at each of those levels, closing with a list of open questions for future research.

4.1. Implications at the personal level

There are two main contributions that modelers themselves can make to facilitate the creation of large multi-scale models. First, they can focus on model reusability as integral part of their model design. By providing changelogs, reference simulation outputs, structured

documentation, CI, and CD, the barrier for someone who has never seen the model before to start developing their own extensions and derivations should become as low as possible. In particular, it should take as few steps as possible to get from the article text to running a simulation of the described model. Researchers should also be proud of their own reusability efforts and should advertise them in model descriptions and scientific articles in order to proliferate the reusability techniques that they used.

The second major contribution on the individual level is making conscious, informed choices at every step. This means choosing Modelica or ModelingToolkit.jl over SBML and CellML for large projects where it may provide benefits, but it could also mean to use Antimony instead, if compatibility to existing systems biology modeling tools is valued higher than the advanced modularity features of Modelica, or even using SBML directly if there is an additional argument to use SBML features not supported by Antimony. The important part is just that there is a reason for choosing a language or tool beyond the fact that using it is the path of least resistance. This reason should then also guide the model design, ensuring that the required features are actually used to their fullest potential. In particular, if Modelica is chosen as a programming language, models should also be written in a *modelica-esque* style. Finally, the structure of the model code and equations themselves should also be consciously chosen to reflect the structure of the biological system that is being modeled as far as possible. Following these guidelines, it is likely that both the implementation process will become more pleasurable and the resulting model will be of more value to the scientific community.

4.2. Implications at the institutional level

Institutions can work to acquire software engineering expertise in their research groups and can provide infrastructure to facilitate the applications of the techniques presented in this dissertation. For example, they can provide custom GitHub actions like `setup-openmodelica`, a Jenkins server (Jenkins Governance Board 2022) with specialized Docker images (Boettiger 2015) or NF-CORE (Ewels et al. 2020) pipelines that contain all the software that is required to simulate models and create plots. Such a service can be used to establish CI/CD pipelines, which only have to be set up once and can then be reused by all modelers in the institution or working group with minor modifications.

Once the infrastructure is available, it may also be worthwhile to establish policies that incentivize and promote the use of such structures. Institutions can write guidelines for good practice regarding reusability, reproducibility and archiving, and create lab workflows that both make it frictionless and intuitive to follow them, and check that they are actually being followed. To attain the expertise required for this, working groups at universities can

directly cooperate with their software engineering departments on publications, and they can also encourage their systems biology students to take optional courses in object-oriented programming, software testing, and software design in general.

Last but not least, reproducibility can also be facilitated if more reproducibility studies are undertaken and published. Reproducing the methods of an *in silico* study can, for example, be a good introduction for graduate or PhD students into the academic publishing workflow.

4.3. Implications at the community level

The largest Multi-scale models have already reached an extent that goes beyond what a single working group or institution can manage. Consequently, it is clear that interdisciplinary structures at the level of the whole systems biology community are required to facilitate their creation. The most important requirement might be the recognition and training of what I would call a "component library engineer". Since all successful biological Modelica models that I have come across are based on a well-designed library of base components, it is likely that future multi-scale models should also be based on similar structures. The problem here is that the creation of such a library both requires a strong software engineering background, in order to make the library scale well from small test examples to large actual models, and an equally strong understanding of mathematical modeling and the biology of the modeled system. Currently, this combination of expertise is rare, and it is highly unlikely that all modelers who are interested in multi-scale modeling will be able to acquire such detailed knowledge and experience in software engineering. However, the uplifting perspective on this situation is that a well-designed component library can support a multitude of models. Take, for example, the NEST simulator (Jordan et al. 2019) for spiking neural network models, which powers hundreds of models in computational neuroscience.

This example shows that only a few "component library engineers" are required, which could collaborate from different working groups of adjacent disciplines. I strongly believe that Modelica would currently be the right choice for most of these component library projects, because unlike other alternatives like SBML, CellML, and ModelingToolkit.jl, Modelica is specifically designed for this kind of workflow, focusing both on powerful design features for library creators and ease of use for library users.

The second major requirement on the community level are incentives for reusability and good model design. First and foremost, *model engineering* should be recognized not only as a service discipline but as primary source for scientific improvement in systems biology. This warrants its inclusion as subsection or article type within existing journals or perhaps also the creation of a dedicated journal. Additionally, systems biology journals can use check-lists

or “seals of approval” for the adherence to reusability and reproducibility requirements. This would put more emphasis on good model design and would provide incentives for researchers to refactor their code for usability. Maybe this could also lead to a “component library award” much like the library award issued at the Modelica conference to promote the creation of high-quality Modelica libraries (Modelica Association 2022c).

As a last consideration, infrastructure at the community level should acknowledge that while standardization and interoperability are highly desirable, there will always be good reasons to use non-standard tools. As a concrete example, the BioModels database was originally designed to just host SBML models, but has been opened to other models formats. This openness towards experimentation with new solutions is crucial to overcome the limitations of existing standards. To save resources, BioModels currently does not provide the same indexing and curation capabilities for non-SBML formats, but if a format is more widely used over time, the support could be extended. Better yet, if the whole platform code is published under an open-source license[18], advocates for a specific model format can implement their own converters and provide a way to extract metadata from models that improves interoperability with the format of their choice and the formats promoted by the platform.

4.4. Open questions

The following open questions represent limitations of dissertation with regard to the width of the investigation. Several topics could not be explored here but would be interesting for the broader question of how modeling languages can support large and complex multi-scale models in general.

4.4.1. Partial differential equations

In this dissertation, I have focused on ODE/DAE models with discrete subsystems, but there are other formalisms which are important for multi-scale modeling. In particular, partial differential equations (PDE) allow a natural representation of variable changes across spatial dimensions. This can be, for example, used for two- or three-dimensional representations on the tissue or organ level, which are especially common in cardiovascular modeling. Modelica currently does not allow PDE, but in recent years, research for PDE support has gained traction with PDEModelica1 (Šilar, Ježek, and Kofránek 2018). Of all the alternative languages

[18] Currently, the main tools are available on Sourceforge (Chelliah et al. 2022), including a Java-library that interfaces a web application programming interface (API), but the server-side setup does not seem to be available.

mentioned in 3.3, only Julia (with DifferentialEquations.jl or ModelingToolkit.jl) and MATLAB support PDEs, with the latter offering no support to integrate them into Simulink/Simscape models.

Within the systems biology community there are two additional projects that are worth to consider: FieldML is a cousin to CellML developed by the IUPS Physiome project, which is intended to be interoperable with CellML models in future versions (Britten et al. 2013). JSim, the modeling language developed and used for the NPR Physiome project, supports both ODE and PDE, but unfortunately mostly lacks support for modularity (Butterworth et al. 2013). In this area, a more thorough investigation akin to the one performed in this dissertation would be worthwhile to identify whether PDE models have similar or different requirements for their design and which languages would be most suited to couple ODE/DAE and PDE models together for multi-scale modeling.

4.4.2. Stochastic differential equations

Similar to PDE, SDE were excluded from this dissertation. In principle, many modeling languages including SBML, Modelica, the MATLAB environment SimBiology, and ModelingToolkit.jl support SDE. However, it is to be expected that there are also differences between the languages which make them more or less suited for complex stochastic models. An in-depth investigation of this feature could reveal another language characteristic that is important in this area, or it could strengthen the evidence for the benefit of the existing MoDROGH characteristics.

4.4.3. Parameter estimation

The MoDROGH criteria only cover model creation and communication. However, another integral part of *in silico* experiments is the fitting of model parameters to reproduce data obtained from a *wet lab* experiment. Mathematical modeling can only produce biological insight, if a close relationship between the mathematical formulations and the real biological world is maintained. All the languages investigated in this dissertation provide some form of parameter estimation either directly embedded in the language or as separate tools in the language ecosystem. Modelica in particular has a language extension called Optimica (Åkesson 2008), which can be used with OpenModelica (Open Source Modelica Consortium 2022) in order to optimize models to fit a broad range of conditions. Like for PDE and SDE, it might be interesting to investigate whether there are fundamental differences between languages in this regard and whether these differences only come in the form of tool support or can be tied to language features.

4.4.4. Alternatives to classical object-oriented programming

While I have shown that object orientation is well-suited to handle the kind of complexity that comes with large multi-scale models, this should not be seen as a reason to discard other approaches from the software engineering domain completely. Most modern high-level programming languages are either *multi-paradigm* languages, which freely mix different paradigms such as functional or object-oriented programming, or they focus on one paradigm but borrow some concepts from other paradigms. An example is the inclusion of lambda abstractions to create anonymous functions in the most widely used object-oriented languages Java and C++ (Mazinanian et al. 2017; Järvi and J. Freeman 2010). As mentioned in section 3.4.1, Modelica also has some features that could be interpreted as functional programming. However, using these features requires some effort, so it might be interesting to investigate if a fully-featured functional programming style can be of use in mathematical modeling. In particular, ModelingToolkit.jl might be a good staring point for this investigation as Julia allows a functional programming style in principle and can be easily extended with more complex functional features. The multiple dispatch mechanism of Julia, which is an unusual but very flexible implementation of the object-oriented paradigm, might also in itself be interesting to investigate in a modeling context.

4.4.5. Support for multi-scale techniques in languages

As mentioned in section 1.3, *micro-level* models, which simulate all parts of the model at the smallest scale, are limited by their computational complexity. If more than two organizational levels should be spanned by a model, special techniques are required to combine mathematical descriptions at different scales, only simulating with the finest granularity when and where it is required. Dada and P. Mendes (2011) list a total of nine different approaches to do this, but during my research for this dissertation I have not found any language or tool that aims to incorporate any of them into solvers or language constructs. In order to grow multi-level models beyond the current state of the art, an investigation would be required how such techniques can be applied systematically and with minimal modeling effort overhead.

4.4.6. Language Server Protocol implementation for Modelica

In the introduction of Mo|E I mentioned that the language server approach was inspired by the ENSIME project. This project is now retired in favor of metals (Metals 2022), a similar language server for Scala, which uses the Language Server Protocol (LSP) (Microsoft 2022a). The benefit of LSP is that it is a programming language-independent standard for

communication between a compiler and a text editor. As such, clients for a LSP server can be built upon generic solutions, which requires even less effort than implementing the HTTP calls required for Mo|E while at the same time providing much more advanced features like code refactoring. A re-implementation of Mo|E with the LSP, which was already started at our lab as proof of concept, would allow tool vendors to support Modelica with minimal effort, increasing the interoperability of Modelica models. A similar project could also be undertaken for other languages like SBML and CellML.

5. Data and code availability

The models and software developed in this dissertation can be found on GitHub, Zenodo, and BioModels as seen in tables 5.1 and 5.2. All other data that might be relevant to reproduce or continue this work can be found in my thesis archive at `https://github.com/CSchoel/thesis-archive`.

Model	GitHub	Zenodo	BioModels
SHM	CSchoel/shm	10.5281/zenodo.5027354	MODEL2101280001
SHM-conduction	CSchoel/shm-conduction	10.5281/zenodo.4585654	MODEL2103050002
HH-Modelica	CSchoel/hh-modelica	10.5281/zenodo.5018521	MODEL2103050003
InaMo	CSchoel/inamo	10.5281/zenodo.4775302	MODEL2102090002

Table 5.1.: **Database identifiers for models developed in this dissertation.** Lists user and repository name for GitHub, DOI for Zotero, and model ID for BioModels.

Software	GitHub	Zenodo
Mo|E	THM-MoTE/mope-server THM-MoTE/mope-atom-plugin	
MoVE	THM-MoTE/MoVE	
MoDE	THM-MoTE/MoDE	
MoNK	THM-MoTE/MoNK	10.5281/zenodo.4134955
MoST.jl	THM-MoTE/ModelicaScriptingTools.jl	10.5281/zenodo.4792305
setup-openmodelica	THM-MoTE/setup-openmodelica	

Table 5.2.: **Database identifiers for software developed in this dissertation.** Lists user and repository name for GitHub and DOI for Zotero. Mo|E, MoVE, and MoDE were not developed by me but by students under my supervision.

6. Acknowledgements

When I talk about my dissertation, I often brag that I have not only one but four doctoral supervisors: Prof. Dr. Andreas Dominik was a perfect main advisor. When I had a question, he was always there for genuine and often lengthy discussions and his heartfelt encouragement helped me to overcome all obstacles along the way. Prof. Dr. Thomas Letschert was also deeply involved in the dissertation from the beginning, and I am immensely grateful for each of his pointed questions, which more than once led me to rethink and refine my whole approach. I also thank Prof. Dr. Alexander Goesmann for championing my dissertation at the JLU and for helping me to put my work into the wider perspective of systems biology as a whole and Prof. Dr. Christoph Müller for his encouragement to write a cumulative dissertation and for several lengthy, productive meetings and emails.

I am grateful for all my colleagues and all the guests in my office from student to professor, who took the time to listen to my rants and questions and provided me with fresh perspectives and answers. Of these I have to especially highlight Valeria Blesius. She was like a co-pilot for this dissertation, always there to consult for a second opinion, meticulous in her feedback, and an immensely valuable moral support without which I would have lost my sanity more than once.

Along with Valeria, I also thank my other proofreaders Annina Hofferberth, Björn Pfarr, Prof. Dr. Andreas Dominik, Prof. Dr. Alexander Goesmann, and Dr. Gernot Ernst for their valuable feedback.

Equally important as proofreading is having time to write the dissertation in the first place. In this regard, I thank Prof. Dr. Andreas Dominic, Prof. Dr. Andreas Gogol-Döring, and Prof. Dr. Alexander Dworschak for freeing up time in my teaching schedule and therefore allowing me to finish my dissertation. I am also very grateful for funding from the strategic research fund of the THM, which allowed me to only focus on writing articles for half a year, and for the central publication fund of the THM for covering my article processing charges.

I do not only want to thank my superiors and colleagues, but also all my bright students for making my work as a teacher so enjoyable and especially for performing preliminary and ancillary studies which greatly facilitated my own research. In particular, this includes the work of Michael Menzel, Nicola Justus, Marcel Hoppe, Peter Koch, Denis Ilgen, Cornelius Kohl, Rodney Tabernero, and Daniel Otto.

Any research project is in part a community effort and this dissertation is no exception. Without the following researchers from other groups I would have been completely stuck at some point. I therefore thank Prof. Dr. Denis Noble for actually answering my email; Prof. Dr. Peter Hunter, Dr. David Nickerson, Prof. Dr. James Bassingthwaighte, Prof. Dr. Herbert Sauro, and Dr. Maxwell Neal for enlightening email discussions about the vision behind CellML and JSim; my anonymous reviewers—especially for the MoDROGH paper—for spurring me on to investigate aspects I would have missed otherwise; Dr. Henrik Seidel for sending me the source code of the SHM and pointing me to his PhD thesis; Dr. Michelle Lee for finding the source code of the Inada model; and Dr. David Nickerson for clarifying questions about the CellML implementation of the Inada model.

As the strain of a dissertation encroaches into personal life, it is even more important to have friends and family as a counterweight. In place of all the people who brightened my days in this time, I thank Björn, Fabian, and Franziska for strolls, chats, games nights, and nerd talk and for being awesome friends in general. I am also very grateful for the support of my parents, who set me an example of curiosity about the laws of nature and technology alike and who taught me to value knowledge and creativity above all else. Furthermore, I actually had two families ever since I met my wife Annina as my parents-in-law adopted me into theirs very early on and continued to encourage me at every visit with interested questions and a very open display of their pride in me.

To save the best for last, not only this dissertation but also I myself would not be the same without my wife Annina. She always supported me and stood up for me, even against my own self-sabotage. She was a shining example, and a source of strength and warmth that guided me through my darkest hours. Annina, I do this for myself, but I am only able to do it because of you. I therefore wholeheartedly dedicate this dissertation to you.

7. Bibliography

Abadie, B. and S. Ledru (2020). *Engineering Code Quality in the Firefox Browser: A Look at Our Tools and Challenges*. Mozilla Hacks - the Web developer blog. URL: https://hacks.mozilla.org/2020/04/code-quality-tools-at-mozilla/ (visited on Feb. 2, 2022).

Adobe (2022). *Industry-Leading Vector Graphics Software | Adobe Illustrator*. URL: https://www.adobe.com/products/illustrator.html (visited on Mar. 7, 2022).

Afgan, E. et al. (2018). "The Galaxy Platform for Accessible, Reproducible and Collaborative Biomedical Analyses: 2018 Update." In: *Nucleic Acids Research* 46.W1, W537–W544. DOI: 10.1093/nar/gky379.

Ahmad, B. et al. (2021). "Exploring the Binding Mechanism of PF-07321332 SARS-CoV-2 Protease Inhibitor through Molecular Dynamics and Binding Free Energy Simulations." In: *International Journal of Molecular Sciences* 22.17, art. no. 9124. DOI: 10.3390/ijms22179124.

Åkesson, J. (2008). "Optimica—an Extension of Modelica Supporting Dynamic Optimization." In: *Proceedings of the 6th International Modelica Conference*. Bielefeld, Germany, pp. 57–66.

Åkesson, J., M. Gäfvert, and H. Tummescheit (2009). "JModelica—An Open Source Platform for Optimization of Modelica Models." In: *Proceedings of the 6th Vienna International Conference on Mathematical Modelling*. Vienna, Austria, pp. 1325–1332.

Alexandru, C. V. et al. (2018). "On the Usage of Pythonic Idioms." In: *Onward! 2018: Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Boston, Massachusetts, pp. 1–11. DOI: 10.1145/3276954.3276960.

Ascher, U. M. and L. R. Petzold (1998). *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Philadelphia, Pennsylvania: SIAM.

Association for Computing Machinery (2022a). *Artifact Review and Badging – Current*. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current (visited on Mar. 7, 2022).

Association for Computing Machinery (2022b). *Artifact Review and Badging – Version 1.0 (Not Current)*. URL: https://www.acm.org/publications/policies/artifact-review-badging (visited on Mar. 7, 2022).

Bader, F. et al. (2021). "Heart Failure and COVID-19." In: *Heart Failure Reviews* 26.1, pp. 1–10. DOI: 10.1007/s10741-020-10008-2.

Baker, M. (2016). "1,500 Scientists Lift the Lid on Reproducibility." In: *Nature* 533.7604, pp. 452–454. DOI: 10.1038/533452a.

Banerjee, S. (2014). *Mathematical Modeling: Models, Analysis and Applications*. Boca Raton, Florida: CRC Press. DOI: 10.1201/b16526.

Bardini, R. et al. (2017). "Multi-Level and Hybrid Modelling Approaches for Systems Biology." In: *Computational and Structural Biotechnology Journal* 15, pp. 396–402. DOI: 10.1016/j.csbj.2017.07.005.

Barr, R. and R. Plonsey (1984). "Propagation of Excitation in Idealized Anisotropic Two-Dimensional Tissue." In: *Biophysical Journal* 45.6, pp. 1191–1202. DOI: 10.1016/S0006-3495(84)84268-X.

Bassingthwaighte, J. B. (2000). "Strategies for the Physiome Project." In: *Annals of Biomedical Engineering* 28.8, pp. 1043–1058. DOI: 10.1114/1.1313771.

Bassingthwaighte, J. and B. Jardine (2022). *Welcome to the NSR Physiome Project!* URL: https://www.physiome.org/ (visited on Mar. 7, 2022).

Bean, B. P. (2007). "The Action Potential in Mammalian Central Neurons." In: *Nature Reviews Neuroscience* 8.6, pp. 451–465. DOI: 10.1038/nrn2148.

Begley, C. G. and L. M. Ellis (2012). "Raise Standards for Preclinical Cancer Research: Drug Development." In: *Nature* 483.7391, pp. 531–533. DOI: 10.1038/483531a.

Bergmann, F., B. E. Shapiro, and M. Hucka (2021). *SBML Software Matrix*. URL: https://web.archive.org/web/20210506185458/http://sbml.org/SBML_Software_Guide/SBML_Software_Matrix (visited on May 12, 2021).

Berthold, M. R. et al. (2008). "KNIME: The Konstanz Information Miner." In: *Data Analysis, Machine Learning and Applications: Proceedings of the 31st Annual Conference of the Gesellschaft Für Klassifikation*. Freiburg, Germany, pp. 319–326. DOI: 10.1007/978-3-540-78246-9_38.

Beutlich, T. et al. (2020). *Modelica Standard Library*. Version 4.0.0.

Bezanson, J. et al. (2017). "Julia: A Fresh Approach to Numerical Computing." In: *SIAM Review* 59.1, pp. 65–98. DOI: 10.1137/141000671.

Bezanson, J. et al. (2022). *Unit Testing · The Julia Language*. URL: https://docs.julialang.org/en/v1/stdlib/Test/ (visited on Feb. 23, 2022).

Bin Im, U. et al. (2008). "Theoretical Analysis of the Magnetocardiographic Pattern for Reentry Wave Propagation in a Three-Dimensional Human Heart Model." In: *Progress in Biophysics and Molecular Biology* 96.1-3, pp. 339–356. DOI: 10.1016/j.pbiomolbio.2007.07.024.

Black, A. J. and A. J. McKane (2012). "Stochastic Formulation of Ecological Models and Their Applications." In: *Trends in Ecology & Evolution* 27.6, pp. 337–345. DOI: 10.1016/j.tree.2012.01.014.

Blesius, V. et al. (2020). "HRT Assessment Reviewed: A Systematic Review of Heart Rate Turbulence Methodology." In: *Physiological Measurement* 41.8, art. no. 08TR01. DOI: 10.1088/1361-6579/ab98b3.

Blesius, V. et al. (2022). "Comparability of Heart Rate Turbulence Methodology: 15 Intervals Suffice to Calculate Turbulence Slope A Methodological Analysis Using PhysioNet Data of 1074 Patients." In: *Frontiers in Cardiovascular Medicine* 9, art. no. 793535. DOI: 10.3389/fcvm.2022.793535.

Blochwitz, T. et al. (2012). "Functional Mockup Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models." In: *Proceedings of the 9th International Modelica Conference*. Munich, Germany, pp. 173–184. DOI: 10.3384/ecp12076173.

Boettiger, C. (2015). "An Introduction to Docker for Reproducible Research." In: *ACM SIGOPS Operating Systems Review* 49.1, pp. 71–79. DOI: 10.1145/2723872.2723882.

Bologna, M., K. J. Chandía, and J. Flores (2016). "A Non-Linear Mathematical Model for a Three Species Ecosystem: Hippos in Lake Edward." In: *Journal of Theoretical Biology* 389, pp. 83–87. DOI: 10.1016/j.jtbi.2015.10.026.

Borzì, A. (2020). *Modelling with Ordinary Differential Equations: A Comprehensive Approach*. Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series. Boca Raton, Florida: CRC Press.

Bouvy, C. et al. (2012). "Holistic Vehicle Simulation Using Modelica — An Application on Thermal Management and Operation Strategy for Electrified Vehicles." In: *Processings of the 9th International Modelica Conference*. Munich, Germany, pp. 264–270. DOI: 10.3384/ecp12076263.

Brand, S. (2018). "Pace Layering: How Complex Systems Learn and Keep Learning." In: *Journal of Design and Science* 3. In collab. with L. LeJeune, N. Philip, and C. Ahearn. DOI: 10.21428/7f2e5f08.

Braun, W. et al. (2020). "Contributions to the Efficient and Parallel Jacobian Evaluation and Its Application in OpenModelica." In: Boulder, Colorado, pp. 159–167. DOI: 10.3384/ecp20169159.

Briese, L. E., A. Klöckner, and M. Reiner (2017). "The DLR Environment Library for Multi-Disciplinary Aerospace Applications." In: *Proceedings of the 12th International Modelica Conference*. Prague, Czech Republic, pp. 929–938. DOI: 10.3384/ecp17132929.

Brinkrolf, C. et al. (2014). "VANESA - A Software Application for the Visualization and Analysis of Networks in Systems Biology Applications." In: *Journal of Integrative Bioinformatics* 11.2, art. no. 239. DOI: 10.1515/jib-2014-239.

Brinkrolf, C. et al. (2018). "Modeling and Simulating the Aerobic Carbon Metabolism of a Green Microalga Using Petri Nets and New Concepts of VANESA." In: *Journal of Integrative Bioinformatics* 15.3, art. no. 20180018. DOI: 10.1515/jib-2018-0018.

Britten, R. D. et al. (2013). "FieldML, a Proposed Open Standard for the Physiome Project for Mathematical Model Representation." In: *Medical & Biological Engineering & Computing* 51.11, pp. 1191–1207. DOI: 10.1007/s11517-013-1097-7.

Butterworth, E. et al. (2013). "JSim, an Open-Source Modeling System for Data Analysis." In: *F1000Research* 2, art. no. 288. DOI: 10.12688/f1000research.2-288.v1.

California Institute of Technology (2022). *SBML Test Suite Database: Test Cases*. URL: http://raterule.caltech.edu/Facilities/Database (visited on Jan. 31, 2022).

Casella, F. and A. Leva (2006). "Modelling of Thermo-Hydraulic Power Generation Processes Using Modelica." In: *Mathematical and Computer Modelling of Dynamical Systems* 12.1, pp. 19–33. DOI: 10.1080/13873950500071082.

Casella, F. et al. (2016). "Object-Oriented Modelling and Simulation of Large-Scale Electrical Power Systems Using Modelica: A First Feasibility Study." In: *Proceedings of the IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*. Florence, Italy, pp. 6298–6304. DOI: 10.1109/IECON.2016.7793558.

Cellier, F. E. and À. Nebot (2005a). "Object-Oriented Modeling in the Service of Medicine." In: *Proceedings of the Sixth Asia Simulation Conference*. Beijing, China, pp. 33–40.

Cellier, F. E. and À. Nebot (2005b). "The Modelica Bond Graph Library." In: *Proceedings of the 4th International Modelica Conference*. Hamburg, Germany, pp. 57–65.

Cern Data Centre (2022). *Zenodo - Research. Shared.* URL: https://zenodo.org/ (visited on Mar. 7, 2022).

Chacon, S., J. Long, and the Git community (2022). *Git.* URL: https://git-scm.com/ (visited on Mar. 7, 2022).

Chelliah, V. et al. (2022). *BioModels Database Download | SourceForge.Net.* URL: https://sourceforge.net/projects/biomodels/ (visited on Feb. 3, 2022).

Chiara, M. et al. (2021). "Next Generation Sequencing of SARS-CoV-2 Genomes: Challenges, Applications and Opportunities." In: *Briefings in Bioinformatics* 22.2, pp. 616–630. DOI: 10.1093/bib/bbaa297.

Chivers, I. and J. Sleightholme (2018). *Introduction to Programming with Fortran*. 4th ed. London, England: Springer Nature. DOI: 10.1007/978-3-319-75502-1.

Choi, K. et al. (2018). "Tellurium: An Extensible Python-Based Modeling Environment for Systems and Synthetic Biology." In: *Biosystems* 171, pp. 74–79. DOI: 10.1016/j.biosystems.2018.07.006.

Chou, I.-C. and E. O. Voit (2009). "Recent Developments in Parameter Estimation and Structure Identification of Biochemical and Genomic Systems." In: *Mathematical Biosciences* 219.2, pp. 57–83. DOI: 10.1016/j.mbs.2009.03.002.

Claerbout, J. F. and M. Karrenbach (1992). "Electronic Documents Give Reproducible Research a New Meaning." In: *SEG Technical Program Expanded Abstracts 1992*. New Orleans, Louisiana, pp. 601–604. DOI: 10.1190/1.1822162.

Clark, K. B. and C. Y. Baldwin (2000). *Design Rules: The Power of Modularity*. Cambridge, Massachusetts: MIT Press.

Clerx, M. et al. (2016). "Myokit: A Simple Interface to Cardiac Cellular Electrophysiology." In: *Progress in Biophysics and Molecular Biology* 120, pp. 100–114. DOI: 10.1016/j.pbiomolbio.2015.12.008.

Clerx, M. et al. (2020). "CellML 2.0." In: *Journal of Integrative Bioinformatics* 17.2-3, art. no. 20200021. DOI: 10.1515/jib-2020-0021.

Clewley, R. (2012). "Hybrid Models and Biological Model Reduction with PyDSTool." In: *PLoS Computational Biology* 8.8, art. no. e1002628. DOI: 10.1371/journal.pcbi.1002628.

Cooling, M. T. et al. (2010). "Standard Virtual Biological Parts: A Repository of Modular Modeling Components for Synthetic Biology." In: *Bioinformatics* 26.7, pp. 925–931. DOI: 10.1093/bioinformatics/btq063.

Cooling, M. T., P. Hunter, and E. J. Crampin (2008). "Modelling Biological Modularity with CellML." In: *IET Systems Biology* 2.2, pp. 73–79. DOI: 10.1049/iet-syb:20070020.

Courtemanche, M., R. J. Ramirez, and S. Nattel (1998). "Ionic Mechanisms Underlying Human Atrial Action Potential Properties: Insights from a Mathematical Model." In: *American Journal of Physiology-Heart and Circulatory Physiology* 275.1, H301–H321. DOI: 10.1152/ajpheart.1998.275.1.H301.

Courtot, M. et al. (2011). "Controlled Vocabularies and Semantics in Systems Biology." In: *Molecular Systems Biology* 7, art. no. 543. DOI: 10.1038/msb.2011.77.

Crick, F. H. (1958). "On Protein Synthesis." In: *Symposia of the Society for Experimental Biology* 12, pp. 138–163.

Cruse, P. et al. (2022). *Dryad Home - Publish and Preserve Your Data*. URL: https://datadryad.org (visited on Mar. 7, 2022).

Dada, J. O. and P. Mendes (2011). "Multi-Scale Modelling and Simulation in Systems Biology." In: *Integrative Biology* 3.2, pp. 86–96. DOI: 10.1039/c0ib00075b.

Darwin, C. R. (1872). *The Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. 6th ed. London, England: John Murray.

Dassault Systèmes (2022). *Dymola*. URL: https://www.3ds.com/products-services/catia/products/dymola/ (visited on Mar. 7, 2022).

Dawkins, R. (1996). *Climbing Mount Improbable*. New York: Norton.

De Meyts, P. (2016). "The Insulin Receptor and Its Signal Transduction Network." In: *Endotext*. Ed. by K. R. Feingold et al. South Dartmouth, Massachusetts: MDText.com.

DeBoer, R. W., J. M. Karemaker, and J. Strackee (1987). "Hemodynamic Fluctuations and Baroreflex Sensitivity in Humans: A Beat-to-Beat Model." In: *The American Journal of Physiology - Heart and Circulatory Physiology* 253.3, H680–H689.

Desmond, H. (2022). *LiSyM - Liver Systems Medicine*. URL: https://www.lisym.org/ (visited on Mar. 7, 2022).

Des Rivières, J. and J. Wiegand (2004). "Eclipse: A Platform for Integrating Development Tools." In: *IBM Systems Journal* 43.2, pp. 371–383. DOI: 10.1147/sj.432.0371.

Di Francesco, D. and D. Noble (1985). "A Model of Cardiac Electrical Activity Incorporating Ionic Pumps and Concentration Changes." In: *Philosophical Transactions of the Royal Society of London. B, Biological Sciences* 307.1133, pp. 353–398. DOI: 10.1098/rstb.1985.0001.

Di Ventura, B. et al. (2006). "From *in Vivo* to *in Silico* Biology and Back." In: *Nature* 443.7111, pp. 527–533. DOI: 10.1038/nature05127.

Díaz-Zuccarini, V. and J. LeFèvre (2007). "An Energetically Coherent Lumped Parameter Model of the Left Ventricle Specially Developed for Educational Purposes." In: *Computers in Biology and Medicine* 37.6, pp. 774–784. DOI: 10.1016/j.compbiomed.2006.07.002.

Doke, S. K. and S. C. Dhawale (2015). "Alternatives to Animal Testing: A Review." In: *Saudi Pharmaceutical Journal* 23.3, pp. 223–229. DOI: 10.1016/j.jsps.2013.11.002.

Dräger, A. et al. (2009). "SBML2LaTeX: Conversion of SBML Files into Human-Readable Reports." In: *Bioinformatics* 25.11, pp. 1455–1456. DOI: 10.1093/bioinformatics/btp170.

Duggento, A., N. Toschi, and M. Guerrisi (2012). "Modeling of Human Baroreflex: Considerations on the Seidel–Herzel Model." In: *Fluctuation and Noise Letters* 11.1, art. no. 1240017. DOI: 10.1142/S0219477512400172.

Durinx, C. et al. (2017). "Identifying ELIXIR Core Data Resources [Version 2; Peer Review: 2 Approved]." In: *F1000Research* 5, art. no. 2422. DOI: 10.12688/f1000research.9656.2.

Eaton, J. W. et al. (2021). *GNU Octave Version 6.4.0 Manual: A High-Level Interactive Language for Numerical Computations*. URL: https://octave.org/doc/v6.4.0/ (visited on Mar. 7, 2022).

École polytechnique fédérale de Lausanne (2022a). *Blue Brain Project*. URL: http://bluebrain.epfl.ch/ (visited on Mar. 7, 2022).

École polytechnique fédérale de Lausanne (2022b). *FAQ - Blue Brain Project - EPFL*. Blue Brain Project. URL: https://www.epfl.ch/research/domains/bluebrain/frequently_asked_questions/ (visited on Mar. 7, 2022).

Elmqvist, H. (1978). "A Structured Model Language for Large Continuous Systems." PhD thesis. Lund, Sweden: Lund University.

Elmqvist, H., T. Henningsson, and M. Otter (2016). "Systems Modeling and Programming in a Unified Environment Based on Julia." In: *ISoLA 2016: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*. Corfu, Greece, pp. 198–217. DOI: 10.1007/978-3-319-47169-3_15.

Elsheikh, A. et al. (2022). *Xogeny/XogenyTest: An Example of a Testing Framework Written in 100% Modelica*. URL: https://github.com/xogeny/XogenyTest (visited on Mar. 7, 2022).

ENSIME contributors (2022). *ENSIME*. URL: https://github.com/ensime (visited on Mar. 7, 2022).

Ernst, G. (2014). *Heart Rate Variability*. London, England: Springer.

Ewels, P. A. et al. (2020). "The Nf-Core Framework for Community-Curated Bioinformatics Pipelines." In: *Nature Biotechnology* 38.3, pp. 276–278. DOI: 10.1038/s41587-020-0439-x.

Fairchild, K. D. et al. (2009). "Endotoxin Depresses Heart Rate Variability in Mice: Cytokine and Steroid Effects." In: *American Journal of Physiology: Regulatory, Integrative and Comparative Physiology* 297.4, R1019–R1027. DOI: 10.1152/ajpregu.00132.2009.

Ferrell, J. E. (2013). "Feedback Loops and Reciprocal Regulation: Recurring Motifs in the Systems Biology of the Cell Cycle." In: *Current Opinion in Cell Biology* 25.6, pp. 676–686. DOI: 10.1016/j.ceb.2013.07.007.

Festing, S. and R. Wilkinson (2007). "The Ethics of Animal Research: Talking Point on the Use of Animals in Scientific Research." In: *EMBO reports* 8.6, pp. 526–530. DOI: 10.1038/sj.embor.7400993.

Fishman, G. S. (2013). *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer Series in Operations Research. New York: Springer.

FitzHugh, R. (1961). "Impulses and Physiological States in Theoretical Models of Nerve Membrane." In: *Biophysical Journal* 1.6, pp. 445–466. DOI: 10.1016/S0006-3495(61)86902-6.

Franke, R. and H. Wiesmann (2014). "Flexible Modeling of Electrical Power Systems — the Modelica PowerSystems Library." In: *Proceedings of the 10th International Modelica Conference*. Lund, Sweden, pp. 515–522. DOI: 10.3384/ecp14096515.

Franklin, R. E. and R. G. Gosling (1953). "Molecular Configuration in Sodium Thymonucleate." In: *Nature* 171.4356, pp. 740–741. DOI: 10.1038/171740a0.

Freeman, E. et al. (2004). *Head First Design Patterns*. Sebastopol, California: O'Reilly.

Freeman, S. and N. Pryce (2010). *Growing Object-Oriented Software, Guided by Tests*. The Addison-Wesley Signature Series. Upper Saddle River, New Jersey: Addison Wesley.

Fritzson, P. et al. (2005). "The OpenModelica Modeling, Simulation, and Development Environment." In: *Proceedings of the 46th Conference on Simulation and Modelling of the Scandinavian Simulation Society*. Trondheim, Norway.

Gardiner, B. S. et al. (2011). "A Mathematical Model of Diffusional Shunting of Oxygen from Arteries to Veins in the Kidney." In: *American Journal of Physiology-Renal Physiology* 300.6, F1339–F1352. DOI: 10.1152/ajprenal.00544.2010.

Garny, A. and P. J. Hunter (2015). "OpenCOR: A Modular and Interoperable Approach to Computational Biology." In: *Frontiers in Physiology* 6, art. no. 26. DOI: 10.3389/fphys.2015.00026.

Gatherer, D. (2010). "So What Do We Really Mean When We Say That Systems Biology Is Holistic?" In: *BMC Systems Biology* 4.1, art. no. 22. DOI: 10.1186/1752-0509-4-22.

Gershenfeld, N. A. (2011). *The Nature of Mathematical Modeling*. Cambridge, England: Cambridge University Press.

Gerstner, W. et al. (2014). *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge, England: Cambridge University Press.

Gilad, E. et al. (2004). "Ecosystem Engineers: From Pattern Formation to Habitat Creation." In: *Physical Review Letters* 93.9, art. no. 098105. DOI: 10.1103/PhysRevLett.93.098105.

Gillespie, D. T. (1977). "Exact Stochastic Simulation of Coupled Chemical Reactions." In: *The Journal of Physical Chemistry* 81, pp. 2340–2361.

Gilpin, W. (2021). "Desynchronization of Jammed Oscillators by Avalanches." In: *Physical Review Research* 3.2, art. no. 023206. DOI: 10.1103/PhysRevResearch.3.023206.

GitHub (2022a). *Atom*. URL: https://atom.io/ (visited on Mar. 7, 2022).

GitHub (2022b). *GitHub*. URL: https://github.com/ (visited on Mar. 7, 2022).

Goldberg, A. D., C. D. Allis, and E. Bernstein (2007). "Epigenetics: A Landscape Takes Shape." In: *Cell* 128.4, pp. 635–638. DOI: 10.1016/j.cell.2007.02.006.

Goodman, S. N., D. Fanelli, and J. P. A. Ioannidis (2016). "What Does Research Reproducibility Mean?" In: *Science Translational Medicine* 8.341, art. no. 341ps12. DOI: 10.1126/scitranslmed.aaf5027.

Green, S. (2015). "Revisiting Generality in Biology: Systems Biology and the Quest for Design Principles." In: *Biology & Philosophy* 30.5, pp. 629–652. DOI: 10.1007/s10539-015-9496-9.

Greenstein, J. L. et al. (2000). "Role of the Calcium-Independent Transient Outward Current I_{to1} in Shaping Action Potential Morphology and Duration." In: *Circulation Research* 87.11, pp. 1026–1033. DOI: 10.1161/01.RES.87.11.1026.

Grimmett, G. R. and D. R. Stirzaker (2020). *Probability and Random Processes*. 4th ed. Oxford, England: Oxford University Press.

Gündel, M. et al. (2013). "HuPSON: The Human Physiology Simulation Ontology." In: *Journal of Biomedical Semantics* 4.1, art. no. 35. DOI: 10.1186/2041-1480-4-35.

Guyton, A. C., T. G. Coleman, and H. J. Granger (1972). "Circulation: Overall Regulation." In: *Annual Review of Physiology* 34.1, pp. 13–44. DOI: 10.1146/annurev.ph.34.030172.000305.

Hammer, G. L. et al. (2004). "On Systems Thinking, Systems Biology, and the in Silico Plant." In: *Plant Physiology* 134.3, pp. 909–911. DOI: 10.1104/pp.103.034827.

Hamzi, B. and H. Owhadi (2021). "Learning Dynamical Systems from Data: A Simple Cross-Validation Perspective, Part I: Parametric Kernel Flows." In: *Physica D: Nonlinear Phenomena* 421, art. no. 132817. DOI: 10.1016/j.physd.2020.132817.

Hellerstein, J. L. et al. (2019). "Recent Advances in Biomedical Simulations: A Manifesto for Model Engineering." In: *F1000Research* 8, art. no. 261. DOI: 10.12688/f1000research.15997.1.

Hester, R. L. et al. (2011). "HumMod: A Modeling Environment for the Simulation of Integrative Human Physiology." In: *Frontiers in Physiology* 2, art. no. 12. DOI: 10.3389/fphys.2011.00012.

Hesthaven, J. S. (2018). *Numerical Methods for Conservation Laws: From Analysis to Algorithms*. Computational Science & Engineering 18. Philadelphia, Pennsylvania: SIAM.

Hill, C. M., R. D. Waightm, and W. G. Bardsley (1977). "Does Any Enzyme Follow the Michaelis—Menten Equation?" In: *Molecular and Cellular Biochemistry* 15.3, pp. 173–178. DOI: 10.1007/BF01734107.

Hodgkin, A. L. and A. F. Huxley (1952). "A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve." In: *The Journal of Physiology* 117.4, pp. 500–544. DOI: 10.1113/jphysiol.1952.sp004764.

Hoops, S. et al. (2006). "COPASI—a COmplex PAthway SImulator." In: *Bioinformatics* 22.24, pp. 3067–3074. DOI: 10.1093/bioinformatics/btl485.

Hoppe, M. (2017). "Weiterentwicklung eines Diagramm-Editors für Modelica." BA thesis. Giessen, Germany: Technische Hochschule Mittelhessen - THM University of Advanced Sciences.

Hoppe, M. (2022). *THM-MoTE/MoDE: Modelica Diagram Editor*. URL: https://github.com/THM-MoTE/MoDE (visited on Mar. 7, 2022).

Hou, Y. J. et al. (2020). "SARS-CoV-2 D614G Variant Exhibits Efficient Replication Ex Vivo and Transmission in Vivo." In: *Science* 370.6523, pp. 1464–1468. DOI: 10.1126/science.abe8499.

Hucka, M. et al. (2003). "The Systems Biology Markup Language (SBML): A Medium for Representation and Exchange of Biochemical Network Models." In: *Bioinformatics* 19.4, pp. 524–531. DOI: 10.1093/bioinformatics/btg015.

Hulsmans, M. et al. (2017). "Macrophages Facilitate Electrical Conduction in the Heart." In: *Cell* 169.3, pp. 510–522. DOI: 10.1016/j.cell.2017.03.050.

Hunter, P., P. Robbins, and D. Noble (2002). "The IUPS Human Physiome Project." In: *Pflügers Archiv - European Journal of Physiology* 445.1, pp. 1–9. DOI: 10.1007/s00424-002-0890-1.

Inada, S. et al. (2009). "One-Dimensional Mathematical Model of the Atrioventricular Node Including Atrio-Nodal, Nodal, and Nodal-His Cells." In: *Biophysical Journal* 97.8, pp. 2117–2127. DOI: 10.1016/j.bpj.2009.06.056.

Inkscape developers (2022). *Draw Freely | Inkscape*. URL: https://inkscape.org/ (visited on Mar. 7, 2022).

International Union of Physiological Sciences (2022). *Physiome Project: Home*. URL: http://physiomeproject.org/ (visited on Mar. 7, 2022).

Ip, J. E. and B. B. Lerman (2018). "Idiopathic Malignant Premature Ventricular Contractions." In: *Trends in Cardiovascular Medicine* 28.4, pp. 295–302. DOI: 10.1016/j.tcm.2017.11.004.

ISO/IEC (2016). *Information Technology — Mathematical Markup Language (MathML) Version 3.0*. International Standard 40314. Geneva, Switzerland: ISO/IEC JTC 1.

Iyer, V., R. Mazhari, and R. L. Winslow (2004). "A Computational Model of the Human Left-Ventricular Epicardial Myocyte." In: *Biophysical Journal* 87.3, pp. 1507–1525. DOI: 10.1529/biophysj.104.043299.

Jansen, P. L. M. et al. (2019). "Editorial: Systems Biology and Bioinformatics in Gastroenterology and Hepatology." In: *Frontiers in Physiology* 10, art. no. 1438. DOI: 10.3389/fphys.2019.01438.

Järvi, J. and J. Freeman (2010). "C++ Lambda Expressions and Closures." In: *Science of Computer Programming* 75.9, pp. 762–772. DOI: 10.1016/j.scico.2009.04.003.

Jenkins Governance Board (2022). *Jenkins*. URL: https://www.jenkins.io/ (visited on Mar. 7, 2022).

Jordan, J. et al. (2019). *NEST*. Version 2.18.0. Zenodo. DOI: 10.5281/zenodo.2605422.

Jorissen, F. et al. (2018). "Implementation and Verification of the IDEAS Building Energy Simulation Library." In: *Journal of Building Performance Simulation* 11.6, pp. 669–688. DOI: 10.1080/19401493.2018.1428361.

Julia Computing (2022). *JuliaSim - Julia Computing*. URL: https://juliacomputing.com/products/juliasim/ (visited on Jan. 31, 2022).

Justus, N. (2016a). "Design and implementation of a client/server application for editing Modelica inside various text editors." BA thesis. Giessen, Germany: Technische Hochschule Mittelhessen - THM University of Advanced Sciences.

Justus, N. (2016b). *Die Entstehung von MoVE*. Student's Project. Giessen, Germany: Technische Hochschule Mittelhessen - THM University of Advanced Sciences.

Justus, N. (2019). "Webmodelica: A web-based editing and simulation environment for Modelica." MA thesis. Giessen, Germany: Technische Hochschule Mittelhessen - THM University of Advanced Sciences.

Justus, N. (2022a). *Modelica | Editor*. URL: https://thm-mote.github.io/projects/mope (visited on Mar. 7, 2022).

Justus, N. (2022b). *THM-MoTE/Mope-Server: Server Process for Modelica | Editor*. URL: https://github.com/THM-MoTE/mope-server (visited on Mar. 7, 2022).

Justus, N. (2022c). *THM-MoTE/MoVE: Modelica Vector Graphics Editor*. URL: https://github.com/THM-MoTE/MoVE (visited on Mar. 7, 2022).

Justus, N. and C. Ifland (2022). *THM-MoTE/Webmodelica: A Web-Based Modelica-toolbox*. URL: https://github.com/THM-MoTE/webmodelica (visited on Mar. 7, 2022).

Justus, N., C. Schölzel, and A. Dominik (2017). "MoVE – A Standalone Modelica Vector Graphics Editor." In: *Proceedings of the 12th International Modelica Conference*. Prague, Czech Republic, pp. 809–814. DOI: 10.3384/ecp17132809.

Justus, N. et al. (2017). "Mo|E – A Communication Service between Modelica Compilers and Text Editors." In: *Proceedings of the 12th International Modelica Conference*. Prague, Czech Republic, pp. 815–822. DOI: 10.3384/ecp17132815.

Justus, N. et al. (2022). *Modelica Tool Ensemble*. URL: https://github.com/THM-MoTE (visited on Mar. 7, 2022).

Kamoi, S. et al. (2014). "Continuous Stroke Volume Estimation from Aortic Pressure Using Zero Dimensional Cardiovascular Model: Proof of Concept Study from Porcine Experiments." In: *PLoS ONE* 9.7, art. no. e102476. DOI: 10.1371/journal.pone.0102476.

Karczewski, K. J. and M. P. Snyder (2018). "Integrative Omics for Health and Disease." In: *Nature Reviews Genetics* 19.5, pp. 299–310. DOI: 10.1038/nrg.2018.4.

Karr, J. R. et al. (2012). "A Whole-Cell Computational Model Predicts Phenotype from Genotype." In: *Cell* 150.2, pp. 389–401. DOI: 10.1016/j.cell.2012.05.044.

Kay, A. (2003). *Clarification of "Object-Oriented"*. E-mail. URL: https://www.purl.org/stefan_ram/pub/doc_kay_oop_en (visited on Feb. 1, 2022).

Keating, S. M. et al. (2020). "SBML Level 3: An Extensible Format for the Exchange and Reuse of Biological Models." In: *Molecular Systems Biology* 16.8, art. no. e9110. DOI: 10.15252/msb.20199110.

Kelley, D. H. and N. T. Ouellette (2013). "Emergent Dynamics of Laboratory Insect Swarms." In: *Scientific Reports* 3.1, art. no. 1073. DOI: 10.1038/srep01073.

Kernighan, B. W. and D. M. Ritchie (1988). *The C Programming Language*. 2nd ed. Englewood Cliffs, New Jersey: Prentice Hall.

Kirouac, D. C., B. Cicali, and S. Schmidt (2019). "Reproducibility of Quantitative Systems Pharmacology Models: Current Challenges and Future Opportunities." In: *CPT: Pharmacometrics & Systems Pharmacology* 8.4, pp. 205–210. DOI: 10.1002/psp4.12390.

Kitano, H. (2002). "Systems Biology: A Brief Overview." In: *Science* 295.5560, pp. 1662–1664. DOI: 10.1126/science.1069492.

Klabunde, R. E. (2012). *Cardiovascular Physiology Concepts*. 2nd ed. Philadelphia, Pennsylvania: Lippincott Williams & Wilkins.

Kofránek, J., J. Rusz, and S. Matoušek (2007). "Guyton's Diagram Brought to Life - From Graphic Chart to Simulation Model for Teaching Physiology." In: *Technical Computing Prague 2007: 15th Annual Conference Proceedings*. Prague, Czech Republic, pp. 1–13.

Kofránek, J. et al. (2008). "Causal or Acausal Modeling: Labour for Humans or Labour for Machines." In: *Technical Computing Prague*. Prague, Czech Republic, pp. 1–16.

Kofránek, J. et al. (2017). "Integrative Physiology in Modelica." In: *Proceedings of the 12th International Modelica Conference*. Prague, Czech Republic, pp. 589–603. DOI: 10.3384/ecp17132589.

Kotani, K. et al. (2002). "Model for Cardiorespiratory Synchronization in Humans." In: *Physical Review E* 65.5, art. no. 051923. DOI: 10.1103/PhysRevE.65.051923.

Kotani, K. et al. (2005). "Model for Complex Heart Rate Dynamics in Health and Diseases." In: *Physical Review E* 72.4, art. no. 041904. DOI: 10.1103/PhysRevE.72.041904.

Kramer, D. (1999). "API Documentation from Source Code Comments: A Case Study of Javadoc." In: *Proceedings of the 17th Annual International Conference on Computer Documentation - SIGDOC '99*. New Orleans, Louisiana, pp. 147–153. DOI: 10.1145/318372.318577.

Kumar, A. et al. (2022). *OpenModelica/OMJulia.Jl: Julia Scripting OpenModelica Interface*. URL: https://github.com/OpenModelica/OMJulia.jl (visited on Mar. 7, 2022).

Kurata, Y. et al. (2002). "Dynamical Description of Sinoatrial Node Pacemaking: Improved Mathematical Model for Primary Pacemaker Cell." In: *American Journal of Physiology-Heart and Circulatory Physiology* 283.5, H2074–H2101. DOI: 10.1152/ajpheart.00900.2001.

Lacan, O. and T. Fortune (2017). *Keep a Changelog*. URL: https://keepachangelog.com/en/1.0.0/ (visited on Feb. 23, 2022).

Larsdotter Nilsson, E. and P. Fritzson (2005a). "A Metabolic Specialization of a General Purpose Modelica Library for Biological and Biochemical Systems." In: *Proceedings of the 4th International Modelica Conference*. Hamburg, Germany, pp. 85–93. DOI: 10.1.1.151.6207.

Larsdotter Nilsson, E. and P. Fritzson (2005b). "Biochemical and Metabolic Modeling and Simulation with Modelica." In: *BioMedSim'2005: Proceedings of the Conference on Modeling and Simulation in Biology, Medicine and Biomedical Engineering*. Linköping, Sweden, pp. 115–124.

Lee, E. A. et al. (2015). "Modeling and Simulating Cyber-Physical Systems Using CyPhySim." In: *2015 International Conference on Embedded Software (EMSOFT)*. Amsterdam, Netherlands, pp. 115–124. DOI: 10.1109/EMSOFT.2015.7318266.

Leite, M. C. A. and Y. Wang (2010). "Multistability, Oscillations and Bifurcations in Feedback Loops." In: *Mathematical Biosciences and Engineering* 7.1, pp. 83–97. DOI: 10.3934/mbe.2010.7.83.

Lewis, J. et al. (2016). "Where next for the Reproducibility Agenda in Computational Biology?" In: *BMC Systems Biology* 10.1, art. no. 52. DOI: 10.1186/s12918-016-0288-x.

Liggesmeyer, P. (2009). *Softwarequalität: Testen, Analysieren und Verifizieren von Software*. 2nd ed. Heidelberg, Germany: Spektrum.

Lindblad, D. S. et al. (1996). "A Model of the Action Potential and Underlying Membrane Currents in a Rabbit Atrial Cell." In: *American Journal of Physiology - Heart and Circulatory Physiology* 271.4, H1666–H1696. DOI: 10.1152/ajpheart.1996.271.4.H1666.

Lloyd, C. M. (2009). *Inada 2009 — Physiome Model Repository*. URL: https://models.physiomeproject.org/workspace/inada_2009 (visited on Mar. 7, 2022).

Lloyd, J. W. (1994). "Practical Advantages of Declarative Programming." In: *GULP-PRODE'94: 1994 Joint Conference on Declarative Programming*. Peñíscola, Spain, pp. 3–17.

Lopez, C. F. et al. (2013). "Programming Biological Models in Python Using PySB." In: *Molecular Systems Biology* 9.1, art. no. 646. DOI: 10.1038/msb.2013.1.

Lotka, A. J. (1910). "Contribution to the Theory of Periodic Reactions." In: *The Journal of Physical Chemistry* 14.3, pp. 271–274. DOI: 10.1021/j150111a004.

Ma, Y. et al. (2021). "ModelingToolkit: A Composable Graph Transformation System for Equation-Based Modeling." preprint. arXiv: 2103.05244v3 [cs.MS].

Macal, C. and M. North (2005). "Tutorial on Agent-Based Modeling and Simulation." In: *Proceedings of the 2005 Winter Simulation Conference*. Orlando, Florida, pp. 2–15. DOI: 10.1109/WSC.2005.1574234.

Maggioli, F., T. Mancini, and E. Tronci (2020). "SBML2Modelica: Integrating Biochemical Models within Open-Standard Simulation Ecosystems." In: *Bioinformatics* 36.7, pp. 2165–2172. DOI: 10.1093/bioinformatics/btz860.

Makowski, D. et al. (2021). "NeuroKit2: A Python Toolbox for Neurophysiological Signal Processing." In: *Behavior Research Methods* 53.4, pp. 1689–1696. DOI: 10.3758/s13428-020-01516-y.

Malik-Sheriff, R. S. et al. (2019). "BioModels—15 Years of Sharing Computational Models in Life Science." In: *Nucleic Acids Research* 48.D1, pp. D407–D415. DOI: 10.1093/nar/gkz1055.

Mandal, M. et al. (2020). "A Model Based Study on the Dynamics of COVID-19: Prediction and Control." In: *Chaos, Solitons & Fractals* 136, art. no. 109889. DOI: 10.1016/j.chaos.2020.109889.

Margolis, B. W. L. (2017). "SimuPy: A Python Framework for Modeling and Simulating Dynamical Systems." In: *The Journal of Open Source Software* 2.17, art. no. 396. DOI: 10.21105/joss.00396.

Markram, H. (2006). "The Blue Brain Project." In: *Nature Reviews Neuroscience* 7.2, pp. 153–160. DOI: 10.1038/nrn1848.

Marrouche, N. F. et al. (2018). "Catheter Ablation for Atrial Fibrillation with Heart Failure." In: *New England Journal of Medicine* 378.5, pp. 417–427. DOI: 10.1056/NEJMoa1707855.

Martin, A. R. et al. (2021). *From Neuron to Brain*. 6th ed. Sunderland, Massachusetts: Sinauer.

Martraire, C. (2019). *Living Documentation: Continuous Knowledge Sharing by Design*. Boston, Massachusetts: Pearson Education.

Mateják, M. and J. Kofránek (2015). "Physiomodel — An Integrative Physiology in Modelica." In: *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. Milan, Italy, pp. 1464–1467. DOI: 10.1109/EMBC.2015.7318646.

Mateják, M. et al. (2014). "Physiolibrary - Modelica Library for Physiology." In: *Proceedings of the 10th International Modelica Conference*. Lund, Sweden, pp. 499–505. DOI: 10.3384/ecp14096499.

Mateják, M. et al. (2015). "Free Modelica Library for Chemical and Electrochemical Processes." In: *Proceedings of the 11th International Modelica Conference*. Versailles, France, pp. 359–366. DOI: 10.3384/ecp15118359.

Mattsson, S. E. and H. Elmqvist (1997). "Modelica – An International Effort to Design the next Generation Modeling Language." In: *7th IFAC Symposium on Computer Aided Control Systems Design, CACSD'97*. Gent, Belgium, pp. 151–155. DOI: 10.1016/S1474-6670(17)43628-7.

Mazhari, R. et al. (2001). "Molecular Interactions between Two Long-QT Syndrome Gene Products, *HERG* and *KCNE2*, Rationalized by in Vitro and in Silico Analysis." In: *Circulation Research* 89.1, pp. 33–38. DOI: 10.1161/hh1301.093633.

Mazinanian, D. et al. (2017). "Understanding the Use of Lambda Expressions in Java." In: *Proceedings of the ACM on Programming Languages* 1 (OOPSLA), pp. 1–31. DOI: 10.1145/3133909.

McDougal, R. A. et al. (2017). "Twenty Years of ModelDB and beyond: Building Essential Modeling Tools for the Future of Neuroscience." In: *Journal of Computational Neuroscience* 42.1, pp. 1–10. DOI: 10.1007/s10827-016-0623-7.

McLaughlin, B., G. Pollice, and D. West (2007). *Head First Object-Oriented Analysis and Design*. Head First Series. Sebastopol, California: O'Reilly.

Medley, J. K., A. P. Goldberg, and J. R. Karr (2016). "Guidelines for Reproducibly Building and Simulating Systems Biology Models." In: *IEEE Transactions on Bio-medical Engineering* 63.10, pp. 2015–2020. DOI: 10.1109/TBME.2016.2591960.

Medley, J. K. et al. (2018). "Tellurium Notebooks—An Environment for Reproducible Dynamical Modeling in Systems Biology." In: *PLOS Computational Biology* 14.6, art. no. e1006220. DOI: 10.1371/journal.pcbi.1006220.

Mendoza-Juez, B. et al. (2012). "A Mathematical Model for the Glucose-Lactate Metabolism of in Vitro Cancer Cells." In: *Bulletin of Mathematical Biology* 74.5, pp. 1125–1142. DOI: 10.1007/s11538-011-9711-z.

Menzel, M. et al. (2015). "Silicon Heart: An Easy to Use Interactive Real-Time Baroreflex Simulator." In: *Computing in Cardiology* 42, pp. 973–976. DOI: 10.1109/CIC.2015.7411075.

Metals (2022). *Metals | Metals*. URL: https://scalameta.org/metals/ (visited on Feb. 24, 2022).

Michal, G. and D. Schomburg, eds. (2012). *Biochemical Pathways: An Atlas of Biochemistry and Molecular Biology*. 2nd ed. Hoboken, New Jersey: Wiley. DOI: 10.1002/9781118657072.

Microsoft (2022a). *Official Page for Language Server Protocol*. URL: https://microsoft.github.io/language-server-protocol/ (visited on Feb. 23, 2022).

Microsoft (2022b). *Visual Studio Code - Code Editing. Redefined*. URL: https://code.visualstudio.com/ (visited on Mar. 7, 2022).

Millard, P., K. Smallbone, and P. Mendes (2017). "Metabolic Regulation Is Sufficient for Global and Robust Coordination of Glucose Uptake, Catabolism, Energy Production and Growth in Escherichia Coli." In: *PLOS Computational Biology* 13.2, art. no. e1005396. DOI: 10.1371/journal.pcbi.1005396.

Miller, A. K. et al. (2010). "An Overview of the CellML API and Its Implementation." In: *BMC Bioinformatics* 11.1, art. no. 178. DOI: 10.1186/1471-2105-11-178.

Miller, S. C. et al. (2009). "An Examination of Changes in Oxytocin Levels in Men and Women before and after Interaction with a Bonded Dog." In: *Anthrozoös* 22.1, pp. 31–42. DOI: 10.2752/175303708X390455.

Modelica Association (2022a). *14th International Modelica Conference 2021*. URL: https://2021.international.conference.modelica.org/ (visited on Mar. 7, 2022).

Modelica Association (2022b). *17 State Machines - Modelica Language Specification Version 3.4*. URL: https://specification.modelica.org/v3.4/Ch17.html (visited on Jan. 28, 2022).

Modelica Association (2022c). *Modelica Conference Series — Modelica Association*. URL: https://modelica.org/publications/ModelicaConference (visited on Feb. 21, 2022).

Modelica Association (2022d). *Modelica Language — Modelica Association*. URL: https://modelica.org/modelicalanguage.html (visited on Mar. 8, 2022).

Modelica Association (2022e). *Modelica.Electrical*. URL: https://doc.modelica.org/Modelica%204.0.0/Resources/helpDymola/Modelica_Electrical.html#Modelica.Electrical (visited on Jan. 28, 2022).

Modelica Association (2022f). *The Modelica Association — Modelica Association*. URL: https://modelica.org/association.html (visited on Mar. 7, 2022).

Modelica Association (2022g). *Tools | Functional Mock-up Interface*. URL: https://fmi-standard.org/tools/ (visited on Jan. 28, 2022).

Modelon AB (2022). *JModelica.Org*. JModelica.org. URL: http://jmodelica.org (visited on Mar. 7, 2022).

Moe, G. K., W. C. Rheinboldt, and J. Abildskov (1964). "A Computer Model of Atrial Fibrillation." In: *American Heart Journal* 67.2, pp. 200–220. DOI: 10.1016/0002-8703(64)90371-0.

Morris, P. D. et al. (2016). "Computational Fluid Dynamics Modelling in Cardiovascular Medicine." In: *Heart* 102.1, pp. 18–28. DOI: 10.1136/heartjnl-2015-308044.

Mulugeta, L. et al. (2018). "Credibility, Replicability, and Reproducibility in Simulation for Biomedicine and Clinical Applications in Neuroscience." In: *Frontiers in Neuroinformatics* 12, art. no. 18. DOI: 10.3389/fninf.2018.00018.

Nagumo, J., S. Arimoto, and S. Yoshizawa (1962). "An Active Pulse Transmission Line Simulating Nerve Axon." In: *Proceedings of the IRE* 50.10, pp. 2061–2070. DOI: 10.1109/JRPROC.1962.288235.

Naik, A., D. Rozman, and A. Belič (2014). "SteatoNet: The First Integrated Human Metabolic Model with Multi-Layered Regulation to Investigate Liver-Associated Pathologies." In: *PLoS Computational Biology* 10.12, art. no. e1003993. DOI: 10.1371/journal.pcbi.1003993.

Ndaïrou, F. et al. (2020). "Mathematical Modeling of COVID-19 Transmission Dynamics with a Case Study of Wuhan." In: *Chaos, Solitons & Fractals* 135, art. no. 109846. DOI: 10.1016/j.chaos.2020.109846.

Neal, M. L. et al. (2015). "Semantics-Based Composition of Integrated Cardiomyocyte Models Motivated by Real-World Use Cases." In: *PLOS ONE* 10.12, art. no. e0145621. DOI: 10.1371/journal.pone.0145621.

Neal, M. L. et al. (2014). "A Reappraisal of How to Build Modular, Reusable Models of Biological Systems." In: *PLOS Computational Biology* 10.10, art. no. e1003849. DOI: 10.1371/journal.pcbi.1003849.

Nickerson, D. P. and P. J. Hunter (2017). "Introducing the Physiome Journal: Improving Reproducibility, Reuse, and Discovery of Computational Models." In: *2017 IEEE 13th International Conference on E-Science (e-Science)*. Auckland, New Zealand, pp. 448–449. DOI: 10.1109/eScience.2017.65.

Noble, D. (1962). "A Modification of the Hodgkin-Huxley Equations Applicable to Purkinje Fibre Action and Pacemaker Potentials." In: *The Journal of Physiology* 160.2, pp. 317–352. DOI: 10.1113/jphysiol.1962.sp006849.

Noble, D. (2002). "The Rise of Computational Biology." In: *Nature Reviews Molecular Cell Biology* 3.6, pp. 459–463. DOI: 10.1038/nrm810.

Noble, D., A. Garny, and P. J. Noble (2012). "How the Hodgkin-Huxley Equations Inspired the Cardiac Physiome Project." In: *The Journal of Physiology* 590.11, pp. 2613–2628. DOI: 10.1113/jphysiol.2011.224238.

Novère, N. L. et al. (2005). "Minimum Information Requested in the Annotation of Biochemical Models (MIRIAM)." In: *Nature Biotechnology* 23.12, pp. 1509–1515. DOI: 10.1038/nbt1156.

O'Hara, T. and Y. Rudy (2012). "Quantitative Comparison of Cardiac Ventricular Myocyte Electrophysiology and Response to Drugs in Human and Nonhuman Species." In: *American Journal of Physiology: Heart and Circulatory Physiology* 302.5, H1023–H1030. DOI: 10.1152/ajpheart.00785.2011.

Olivier, B. G., J. M. Rohwer, and J.-H. S. Hofmeyr (2005). "Modelling Cellular Systems with PySCeS." In: *Bioinformatics* 21.4, pp. 560–561. DOI: 10.1093/bioinformatics/bti046.

Open Source Modelica Consortium (2022). *Optimization with OpenModelica — OpenModelica User's Guide v1.19.0-Dev-719-G9f4e15ecb7 Documentation*. URL: https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/optimization.html (visited on Mar. 24, 2022).

Ostrand, T., E. Weyuker, and R. Bell (2005). "Predicting the Location and Number of Faults in Large Software Systems." In: *IEEE Transactions on Software Engineering* 31.4, pp. 340–355. DOI: 10.1109/TSE.2005.49.

Palsson, B. (2000). "The Challenges of in Silico Biology." In: *Nature Biotechnology* 18.11, pp. 1147–1150. DOI: 10.1038/81125.

Pammolli, F., L. Magazzini, and M. Riccaboni (2011). "The Productivity Crisis in Pharmaceutical R&D." In: *Nature Reviews Drug Discovery* 10.6, pp. 428–438. DOI: 10.1038/nrd3405.

Papin, J. A. et al. (2020). "Improving Reproducibility in Computational Biology Research." In: *PLOS Computational Biology* 16.5, art. no. e1007881. DOI: 10.1371/journal.pcbi.1007881.

Pereira, T. et al. (2020). "Photoplethysmography Based Atrial Fibrillation Detection: A Review." In: *npj Digital Medicine* 3.1, art. no. 3. DOI: 10.1038/s41746-019-0207-9.

Perley, J. et al. (2014). "Resolving Early Signaling Events in T-cell Activation Leading to IL-2 and FOXP3 Transcription." In: *Processes* 2.4, pp. 867–900. DOI: 10.3390/pr2040867.

Phillips, D. (2018). *Python 3 Object-Oriented Programming: Build Robust and Maintainable Software with Object-Oriented Design Patterns in Python 3.8*. Birmingham, England: Packt.

Piibeleht, M., M. Hatherly, and F. Ekre (2022). *JuliaDocs/Documenter.Jl: A Documentation Generator for Julia*. URL: https://github.com/JuliaDocs/Documenter.jl (visited on Mar. 7, 2022).

Pitt-Francis, J., A. Garny, and D. Gavaghan (2006). "Enabling Computer Models of the Heart for High-Performance Computers and the Grid." In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 364.1843, pp. 1501–1516. DOI: 10.1098/rsta.2006.1783.

Plesser, H. E. (2018). "Reproducibility vs. Replicability: A Brief History of a Confused Terminology." In: *Frontiers in Neuroinformatics* 11, art. no. 76. DOI: 10.3389/fninf.2017.00076.

Ploch, T. et al. (2019). "Multiscale Dynamic Modeling and Simulation of a Biorefinery." In: *Biotechnology and Bioengineering* 116.10, pp. 2561–2574. DOI: 10.1002/bit.27099.

Pop, A. D. I. et al. (2006). "OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging." In: *Proceedings of the 5th International Modelica Conference*. Vienna, Austria, pp. 459–465.

Porubsky, V. L. et al. (2020). "Best Practices for Making Reproducible Biochemical Models." In: *Cell Systems* 11.2, pp. 109–120. DOI: 10.1016/j.cels.2020.06.012.

Potvin, R. (2015). "The Motivation for a Monolithic Codebase." video talk. Dev Tools @Scale (San Jose, California). URL: https://www.youtube.com/watch?v=W71BTkUbdqE (visited on Mar. 30, 2022).

Proß, S. et al. (2012). "PNlib — A Modelica Library for Simulation of Biological Systems Based on Extended Hybrid Petri Nets." In: *Proceedings of the International Workshop on Biological Processes & Petri Nets, a Sattelite Event of the 33rd International Conference on Application and Theory of Petri Nets and Concurrency*. Hamburg, Germany, pp. 47–61.

Python Software Foundation (2022a). *Unittest — Unit Testing Framework — Python 3.10.2 Documentation*. URL: https://docs.python.org/3/library/unittest.html (visited on Feb. 23, 2022).

Python Software Foundation (2022b). *Welcome to Python.Org*. URL: https://www.python.org/ (visited on Mar. 7, 2022).

Quint, A. (2003). "Scalable Vector Graphics." In: *IEEE Multimedia* 10.3, pp. 99–102. DOI: 10.1109/MMUL.2003.1218261.

Rackauckas, C. et al. (2020). "Accelerated Predictive Healthcare Analytics with Pumas, a High Performance Pharmaceutical Modeling and Simulation Platform." preprint. bioRxiv: 10.1101/2020.11.28.402297v1.

Rackauckas, C. et al. (2021). "Composing Modeling and Simulation with Machine Learning in Julia." preprint. arXiv: 2105.05946 [cs.CE].

Rackauckas, C. and Q. Nie (2017). "DifferentialEquations.Jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia." In: *Journal of Open Research Software* 5, art. no. 15. DOI: 10.5334/jors.151.

Reisig, W. (1985). *Petri Nets: An Introduction*. Berlin, Germany: Springer.

Riaz, M., E. Mendes, and E. Tempero (2009). "A Systematic Review of Software Maintainability Prediction and Metrics." In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. Lake Buena Vista, Florida, pp. 367–377. DOI: 10.1109/ESEM.2009.5314233.

Richardson, M. and S. P. Wallace (2013). *Getting Started with Raspberry Pi*. Sebastopol, California: O'Reilly.

Roth, G. A. et al. (2018). "Global, Regional, and National Age-Sex-Specific Mortality for 282 Causes of Death in 195 Countries and Territories, 1980–2017: A Systematic Analysis for the Global Burden of Disease Study 2017." In: *The Lancet* 392.10159, pp. 1736–1788. DOI: 10.1016/S0140-6736(18)32203-7.

Rouleau, G. (2019). *Simulink and the Functional Mock-up Interface Standard*. Guy on Simulink - MATLAB & Simulink. URL: https://blogs.mathworks.com/simulink/2019/04/15/simulink-and-the-functional-mock-up-interface-standard (visited on Feb. 21, 2022).

Salam, M. A. and Q. M. Rahman (2018). *Fundamentals of Electrical Circuit Analysis*. Singapore: Springer Nature. DOI: 10.1007/978-981-10-8624-3.

Saldamli, L. (2006). "PDEModelica — A High-Level Language for Modeling with Partial Differential Equations." PhD thesis. Linköping, Sweden: Linköping University.

Samlaus, R. et al. (2014). "MoUnit — A Framework for Automatic Modelica Model Testing." In: *Proceedings of the 10th International Modelica Conference*. Lund, Sweden, pp. 549–556. DOI: 10.3384/ecp14096549.

Sampson, K. J. et al. (2010). "A Computational Model of Purkinje Fibre Single Cell Electrophysiology: Implications for the Long QT Syndrome: Purkinje Fibre Cell *in Silico* Model." In: *The Journal of Physiology* 588.14, pp. 2643–2655. DOI: 10.1113/jphysiol.2010.187328.

Sandve, G. K. et al. (2013). "Ten Simple Rules for Reproducible Computational Research." In: *PLoS Computational Biology* 9.10, art. no. e1003285. DOI: 10.1371/journal.pcbi.1003285.

Santana, E. J. et al. (2021). "Detecting and Mitigating Adversarial Examples in Regression Tasks: A Photovoltaic Power Generation Forecasting Case Study." In: *Information* 12.10, art. no. 394. DOI: 10.3390/info12100394.

Sanz, V., F. Bergero, and A. Urquia (2018). "An Approach to Agent-Based Modeling with Modelica." In: *Simulation Modelling Practice and Theory* 83, pp. 65–74. DOI: 10.1016/j.simpat.2017.12.012.

Sarkar, S. et al. (2009). "Modularization of a Large-Scale Business Application: A Case Study." In: *IEEE Software* 26.2, pp. 28–35. DOI: 10.1109/MS.2009.42.

Sarwar, D. M. et al. (2019). "Model Annotation and Discovery with the Physiome Model Repository." In: *BMC Bioinformatics* 20.1, art. no. 457. DOI: 10.1186/s12859-019-2987-y.

Saul, J. P. et al. (1991). "Transfer Function Analysis of the Circulation: Unique Insights into Cardiovascular Regulation." In: *American Journal of Physiology - Heart and Circulatory Physiology* 261.4, H1231–H1245. DOI: 10.1152/ajpheart.1991.261.4.H1231.

Sauro, H. et al. (2022). *Home - Center for Reproducible Biomedical Modeling*. URL: https://reproduciblebiomodels.org/ (visited on Mar. 7, 2022).

SBML.org (2022). *SBML.Org: SBML Level 3 Hierarchical Model Composition*. URL: https://sbml.org/documents/specifications/level-3/version-1/comp/ (visited on Jan. 31, 2022).

Schölzel, C. (2018). "Brutus der Orkschamane erklärt die Brute-Force-Methode: Gamification und E-Learning in der Veranstaltung ‚Algorithmen und Datenstrukturen'." In: *Proceedings der Pre-Conference-Workshops der 16. E-Learning Fachtagung Informatik*. Frankfurt, Germany. URL: http://ceur-ws.org/Vol-2250/WS_Pro_paper1.pdf (visited on Apr. 1, 2022).

Schölzel, C. (2019). *Nonlinear Measures for Dynamical Systems*. Version 0.5.2. Zenodo. DOI: 10.5281/ZENODO.3814723.

Schölzel, C. (2020). *MoNK - A MOdelica iNKscape Extension*. Version v0.2.0. Zenodo. DOI: 10.5281/zenodo.4134955.

Schölzel, C. (2021a). *Modelica Implementation of the Seidel-Herzel Model of the Human Baroreflex (Version 1.7.0)*. Version 1.7.0. Zenodo. DOI: 10.5281/ZENODO.5027354.

Schölzel, C. (2021b). *THM-MoTE/ModelicaScriptingTools.Jl: Release v1.1.0*. Version v1.1.0. Zenodo. DOI: 10.5281/ZENODO.4792305.

Schölzel, C. (2022a). *THM-MoTE/MoNK: Extension for Inkscape to Produce Modelica Icon Annotations*. URL: https://github.com/THM-MoTE/MoNK (visited on Mar. 7, 2022).

Schölzel, C. (2022b). *THM-MoTE/Setup-Openmodelica: GitHub Action for Installing OpenModelica*. URL: https://github.com/THM-MoTE/setup-openmodelica (visited on Feb. 23, 2022).

Schölzel, C. and A. Dominik (2016). "Can Electrocardiogram Classification Be Applied to Phonocardiogram Data? – An Analysis Using Recurrent Neural Networks." In: *Computing in Cardiology* 43, pp. 581–584. DOI: 10.22489/CinC.2016.167-215.

Schölzel, C. et al. (2015). "Modeling Biology in Modelica: The Human Baroreflex." In: *Proceedings of the 11th International Modelica Conference*. Versailles, France, pp. 367–376. DOI: 10.3384/ecp15118367.

Schölzel, C. et al. (2020). "An Understandable, Extensible, and Reusable Implementation of the Hodgkin-Huxley Equations Using Modelica." In: *Frontiers in Physiology* 11, art. no. 583203. DOI: 10.3389/fphys.2020.583203.

Schölzel, C. et al. (2021a). "Characteristics of Mathematical Modeling Languages That Facilitate Model Reuse in Systems Biology: A Software Engineering Perspective." In: *npj Systems Biology and Applications* 7.1, art. no. 27. DOI: 10.1038/s41540-021-00182-w.

Schölzel, C. et al. (2021b). "Countering Reproducibility Issues in Mathematical Models with Software Engineering Techniques: A Case Study Using a One-Dimensional Mathematical Model of the Atrioventricular Node." In: *PLOS ONE* 16.7, art. no. e0254749. DOI: 10.1371/journal.pone.0254749.

Scialo, F. et al. (2020). "ACE2: The Major Cell Entry Receptor for SARS-CoV-2." In: *Lung* 198.6, pp. 867–877. DOI: 10.1007/s00408-020-00408-4.

Seemann, G. et al. (2006). "Heterogeneous Three-Dimensional Anatomical and Electrophysiological Model of Human Atria." In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 364.1843, pp. 1465–1481. DOI: 10.1098/rsta.2006.1781.

Seidel, H. (1997). "Nonlinear Dynamics of Physiological Rhythms." PhD thesis. Berlin, Germany: Technische Universität Berlin.

Seidel, H. and H. Herzel (1998). "Bifurcations in a Nonlinear Model of the Baroreceptor-Cardiac Reflex." In: *Physica D: Nonlinear Phenomena* 115.1-2, pp. 145–160. DOI: 10.1016/S0167-2789(97)00229-7.

Serb, J. M. and D. J. Eernisse (2008). "Charting Evolution's Trajectory: Using Molluscan Eye Diversity to Understand Parallel and Convergent Evolution." In: *Evolution: Education and Outreach* 1.4, pp. 439–447. DOI: 10.1007/s12052-008-0084-1.

Shaffer, F. and J. P. Ginsberg (2017). "An Overview of Heart Rate Variability Metrics and Norms." In: *Frontiers in Public Health* 5, art. no. 258. DOI: 10.3389/fpubh.2017.00258.

Shahin, M., M. Ali Babar, and L. Zhu (2017). "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices." In: *IEEE Access* 5, pp. 3909–3943. DOI: 10.1109/ACCESS.2017.2685629.

Shampine, L. F., M. W. Reichelt, and J. A. Kierzenka (1999). "Solving Index-1 DAEs in MATLAB and Simulink." In: *SIAM Review* 41.3, pp. 538–552. DOI: 10.1137/S003614459933425X.

Shimizu, S. et al. (2018). "Lumped Parameter Model for Hemodynamic Simulation of Congenital Heart Diseases." In: *The Journal of Physiological Sciences* 68.2, pp. 103–111. DOI: 10.1007/s12576-017-0585-1.

Sierra, K. and B. Bates (2005). *Head First Java*. 2nd ed. Sebastopol, California: O'Reilly.

Šilar, J., F. Ježek, and J. Kofránek (2018). "PDEModelica1: A Modelica Language Extension for Partial Differential Equations Implemented in OpenModelica." In: *International Journal of Modelling and Simulation* 38.2, pp. 128–137. DOI: 10.1080/02286203.2017.1404417.

Šilar, J. et al. (2019). "Model Visualization for E-Learning, Kidney Simulator for Medical Students." In: *Proceedings of the 13th International Modelica Conference*. Regensburg, Germany, pp. 393–402. DOI: 10.3384/ecp19157393.

Sjölund, M., A. Pop, and perost (2022). *OpenModelica/OpenModelicaLibraryTesting: Test Script for OMCompiler+OpenModelicaLibraries*. URL: https://github.com/OpenModelica/OpenModelicaLibraryTesting (visited on Mar. 7, 2022).

Smith, L. P. et al. (2009). "Antimony: A Modular Model Definition Language." In: *Bioinformatics* 25.18, pp. 2452–2454. DOI: 10.1093/bioinformatics/btp401.

Steppe, K. et al. (2006). "A Mathematical Model Linking Tree Sap Flow Dynamics to Daily Stem Diameter Fluctuations and Radial Stem Growth." In: *Tree Physiology* 26.3, pp. 257–273. DOI: 10.1093/treephys/26.3.257.

Stodden, V., J. Seiler, and Z. Ma (2018). "An Empirical Analysis of Journal Policy Effectiveness for Computational Reproducibility." In: *Proceedings of the National Academy of Sciences* 115.11, pp. 2584–2589. DOI: 10.1073/pnas.1708290115.

Strocchi, M. et al. (2020). "A Publicly Available Virtual Cohort of Four-Chamber Heart Meshes for Cardiac Electro-Mechanics Simulations." In: *PLOS ONE* 15.6, art. no. e0235145. DOI: 10.1371/journal.pone.0235145.

Stroustrup, B. (2013). *The C++ Programming Language*. 4th ed. Upper Saddle River, New Jersey: Addison-Wesley.

Sublime HQ (2022). *Sublime Text - Text Editing, Done Right*. URL: https://www.sublimetext.com/ (visited on Mar. 7, 2022).

Summers, R., T. Coleman, and J. Meck (2008). "Development of the Digital Astronaut Project for the Analysis of the Mechanisms of Physiologic Adaptation to Microgravity: Validation of the Cardiovascular System Module." In: *Acta Astronautica* 63.7-10, pp. 758–762. DOI: 10.1016/j.actaastro.2007.12.054.

Takebe, T., R. Imai, and S. Ono (2018). "The Current Status of Drug Discovery and Development as Originated in United States Academia: The Influence of Industrial and Academic Collaboration on Drug Discovery and Development." In: *Clinical and Translational Science* 11.6, pp. 597–606. DOI: 10.1111/cts.12577.

Tao, K. et al. (2021). "The Biological and Clinical Significance of Emerging SARS-CoV-2 Variants." In: *Nature Reviews Genetics* 22.12, pp. 757–773. DOI: 10.1038/s41576-021-00408-x.

Task Force of the European Society of Cardiology and The North American Society of Pacing and Elecrophysiology (1996). "Heart Rate Variability: Standards of Measurement, Physiological Interpretation, and Clinical Use." In: *Circulation* 93.5, pp. 1043–1065. DOI: 10.1161/01.CIR.93.5.1043.

The Editors of Encyclopaedia Britannica (2021). *Software*. In: *Encyclopedia Britannica*. URL: https://www.britannica.com/technology/software (visited on Feb. 19, 2021).

The MathWorks (2022a). *Algebraic Loop Concepts - MATLAB & Simulink*. URL: https://www.mathworks.com/help/simulink/ug/algebraic-loops.html (visited on Feb. 18, 2022).

The MathWorks (2022b). *Continuous Integration - MATLAB & Simulink*. URL: https://www.mathworks.com/solutions/continuous-integration.html (visited on Feb. 18, 2022).

The MathWorks (2022c). *MATLAB*. URL: https://www.mathworks.com/products/matlab.html (visited on Mar. 7, 2022).

The MathWorks (2022d). *Simscape*. URL: https://www.mathworks.com/products/simscape.html (visited on Mar. 7, 2022).

The MathWorks (2022e). *Simulink*. URL: https://www.mathworks.com/products/simulink.html (visited on Mar. 7, 2022).

The Regents of the University of California (2022a). *Development — BuildingsPy Documentation*. URL: https://simulationresearch.lbl.gov/modelica/buildingspy/development.html#module-buildingspy.development.regressiontest (visited on Mar. 7, 2022).

The Regents of the University of California (2022b). *UC3 Merritt Home*. URL: https://merritt.cdlib.org/ (visited on Mar. 7, 2022).

Thio, C. H. et al. (2018). "Heart Rate Variability and Its Relation to Chronic Kidney Disease: Results from the PREVEND Study." In: *Psychosomatic Medicine* 80.3, pp. 307–316. DOI: 10.1097/PSY.0000000000000556.

Thoma, J. U. (1975). *Introduction to Bond Graphs and Their Applications*. Braunschweig, Germany: Pergamon.

Tiller, M. (2020). *Modelica by Example*. e-book. self-published. URL: https://mbe.modelica.university/ (visited on Mar. 8, 2022).

Tiwari, K. et al. (2021). "Reproducibility in Systems Biology Modelling." In: *Molecular Systems Biology* 17.2, art. no. e9982. DOI: 10.15252/msb.20209982.

Topalidou, M. et al. (2015). "A Long Journey into Reproducible Computational Neuroscience." In: *Frontiers in Computational Neuroscience* 9.30, pp. 1–2. DOI: 10.3389/fncom.2015.00030.

Tuan, N. H., H. Mohammadi, and S. Rezapour (2020). "A Mathematical Model for COVID-19 Transmission by Using the Caputo Fractional Derivative." In: *Chaos, Solitons & Fractals* 140, art. no. 110107. DOI: 10.1016/j.chaos.2020.110107.

Tyers, M. and M. Mann (2003). "From Genomics to Proteomics." In: *Nature* 422.6928, pp. 193–197. DOI: 10.1038/nature01510.

Van Rossum, G. and F. L. Drake (2009). *Python 3 Reference Manual*. Scotts Valley, California: CreateSpace.

Voit, E. O. (2018). *A First Course in Systems Biology*. 2nd ed. New York: Garland Science.

Volterra, V. (1926a). "Fluctuations in the Abundance of a Species Considered Mathematically." In: *Nature* 118.2972, pp. 558–560. DOI: 10.1038/118558a0.

Volterra, V. (1926b). "Variazioni e Fluttuazioni Del Numero d'individui in Specie Animali Conviventi." In: *Memoria della Reale Accademia dei Lincei* 2, pp. 31–113.

Walpole, J., J. A. Papin, and S. M. Peirce (2013). "Multiscale Computational Models of Complex Biological Systems." In: *Annual Review of Biomedical Engineering* 15.1, pp. 137–154. DOI: 10.1146/annurev-bioeng-071811-150104.

Waltemath, D. and O. Wolkenhauer (2016). "How Modeling Standards, Software, and Initiatives Support Reproducibility in Systems Biology and Systems Medicine." In: *IEEE Transactions on Biomedical Engineering* 63.10, pp. 1999–2006. DOI: 10.1109/TBME.2016.2555481.

Waltemath, D. et al. (2011). "Minimum Information About a Simulation Experiment (MIASE)." In: *PLoS Computational Biology* 7.4, art. no. e1001122. DOI: 10.1371/journal.pcbi.1001122.

Waltemath, D. et al. (2020). "The First 10 Years of the International Coordination Network for Standards in Systems and Synthetic Biology (COMBINE)." In: *Journal of Integrative Bioinformatics* 17.2-3, art. no. 20200005. DOI: 10.1515/jib-2020-0005.

Walters, T. E. et al. (2018). "Left Ventricular Dyssynchrony Predicts the Cardiomyopathy Associated with Premature Ventricular Contractions." In: *Journal of the American College of Cardiology* 72.23, pp. 2870–2882. DOI: 10.1016/j.jacc.2018.09.059.

Walther, M. et al. (2014). "Equation Based Parallelization of Modelica Models." In: *Proceedings of the 10th International Modelica Conference*. Lund, Sweden, pp. 1213–1220. DOI: 10.3384/ecp140961213.

Warner, H. R. (1958). "The Frequency-Dependent Nature of Blood Pressure Regulation by the Carotid Sinus Studied with an Electric Analog." In: *Circulation Research* 6.1, pp. 35–40. DOI: 10.1161/01.RES.6.1.35.

Warshel, A. (2014). "Multiscale Modeling of Biological Functions: From Enzymes to Molecular Machines (Nobel Lecture)." In: *Angewandte Chemie (International Ed. in English)* 53.38, pp. 10020–10031. DOI: 10.1002/anie.201403689.

Watson, J. D. and F. H. C. Crick (1953). "Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid." In: *Nature* 171.4356, pp. 737–738. DOI: 10.1038/171737a0.

Wehrwein, E. A. and M. J. Joyner (2013). "Regulation of Blood Pressure by the Arterial Baroreflex and Autonomic Nervous System." In: *Autonomic Nervous System*. Ed. by R. M. Buijs and D. F. Swaab. Vol. 117. Handbook of Clinical Neurology 3. Amsterdam, Netherlands: Elsevier, pp. 89–102. DOI: 10.1016/B978-0-444-53491-0.00008-0.

Wetter, M. et al. (2014). "Modelica Buildings Library." In: *Journal of Building Performance Simulation* 7.4, pp. 253–270. DOI: 10.1080/19401493.2013.765506.

White, H. C. et al. (2008). "The Dryad Data Repository: A Singapore Framework Metadata Architecture in a DSpace Environment." In: *DCMI '08: Proceedings of the 2008 International Conference on Dublin Core and Metadata Applications*. Berlin, Germany, pp. 157–162.

Wilkinson, M. D. et al. (2016). "The FAIR Guiding Principles for Scientific Data Management and Stewardship." In: *Scientific Data* 3, art. no. 160018. DOI: 10.1038/sdata.2016.18.

Winslow, R. L. et al. (1993). "Simulating Cardiac Sinus and Atrial Network Dynamics on the Connection Machine." In: *Physica D: Nonlinear Phenomena* 64.1-3, pp. 281–298. DOI: 10.1016/0167-2789(93)90260-8.

Wiysonge, C. S. et al. (2017). "Beta-Blockers for Hypertension." In: *Cochrane Database of Systematic Reviews* 1, art. no. CD002003. DOI: 10.1002/14651858.CD002003.pub5.

Wolfram, S. (1983). "Statistical Mechanics of Cellular Automata." In: *Reviews of Modern Physics* 55.3, pp. 601–644. DOI: 10.1103/RevModPhys.55.601.

Wolkenhauer, O. (2001). "Systems Biology: The Reincarnation of Systems Theory Applied in Biology?" In: *Briefings in Bioinformatics* 2.3, pp. 258–270. DOI: 10.1093/bib/2.3.258.

Woller, A. et al. (2016). "A Mathematical Model of the Liver Circadian Clock Linking Feeding and Fasting Cycles to Clock Function." In: *Cell Reports* 17.4, pp. 1087–1097. DOI: 10.1016/j.celrep.2016.09.060.

Wolstencroft, K. et al. (2015). "SEEK: A Systems Biology Data and Model Management Platform." In: *BMC Systems Biology* 9.1, art. no. 33. DOI: 10.1186/s12918-015-0174-y.

Wolstencroft, K. et al. (2017). "FAIRDOMHub: A Repository and Collaboration Environment for Sharing Systems Biology Research." In: *Nucleic Acids Research* 45.D1, pp. D404–D407. DOI: 10.1093/nar/gkw1032.

Wong, C. H., K. W. Siah, and A. W. Lo (2019). "Estimation of Clinical Trial Success Rates and Related Parameters." In: *Biostatistics* 20.2, pp. 273–286. DOI: 10.1093/biostatistics/kxx069.

Xhyheri, B. et al. (2012). "Heart Rate Variability Today." In: *Progress in Cardiovascular Diseases* 55.3, pp. 321–331. DOI: 10.1016/j.pcad.2012.09.001.

Yadav, S. P. (2007). "The Wholeness in Suffix -Omics, -Omes, and the Word Om." In: *Journal of biomolecular techniques: JBT* 18.5, art. no. 277.

Yu, T. et al. (2011). "The Physiome Model Repository 2." In: *Bioinformatics* 27.5, pp. 743–744. DOI: 10.1093/bioinformatics/btq723.

Zhang, H. et al. (2000). "Mathematical Models of Action Potentials in the Periphery and Center of the Rabbit Sinoatrial Node." In: *American Journal of Physiology-Heart and Circulatory Physiology* 279.1, H397–H421. DOI: 10.1152/ajpheart.2000.279.1.H397.

Zhu, X.-G. et al. (2016). "Plants *in Silico*: Why, Why Now and What?-An Integrative Platform for Plant Systems Biology Research." In: *Plant, Cell & Environment* 39.5, pp. 1049–1057. DOI: 10.1111/pce.12673.

8. Publications

8.1. Characteristics of mathematical modeling languages that facilitate model reuse in systems biology: A software engineering perspective

Title	Characteristics of mathematical modeling languages that facilitate model reuse in systems biology: a software engineering perspective
Authors	C. Schölzel, V. Blesius, G. Ernst, and A. Dominik
Publication type	Journal article
Journal	npj Systems Biology and Applications
Publisher	Springer Nature
Year	2021
Volume	7
Issue	1
Article number	27
DOI	10.1038/s41540-021-00182-w

Contributions by dissertation author (using the Contributor Roles Taxonomy):

Conceptualization	Conceived project, formulated research goals
Data curation	Maintained model code on GitHub, Zenodo, and BioModels
Investigation	Performed all experiments, literature research, and other evidence collection
Methodology	Conceived the MoDROGH characteristics
Software	Implemented the SHM-conduction model
Validation	Implemented test script for SHM-conduction model
Visualization	Created all figures and tables
Writing — original draft	Wrote initial draft
Writing — review & editing	Revised manuscript due to feedback of other authors and reviewers

npj | Systems Biology and Applications

www.nature.com/npjsba

ARTICLE OPEN

Check for updates

Characteristics of mathematical modeling languages that facilitate model reuse in systems biology: a software engineering perspective

Christopher Schölzel[1]✉, Valeria Blesius[1], Gernot Ernst[2,3] and Andreas Dominik[1]

Reuse of mathematical models becomes increasingly important in systems biology as research moves toward large, multi-scale models composed of heterogeneous subcomponents. Currently, many models are not easily reusable due to inflexible or confusing code, inappropriate languages, or insufficient documentation. Best practice suggestions rarely cover such low-level design aspects. This gap could be filled by software engineering, which addresses those same issues for software reuse. We show that languages can facilitate reusability by being modular, human-readable, hybrid (i.e., supporting multiple formalisms), open, declarative, and by supporting the graphical representation of models. Modelers should not only use such a language, but be aware of the features that make it desirable and know how to apply them effectively. For this reason, we compare existing suitable languages in detail and demonstrate their benefits for a modular model of the human cardiac conduction system written in Modelica.

npj Systems Biology and Applications (2021)7:27 ; https://doi.org/10.1038/s41540-021-00182-w

INTRODUCTION

As the understanding of biological systems grows, it becomes more and more apparent that their behavior cannot be reliably predicted without the help of mathematical models. In the past, these models were confined to single phenomena, such as the Hodgkin-Huxley model of the generation of neuronal action potentials[1]. They have served their purpose up to a point where now it is necessary to take into account the upward and downward causations that link all levels of organization in a biological system from genes to proteins to cells to tissue to organs to whole organisms, populations, and ecosystems[2]. These causations span effects on multiple scales of space and time, which need to be included in models. This can be achieved by two different approaches. A *micro-level* model combines thousands of individual homogeneous submodels to reach the next higher scale. This approach requires a vast amount of computing power and is therefore usually limited to span a distance of only two scales. More wide-spanning multi-scale models can be achieved by the *multi-level* approach, which combines both macro- and micro-level descriptions of a system by different models[3]. While micro-level parts of such a model may look as described above, the macro-level parts feature heterogeneous descriptions of subsystems and their high-level interactions. For this approach, a wide variety of techniques exist that reduce the computational complexity of resulting models[4]. While both approaches require the reuse of existing models, the multi-level approach additionally involves the combination of independent submodels, which may have been designed for different purposes and in different labs. These submodels may even use different modeling formalisms, thus forming a *multi-class* model[5].

The first step in building a model consisting of several submodels is to regenerate the individual parts from the literature. This can already be a challenge due to several issues with reproducibility in systems biology including incomplete model descriptions, errors in formulas, availability of the code or missing descriptions of experiment setup or design choices[6–8]. A recent study by curators of the BioModels database showed that only 51% of 455 published models were directly reproducible[8]. In an extreme case, Topalidou et al.[9] report requiring three months to reproduce a neuroscientific model of the basal ganglia.

We experienced similar reproducibility issues first-hand when we translated the Seidel-Herzel model (SHM)—a macro-level model of the human baroreflex, which is able to simulate many disease conditions and exhibits interesting dynamical properties—into a form that would be more amenable to extension and reuse[10–12]. Even though we could reach out to the author of the model to obtain his original implementation in C, the translation process was still quite challenging. The C code was monolithic and imperative in nature, describing calculation steps instead of mathematical relations and containing details that where not described in the corresponding PhD thesis. We had to carefully extract the meaning of each line of code in order to build a modular, declarative version that produced the same simulation results. For this task, we chose the modeling language Modelica, since it provides a lot of flexibility for modular model design. However, when we wanted to extend the model with a trigger for premature ventricular contractions (PVCs), it turned out that even our new version was not suitable for reuse. In fact, the component that described the cardiac conduction system remained monolithic and lacked a graphical representation, which made it hard to identify the equations and variables that would have to be changed. This situation—having to untangle the semantics and code of an existing model to extend or adjust it for use in a different context—is emblematic of the challenges faced when building multi-scale and especially multi-level models. Our example shows that issues with reproducibility and reuse reach down to the engineering level. The modeling language and the design principles applied to the construction of a model can facilitate or hamper further use. This also holds for the aforementioned case of Topalidou et al.[9], since the original model

[1]Technische Hochschule Mittelhessen – University of Applied Sciences, Giessen, Germany. [2]Vestre Viken Hospital Trust, Kongsberg, Norway. [3]University of Oslo, Oslo, Norway. ✉email: christopher.schoelzel@mni.thm.de

Published in partnership with the Systems Biology Institute

npj nature partner journals

was implemented in Delphi, which is also an imperative language and, therefore, not well suited for mathematical modeling.

Even though the need for design principles on the engineering level is apparent, most publications about best practices for reproducibility and reusability do not address it. Instead, existing approaches broadly fall into three (overlapping) categories. They tend to focus on (a) biological validity[13–18], (b) high-level choices of modeling formalisms and techniques[19–21], or (c) model documentation, annotation, and distribution[7,22,23]. Apart from these general discussions about reusability, there also are authors who advocate for individual modeling techniques, such as modular model design[24] in CellML or coupling models via semantic annotations[25]. However, while the latter work is applicable to multiple languages, it only focuses on one, albeit central, part of model design, namely the composition of multiple models or model parts. This means that researchers who want to select a suitable language for their modeling task still have little guidance available. The best assistance for choosing a modeling language currently comes from the list of accepted standards published by the Computational Modeling in Biology Network (COMBINE)[22,26]. The COMBINE suggests to use CellML and the Systems Biology Markup Language (SBML) with the main reason that these are standard exchange formats that have a high interoperability among several tools. While this is true and a great improvement over the previous state of the art, standardization, and interoperability alone cannot guarantee reusable model design. For example, the BioModels database[27] features curated models in SBML format, but most of these models are monolithic and therefore require further modification if only parts of the model should be reused[28]. In the aforementioned reproducibility study, the curators of this database found that the reproducibility rate for SBML models was only slightly higher (56%) than the overall rate of reproducibility across all models (51% including models written in SBML, MATLAB, Python, C, R, and other languages)[8]. Our previous example of the translation of the SHM also shows that using a suitable language is a necessary but not sufficient criterion for the model to actually be reusable. Additionally, no single language or even a small set of prescribed languages is likely to cover all use cases which may arise in systems biology, especially when considering multi-class models, which combine entirely different model formalisms[7].

Even when the discussion is restricted to the formalisms of ordinary differential equations (ODEs) and discrete events, there are a multitude of languages to choose from. As mentioned above, the COMBINE lists SBML and CellML as accepted standard languages. Both are markup languages based on the eXtensible Markup Language (XML) and designed to be written and read by software tools and not directly by humans. While SBML has a clear focus on metabolism and cell signaling models, CellML, despite its name, is not targeted toward a specific level of organization. MATLAB is a proprietary domain-specific programming language designed for scientific computing in general, which is also popular in systems biology (https://www.mathworks.com/products/matlab.html). It provides an environment for graphical block diagrams called Simulink (https://www.mathworks.com/products/simulink.html) and a declarative language for designing physical systems called Simscape (https://www.mathworks.com/products/simscape.html). The MATLAB environment SimBiology is another alternative based on block diagrams, which is targeted toward pharmacological models, but can, like SBML, model arbitrary ODE-based dynamical systems (https://de.mathworks.com/products/simbiology.html). While MATLAB is still popular[29,30], the open-source programming language Python also gains increasing interest in the community[29,31–34]. Usually models are not built in Python itself, but researchers have created packages such as PySB[33] and the Python Simulator for Cellular systems (PySCeS)[31] that define embedded domain-specific languages (DSLs) which facilitate the creation of mathematical models for specific use cases. With Tellurium[34] there also exists a broader python-based environment that supports multiple COMBINE standards and uses the declarative modeling language Antimony[35]. Another emerging language for the definition of embedded DSLs for mathematical models is Julia, which has a similar focus as Python but is more extensible and tends to have better runtime performance[36]. Finally, Modelica is an open-source declarative modeling language primarily used in engineering[37]. It has a large user base both in industry and research, but is still largely unknown in systems biology. Notable exception include the Physiolibrary[38]—a Modelica library for physiological models—and SBML2Modelica[39]—a tool that translates SBML models to Modelica. This extensive list of language candidates makes it apparent that researchers need guidelines to choose between these candidates and to write model code that actually uses the desirable characteristics of the chosen language.

In recent years there has been increasing interest to apply techniques from software engineering (such as unit testing, version control, or object-oriented programming) to modeling in systems biology[28,30,40]. Hellerstein et al.[40] even go so far to suggest that systems biologists should rethink the whole modeling process as "model engineering". To date they are the only authors that we are aware of who actually give explicit guidelines for how to write model code (e.g., they suggest to use human-readable variable names).

In this article we share our experience with extending the SHM and generalize our findings from this example to expand on the idea of model engineering in three ways: First, we propose a list of desirable characteristics that make a model language suitable for building reusable multi-scale models. Second, we give guidelines on how these characteristics can be exploited during model design to increase the reproducibility and reusability of a particular model. Third, we compare state-of-the-art language candidates with respect to the aforementioned characteristics.

From these candidates, we chose one, namely Modelica, to demonstrate the reasoning behind the characteristics, guidelines, and language assessment using the example of the cardiac conduction system within the SHM. We transform the existing monolithic model into a modular structure and show how this facilitates the PVC extension. After we present our results we reflect on the impact that each of the language characteristics and the choice of Modelica, in particular, has on the usefulness of the model.

RESULTS

Desirable characteristics for a mathematical modeling language for systems biology

The following characteristics were developed from literature review and/or from our personal experience with Modelica and the SHM. The goal of these characteristics is to facilitate the creation and analysis of multi-scale, multi-level, and multi-class models. We, therefore, focus on increasing reproducibility, understandability, reusability, and extensibility. The resulting characteristics are that a modeling language should be modular, human-readable, hybrid, open, declarative, and graphical. Each characteristic will be introduced with arguments for its usefulness, a brief set of guidelines on how it may be applied to full effect, examples where this was relevant in our implementation of the SHM, and references to other authors that advocate this feature. To easily refer to these characteristics we form the mnemonic MoDROGH for Modular, Descriptive, human-Readable, Open, Graphical, and Hybrid. We will also use the term "MoDROGH language" for a language that exhibits all or most of these characteristics.

MoDROGH characteristics: Modular

In order to replace or reuse parts of a model, they have to be identified in the code. The modeling language should make this as easy as possible, using separable components with clear interfaces. The number of variables in the interface should be minimal, encapsulating internal implementation details so that using and connecting the component becomes as easy as possible. Some languages facilitate this by allowing the definition of connector components, which group interface variables together, so that the interface has, e.g., a single electrical pin connector instead of two separate variables for current and voltage. Interfaces are important to define intended biological transitions between model components and to document assumptions, even if rigid interfaces can limit reuse. It can even be argued that it is beneficial if a component cannot easily be reused in an environment with different assumptions, since such a switch of assumptions will likely require more change than adding a variable to the interface. For quick experimentation, it can be an advantage if the language allows connecting arbitrary internal variables of components, but published versions of a model should always have a clear interface concept to remain understandable.

Modularization and encapsulation are reliable tools to handle complexity in large software projects, so it is reasonable to expect that they will also be able to manage the complexity of biological systems. Modularity also inherently facilitates reusability, since clearly defined self-contained modules are easier to reuse than a set of equations that has to be extracted from a tightly coupled model. To allow reuse of components within the same model, it must be possible to import multiple instances of a module and assign individual identifiers to them. This can be further facilitated by supporting full object-orientation, allowing a component to inherit variables, equations, and possibly annotations from one or more parent components, which define common structure and behavior. Additionally, components are also easier to reuse if individual variables and equations may be overwritten or removed during instantiation and inheritance. Some languages also allow the reuse of models across different languages, tools, and platforms by using a standardized exchange format or a standardized interface. In systems biology, SBML is a standard exchange format for hundreds of tools, allowing the use of models in a multitude of different contexts often through the use of translators that convert SBML to different languages. In contrast, the Functional Mock-up Interface (FMI) is a model exchange format maintained by the Modelica Association[41,42] that focuses more on a unifying interface than a unified language. It is not used to translate models into other languages, but rather to distribute models in an encapsulated format that is independent of the underlying formalism, which is especially interesting for multi-class models.

Guidelines. Modules should be small enough to be understandable at first glance, but still self-contained. If a formula or concept is used multiple times in a model, it should be defined as a module once and then referenced. In software engineering this concept is called DRY for "don't repeat yourself". Modules should have clearly defined, minimal interfaces, which explicitly state possible connection points to the outside world. Both modules and their interfaces should follow the biological structure of the system. If a module represents more than one biological entity or an equation in the module conflates effects from multiple distinct causes, it might be worth to investigate whether splitting up the corresponding module further might increase its understandability and flexibility for reuse and extension. Interfaces should represent the transfer of some physical quantity between biological entities and should only expose variables whose meanings are clear and do not require an understanding of the module's internal organization or function. If possible, each module should be tested individually, which is called a "unit test" in software engineering.

Importance in SHM modeling task. Since the SHM features a multitude of feedback loops, locating errors was very tedious with the original monolithic model. Systematic debugging became only possible when we isolated the different parts of the system, such as the baroreceptors, and subjected them to controlled input signals to observe the component output. It was also possible to reuse several components within the SHM: the parasympathetic and the sympathetic system share a base class that only leaves the sign of the baroreceptor influence open for definition and the four different release equations for norepinephrine and acetylcholine are also governed by a common base class.

References. There is a consensus that multi-scale modeling requires some form of modularity for hierarchical composition[24,25,28,43–45]. More specifically, Hellerstein et al.[40] and Mulugeta et al.[30] both suggest that object-oriented programming might be an especially promising way to implement modularity. Many researchers advocate for clearly defined interfaces[2,28,44,46], but there is also critique with regard to a loss of flexibility for reuse and the requirement to consider all code-level elements of a model as potential coupling points[25,29].

MoDROGH characteristics: human-readable

This characteristic covers two loosely connected aspects: The fundamental readability of model files with a text editor and the readability and understandability of definitions within the model.

Every modeling language has to be both human-readable so that a human can write the code to define a model and machine-readable so that a software tool can interpret that code to run simulations. However, as Fig. 1 shows, there is a trade-off between the two and languages can choose to support the one at the cost of the other. On the one end of the spectrum, languages like Antimony or Modelica, whose syntax is closer to natural language and easier to read and write for humans using just a text editor, require more effort for specialized parsers to build abstract syntax trees, which can then be processed by compilers and other software tools. The middle ground is formed by XML-based languages like SBML and CellML. XML files already have a tree structure and parsers for XML exist for virtually all modern

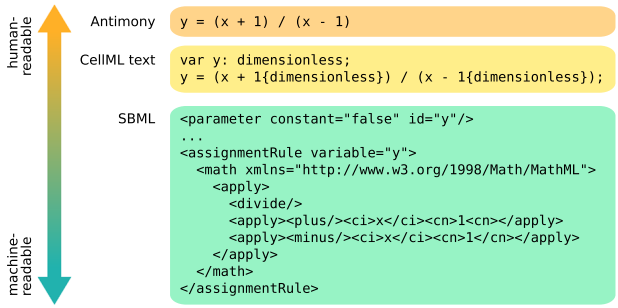


Fig. 1 Simple variable definition with assignment rule in three modeling languages with different levels of focus on human- versus machine-readability. Each of the three code snippets contains the same information defining a variable y, which depends on another preexisting variable x. Antimony mainly focuses on how humans would write equations in text form, but requires a specialized parser. CellML Text—an intermediary editing language used by the tool OpenCOR[69]—adds some syntax that is easy to parse by a machine (due to using braces that do not conflict with other symbols in the code), but is not an intuitive representation of unit information for a human unfamiliar with the language. SBML focuses more on machine-readability, since XML can be parsed by the standard libraries of most modern programming languages, ensuring minimal barriers for tool support. However, while the SBML code is still readable and editable in a text editor, it takes some effort and familiarity with the language to decipher the meaning from the symbols.

a)

```
DSargs = args()
DSargs.name = "LotkaVolterra"
DSargs.ics = {
  'x': 10, # prey
  'y': 10  # predator
}
DSargs.pars = {
  'alpha': 1.1,
  'beta': 0.4,
  'delta': 0.1,
  'gamma': 0.4
}
DSargs.tdata = [0, 20]
DSargs.varspecs = {
  'x': 'alpha*x - beta*x*y',
  'y': 'delta*x*y - gamma*y'
}
```

b)

```
model LotkaVolterra "predator-prey model"
  Real x(start=10, fixed=true) "prey pop.";
  Real y(start=10, fixed=true) "pred. pop.";
  parameter Real alpha = 1.1 "prey birth";
  parameter Real beta = 0.4 "prey death";
  parameter Real delta = 0.1 "predator birth";
  parameter Real gamma = 0.4 "predator death";
equation
  der(x) = alpha * x - beta * x * y;
  der(y) = delta * x * y - gamma * x;
annotation(Documentation(info="<html>
  This model implements the <a href=\"https://
    en.wikipedia.org/wiki/Lotka%E2%80%93
    Volterra_equations\">Lotka-Volterra
    equations</a>.
</html>"));
end LotkaVolterra;
```

Fig. 2 Simple predator-prey model in a language without (PyDSTool) and with (Modelica) support for documentation strings. a While regular Python comments (#) can be used to annotate PyDSTool models, they are ignored by the compiler and are only useful when reading the code directly. **b** Modelica comments are part of the model syntax and can therefore be read by tools to, e.g., provide automated tooltips in dialogs and graphs or to enrich model summaries in databases.

programming languages, which lowers the barrier to implement support for an XML-based format in a software tool and, therefore, increases interoperability between tools. While XML files can still be viewed and edited in a text editor, this requires familiarity with the language and tends to be cumbersome for larger edits. Especially the Mathematical Markup Language (MathML) format used both by SBML and CellML for storing equations can be hard to write and decipher without tool assistance. SBML and CellML, therefore, rely on software tools that use graphical interfaces or intermediary languages to ease model editing. On the machine-readable end of the spectrum, MATLAB Simulink uses a proprietary binary format that is tailored specifically to the MATLAB software toolchain. This can both reduce storage space and implementation effort for parsers, but also means that it is impossible to inspect model files without the corresponding software.

For model exchange and interoperability between different tools, XML-based formats have the clear advantage that supporting their import or export in a tool requires very little programming effort. This is illustrated by the success of SBML and FMI, which are both based on XML and are supported by over 100 tools each (http://sbml.org/SBML_Software_Guide/SBML_Software_Matrix, https://fmi-standard.org/tools). Increased interoperability also facilitates model reusability, because it becomes more likely, that a researcher who wants to reuse a model can simply import it in their tool of choice without having to translate it to another language first.

However, for model development and for publishing models to other researchers, languages with a strong focus on human-readability are preferable, because they allow tool-independent access to a model and because they are more suitable for version control. Due to their verbose syntax, XML-based languages are typically not designed to be written by humans directly but by software tools, which provide intermediary languages or graphical interfaces to facilitate editing. The translation between these different representations is performed automatically during export and import, which is convenient, but if the feature sets of the exporting and the importing tool do not overlap completely, there is a risk that information is lost. For example, a SBML model written with tool A may include layout information for a graphical representation of the model, but when it is loaded in tool B, which uses a purely equation-based representation, this layout information may be discarded. If tool B does not show a warning message, there is no way for the user to know that the model contained this information unless they look at the SBML code itself. This problem is less likely to occur, if the model is written in a language more focused toward human-readability, which is then also used directly for editing. In this case, both tool A and tool B would display the same code and while tool B does not display the graphical representation, the user would notice the presence of the layout annotations and could choose to view the model in a tool that does support them. Additionally, the more a language focuses on human-readability, the more easily it can be translated to slides, websites, articles, and other formats, which makes it easier to communicate the details of a model to other researchers. It can also be archived more safely, as it will still be easily readable decades into the future, even if the tools used to create it and to view its contents will not be available anymore. Finally, version control software can be immensely helpful for tracking errors, for finding the exact versions of a model used to generate plots in an article, and for understanding the rationale behind modeling design choices. Standard solutions like Git operate under the assumption, that the documents under version control are written by humans and that element order, white space, and other details all are results of deliberate choices and, therefore, carry meaning. This is not the case for XML-based documents written by tools, which can artificially inflate changesets between document versions with management data or structural changes that carry no meaning and therefore obscure the changes that are actually relevant. While there are specialized solutions to distinguish semantic from structural changes in XML documents[47], this is still an active field of research and not yet broadly implemented in version control software. Also, even with these solutions, researchers might only be familiar with the tool that generated the file and not the content of the file itself, which makes it harder for them to localize and understand changes between model versions.

It is possible to combine the benefits of XML-based exchange formats and languages that focus more on human-readability, if these exchange formats are used and published *in addition* to a more human-readable representation of a model. This can be seen as analogous to software packages in general purpose programming languages. Open source software projects are usually both published as some kind of easily installable artifact—a file that not even has to be human-readable at all—and also as human-readable code in an online repository, which can be used to analyze and extend the software.

Moving from the question of the general file format to the content of the file, it can be said that readable code is largely the responsibility of its authors. However, a language may facilitate a clean coding style by providing expressive language constructs and documentation features or hinder it by introducing visual

clutter. One example for this is the verbose use of `{dimensionless}` that is required after each constant in an equation in CellML Text as seen in Fig. 1. Additionally, languages can also add human-readable documentation strings to variables and components or incorporate an HTML document for a more detailed model description. In contrast to comments in traditional programming languages, which are ignored by the compiler, these documentation features can enrich model presentation across various tools and representations including graphical dialogs or HTML representations within a model database. An example for this can be seen in Fig. 2.

Guidelines. Model files stored in a version-controlled repository and published in model databases should be written in languages that focus on human-readability. If possible, models should additionally be published in a more easily machine-readable exchange format like XML to lower the barrier for direct reuse. If the language has support for structured documentation that is semantically tied to individual components or variables, this form of documentation should be preferred over unstructured comments. Every parameter, variable, and model component should at least be documented with a short human-readable label. Any non-obvious design choices or complex equations should also be documented.

Importance in SHM modeling task. On several occasions during our implementation, we accidentally introduced errors in one part of the model while correcting an issue in a different part. To recover from these errors, it was crucial that we could quickly skip through the changes made since the last known working version. This was facilitated by the fact that Modelica focuses on keeping model files easily human-readable. With an XML-based format, we would have had more difficulties to make sense of the differences between versions.

References. Hellerstein et al.[40] and Zhu et al.[43] stress the importance of keeping model files under version control. The authors of Tellurium specifically state that human-readable languages can facilitate reproducibility and exchangeability[34]. Dräger et al.[48] found that existing tools struggle to make all the information in the XML-based description of SBML models accessible in a comprehensive form, which led them to develop a tool called SBML2LaTeX, which generates human-readable reports from SBML models.

MoDROGH characteristics: hybrid

A language is *hybrid* if it supports multiple modeling formalisms and thus multi-class models. The most common form of hybrid models and languages cover both continuous ODE or differential algebraic equations (DAEs) and discrete events, but other combinations are possible. The distinction between ODE and DAEs is important here since physical conservation laws, such as conservation of mass or energy, are *algebraic* constraints, which cannot always be formulated with pure ODE. Incorporating them in a model can, however, have the benefit of making connections between components *acausal*, which means that variables do not have to be designated as input or output and instead the solver can choose the appropriate resolution order. This avoids errors and performance issues related to algebraic loops, simplifies model descriptions, and allows reusing components in different contexts. As with DAEs, support for discrete events also comes in different forms. Many languages support the reinitialization of continuous variables through discrete events. In this formalism the only discrete part of the model is a set of equations that define boolean values based on the state of the system. When these values switch from false to true, events are generated, which can introduce discontinuities in an otherwise continuous system. For a fully hybrid model that involves more complex discrete parts it is preferable that the language also supports the explicit declaration of discrete variables. The value of these variables remains constant between events, but they may be defined with complex equation systems, which are solved during each event instance. As a result, they can make models that require complex event triggers more understandable as can be seen in Fig. 3.

It is important to note that we do not argue that a modeling language should support as many formalisms as possible, but rather a combination of formalisms that go well together. If other formalisms are required, the language should rather aim to allow the coupling of models across languages with standardized interfaces such as the FMI[41,42]. Additionally, there is a trade-off between fully supporting a modeling formalism, such as ODE or DAEs and being able to assign a domain-specific meaning to language constructs. For example, SBML, PySB, and Antimony all use biological terms tied to the biochemical level to describe the parts of a model. This makes the language easier to understand and use for domain experts, but may prove challenging when building a multi-scale model that has to extend beyond the biochemical level.

Guidelines. A model should clearly indicate which variables are discrete and which are continuous. Event triggers, which define the transition between discrete and continuous parts of the model, should be examined and tested with extra care. If the language allows them, acausal connections should be preferred over causal input-output relationships, since acausality facilitates reuse.

a)

```
1  ...
2    Boolean in_window = time < event+2;
3    discrete Real x_max;
4  initial equation
5    in_window = false;
6    x_max = x;
7  equation
8    when in_window and der(x) < 0 then
9      x_max = max(x, pre(x_max));
10   end when;
11 ...
```

b)

```
...
in_window = 0;
in_window' = 0;
at (time < event+2): in_window = 1;
at (time >= event+2): in_window = 0;

x_max = x;
x_max' = 0;
at ((in_window > 0) && (x' < 0)):
  x_max = max(x, x_max);
...
```

Fig. 3 Definition of a discrete variable `x_max` in a language with (Modelica) and without (Antimony) support for declaring discrete variables. The variable `x_max` captures the peak value of the continuous variable `x` obtained within two seconds after an event `event`. **a** The Modelica model defines a discrete boolean variable `in_window`, to simplify the `when` condition later in the code. The information that this variable is discrete already lies in the type definition as `Boolean`. For real variables like `x_max`, there exists a keyword `discrete`, which determines that the variable value may only change within a `when` equation. **b** The same model structure and semantics can also be achieved in Antimony, but the discrete variables `in_window` and `x_max` each need an explicit rate rule to ensure that their value only changes when an event occurs (lines 3 and 8). Additionally, two events are needed to emulate the boolean variable `in_window`: One to update the value when the condition becomes true (line 4) and one to do so when it becomes false (line 5).

Importance in SHM modeling task. At first, it was not clear for us whether the contractility of the heart in the SHM was a continuous or discrete variable. This confusion led to a severe error in an early version of the model. Our current implementation defines the variable with the keywords `discrete Real` to clearly indicate this distinction. Discrete variables were also required to disentangle the semantics of the cardiac conduction system, which is introduced in detail in the "Results" section.

References. In their 2017 review, Bardini et al.[2] argue that multi-scale models in systems biology, in general, should strive toward a hybrid approach. The same has also previously been stated by other researchers[49–51]. In particular, the authors of PyDSTool argue that hybrid models based on DAEs are well-suited to represent multi-scale models[32].

MoDROGH characteristics: open

As a prerequisite for reproducibility and collaboration, models and simulation tools need to be accessible for everybody. In particular, the hurdle to run a quick simulation of a model to determine its usefulness for a specific task should be as low as possible. An openly accessible model definition also means that readers can offer feedback and corrections to improve the model. Preferably the language itself, the compiler and associated tools should all have an open-source license. Additionally, collaboration is also facilitated if the language can be used on different platforms.

Guidelines. Readers of a paper should be able to download the model code and to simulate it with open-source tools. The download should also include explicit licensing information. The model repository should include everything necessary to reproduce plots and other results of the corresponding paper. It should also be under version control and include a human-readable changelog. Other researchers should be able to point out errors and suggest corrections.

Importance in SHM modeling task. Without the reference code of Seidel, our re-implementation of the SHM would not have achieved a perfect agreement with the original. To weed out our last errors, which only showed quantitative and not qualitative differences in the plots, we needed to simulate both models with identical solver settings and manually compare the output data. Additionally, some small errors in the published formulas became only apparent when we compared them with their C implementation.

References. Many large projects and databases such as the Physiome Model Repository of the IUPS Physiome project[52], the NSR Physiome project[53], the BioModels database[27], the virtual liver network[54], Plants in silico[43] and SEEK[23] already provide open-source implementations of models. Mulugeta et al.[30] also specifically advocate for more version control and changelogs (in the form of e-notebooks).

MoDROGH characteristics: declarative

The mathematical formalism for biological models can already be complicated in itself. A modeling language should not require the adaptation of the model to the execution logic of the language, obscuring the original definition. Instead, the language should adapt to the model if it is presented in a clean mathematical formulation. This way the code can focus on expressing meaning rather than structure, which facilitates understanding. This also includes the possibility to formulate ODE and DAEs not only in explicit form, i.e., with a single variable on the left-hand side of the equation, but also in implicit form, i.e., with arbitrary mathematical terms on the left and right-hand side. For example, specifying `u = r * i` should be equivalent to the equation `u / r = i` and the solver should decide for which variable this equation needs to be solved.

As a consequence of such a declarative style, errors reported by the compiler can also focus on meaning rather than just grammar, increasing the soundness of the model. One important example of this is that declarative languages usually allow the declaration and automatic checking of proper units for variables and parameters. Missing or wrong unit conversions are a common source of error in modeling that can be all but eliminated this way. Unit definitions also add semantic information and therefore make the model more understandable. Additionally, if the model is described in a declarative style, it is possible for automated tools to identify and extract meaningful parts of the model. This facilitates tool support—e.g., in the form of numerical solvers, optimization, and verification toolchains—and also allows connecting model parts to standardized ontological terms. For the latter it is preferable if the support for ontologies is already included in the language itself.

Guidelines. Models should follow strict mathematical rules to fit nicely into the chosen formalism. If the language allows it, equations should be written exactly as one would write them in a scientific paper to convey their meaning, choosing freely between explicit and implicit form. If a model needs workarounds in the form of code that has to be added to make the model compile but that does not add new information about the modeled system, it may be worthwhile to revisit design choices and check the mathematical soundness of the model. In our own experience we found that most workarounds could be removed and the resulting model behaved more soundly and was easier to understand. Models should also specify units for all variables, preferably using the International System of Units (SI). If possible, automated unit consistency checks should be performed before publishing a model. Additionally, if the language supports semantic annotation with ontological terms, this feature should be used for all variables and components.

Importance in SHM modeling task. The original SHM was implemented in C, which is an imperative language. Most of the reference code that we consulted for our re-implementation was responsible for management tasks, such as storing a history of variables that enter equations with a delay, debugging output, or a manual implementation of an integration loop with the Runge-Kutta method. Although most equations were defined as separate functions, we sometimes had difficulties untangling the semantics of the model from the main integration loop.

One area of the model that was highlighted by the Modelica compiler as not mathematically sound was the systemic arterial blood pressure, which is given by an algebraic equation during systole and by an entirely separate differential equation in diastole. This issue only became apparent, because we had to translate the imperative C code, which simply used an if-expression to switch between the two states, into a declarative form, which required a consistent equation structure. This consistent structure could be established by manually differentiating the systolic equation and then only switching between two different expressions for the derivative.

References. Few researchers in systems biology explicitly distinguish between imperative and declarative languages. Zhu et al.[55] state that declarative languages are desirable, because it allows the description of the biological processes "in a natural way". Several language authors also state that their respective modeling language is declarative[29,33,56,57], but they do not explain why this is important. Of these, only the authors of JSim[57] and Myokit[29] state that declarative languages allow concentrating on what is modeled and not how the equations are solved, make models more understandable, and facilitate their analysis both by researchers and software tools.

MoDROGH characteristics: graphical

Discussing or even just understanding a model is difficult if the model is only described in the form of code or mathematical equations. This is especially true when the input of domain experts is required, who are not computer scientists or mathematicians. For this purpose most papers in biology use some kind of diagram to transport the general structure of the model in a graphical way. Here, there is a trade-off between two different visualization types:

1. Automatically generated abstract graphs of variable dependencies are an exact representation of the model and are well-supported by tools, which reduces the effort required to build these representations. However, automatic graph layout is a nontrivial problem: Different algorithms or parameter settings can lead to large differences in the layout[58]. Most algorithms also do not scale well to large graphs and additional techniques are required to group nodes according to semantic similarity[59]. Additionally, to the lack of grouping capabilities, automatic graph visualizations also solely rely on the variable names to convey the role of a variable—e.g., whether it is the product, reactant or catalyst of a reaction—or the kind of variable interactions—e.g., if the correlation is positive or negative. Consequently, this approach is mainly suited to represent the mathematical dependencies of variables, but not to give an intuitive overview of the model structure or to analyze the biological relations between modeled concepts.
2. Manual drawings of the biological interactions with respective images and symbols capture the essence of the information required to understand the model and can quickly be processed by the reader. This also has the additional benefit that the model can be discussed with domain experts that are familiar with the biological concepts, but not with mathematical modeling. However, they are less accurate, not standardized, and require a lot of manual effort. This can also mean that when a model is extended or otherwise updated, changes may not be immediately reflected in the drawing, since it may only be updated at a later stage or not at all.

There are multiple hybrid approaches that try to address the shortcomings of pure type 1 or type 2 visualization. The Systems Biology Graphical Notation (SBGN)[60] allows the illustration of models with standardized abstract structural diagrams, which serve a similar function as circuit diagrams in electrical engineering. SBGN diagrams do not display variables, but represent the actual physical entities and processes with unambiguous, standardized glyphs. While there exist tools that can generate SBGN diagrams automatically, like CySBGN[61], some manual arrangement is required to produce satisfactory results. The standardization of SBGN also comes at the expense of the biological intuitiveness of the resulting diagram. Instead of immediately recognizable biological icons, researchers have to learn and interpret a series of abstract glyphs. For metabolism pathways this is no issue, since species that are part of a reaction are typically identified by their name and not by any two- or three-dimensional structure that could be used as an icon. However, e.g., for action potential models it would be preferable to represent ion channels and pumps by schematic drawings and to have a visual separation between the inside and outside of a cell. Such a graphical representation is especially helpful if it is standardized across different models. For example, the Physiome Model Repository[52] uses the same icons for ion channels and pumps across all curated action potential models, which are drawn by the now discontinued tool OpenCell (http://physiomeproject.org/software/opencell/about). A similar standardized icon language could also be beneficial for models at the tissue or organ level.

Like type 1 and type 2 visualizations, SBGN graphs are independent of the capabilities of the language with which the model was written. They are generated by tools that do not need to have any connection with the modeling language itself. Modeling languages can support them by referencing image files or XML files containing SBGN as part of the model documentation, but they have to be maintained separately. An example of this can be seen on the left side in Fig. 4.

This is addressed by another hybrid approach that goes a step further toward the analogy with circuit diagrams and integrates layout and rendering information directly into the model structure. It is mainly prevalent in languages with an industrial background, such as Modelica and MATLAB, but is also implemented in the SBML level 3 layout and rendering packages[62,63]. In this approach, model components are assigned graphical annotations, which define how the component should look and where it should be placed in the diagram representation

a)

```
...
<informalfigure float="0" id="frd">
  <mediaobject>
    <imageobject>
      <objectinfo>
        <title>
          model diagram
        </title>
      </objectinfo>
      <imagedata
        fileref="model.png"/>
    </imageobject>
  </mediaobject>
  <caption>
    Diagram of the XYZ model.
  </caption>
</informalfigure>
...
```

b)

```
model MyModel
  //(2) component location in diagram
  MyComponent comp annotation(
    Placement(transformation(
      extent = {{0, 0}, {20, 20}}
    ))
  );
  ...
equation
  //(3) line connecting two components
  connect(comp.c, other.c) annotation(
    Line(points={{10, 0}, {10, 100}})
  );
  ...
//(1) component icon as vector graphic
annotation(Icon(graphics={
  Rectangle(extent={{0,0},{100,10}})
}));
end MyModel;
```

Fig. 4 Two different ways in which modeling languages can support graphical representations of models as part of their syntax. a CellML allows to include diagrams or plots as figures in the model documentation. The image files remain separate from the model code and have no semantic connection to it except for the figure caption. **b** Modelica allows to add graphical annotations using a vector graphics syntax. Models and their components can have icons graphics (*//(1)*), which can be placed in a diagram coordinate system (*//(2)*) and connected with lines (*//(3)*). This graphical representation is tied to the structure of the model. If, e.g., a component is removed from a model, the placement annotation (*//(2)*) must also be removed, which automatically updates the diagram and ensures that it still accurately reflects the new model structure.

of the model. In a modular language, this information can be used to build tools that allow to construct models by dragging and dropping component icons and connecting them with lines, much like a circuit diagram. An example of this can be seen on the right side in Fig. 4. The resulting diagrams are both an accurate reflection of the connections between model components, because they are intrinsically tied to the functional model code, and can be understood quickly, since they are arranged manually and use biological imagery. If the model changes and, e.g., a component is removed, the graphical annotation also has to be removed, because the compiler would otherwise produce an error message. This ensures that graphical representations stay up to date when a model is changed. Creating symbols and images for components requires effort, but this has to be done only once for each component and the arrangement and connection may even be easier than writing the equations that connect the components in code.

As becomes apparent, this last approach should be favored for multi-scale models, although it has to be noted that it is also possible to combine multiple approaches in the same language.

Guidelines. All interactions between the individual modules in a model should have a graphical representation in the corresponding diagram. Each diagram should only have a few components. If it becomes too crowded, some components should be grouped together to form a hierarchical structure. Each individual component in the diagram should be represented with an intuitive symbol that either corresponds to the appearance or function of its biological equivalent. Components should be visually grouped according to their function and interaction to facilitate understanding.

Importance in SHM modeling task. The original SHM features a graphical representation in the form of 23 text boxes that are connected by arrows. While this does give an overview of the physiological effects present in the model, one of our first steps to better understand the model was to augment this diagram by grouping the effects by the organs to which they belong and adding respective icons. Our Modelica implementation now features a fully visual diagram with 15 components that is guaranteed to be faithful, since it is tied to the equations in the code. It helped us on several occasions to discuss the model with domain experts, such as physicians and chemists.

References. The Physiolibrary is a Modelica library for physiological models that has graphical representations for each component[38]. ProMoT is a modeling tool that allows the composition of modular models in a graphical way[44]. Alves et al.[64] compare 12 different simulator tools, giving higher ratings to those that have graphical representations for model components. Mangourova et al.[65] state that it is preferable when a modeling tool for integrative physiology provides a graphical way of composing models since this can reduce development time.

Existing languages exhibit MoDROGH characteristics to varying extent

As mentioned in the introduction, there are multiple suitable languages available that implement the MoDROGH characteristics to some extent. In this section, the most prominent examples will be discussed with respect to each characteristic (highlighted in *italics*). We consider a "modeling language" to be any language used for describing and distributing mathematical models. This includes exchange formats such as CellML and SBML, languages that are embedded in a general-purpose programming language like Python or Julia, and standalone languages like Modelica. We selected languages that are currently popular either in systems biology or in mathematical modeling in general with a tendency toward general-purpose modeling languages that are not restricted to a specific organizational level or model type. As an additional criterion, languages had to exhibit at least some MoDROGH characteristics. We have to emphasize that the list is not comprehensive, but we tried to cover examples for all major trends in modeling languages.

MoDROGH languages: MATLAB

MATLAB is perhaps the most widely known language used for solving ODE (https://www.mathworks.com/products/matlab.html). The MoDROGH criteria can be best fulfilled when using the Simulink environment (https://www.mathworks.com/products/simulink.html) with the embedded language Simscape (https://www.mathworks.com/products/simscape.html). The SimBiology environment can be used as an alternative, which is comparable to SBML in its expressiveness regarding rules and reactions and can also export models to SBML (https://de.mathworks.com/products/simbiology.html). It is, however, tailored toward pharmacological models and not as feature-rich as Simulink and Simscape, which is why we restrict our analysis on the latter combination. Simscape realizes *modularity* through full object-orientation with class definitions, instantiation, and inheritance, although Simscape classes can only have one parent class, in contrast to MATLAB classes, which allow multiple inheritance. Through Simulink, models can be imported from different languages using the FMI, but export of Simscape models with this interface is currently not supported. The language is also *declarative* allowing to freely mix between implicit and explicit formulation of DAEs, which are written in a concise syntax that focuses on *human-readability*. It supports documentation strings for components and human-readable labels for variables. Unfortunately, the readability of Simscape is hampered by the fact that Simscape has to be used in conjunction with Simulink, which saves models in a proprietary binary format, which is not readable in a text editor and not even openly documented. This issue is further aggravated by the fact that backward compatibility to older versions of the model format is not guaranteed[65]. Units are supported and a mandatory consistency check is performed at the interfaces between components. There is no built-in support for ontologies, but since Simscape supports object-orientation, "is-a" relationships, which designate a component as an instance of a concept, might be expressed by building a large type hierarchy of ontological terms. This would require all models to use this type hierarchy and, therefore, reduce flexibility in designing generic base classes, since Simscape only allows single inheritance. Simscape classes can be used as *graphical* components within Simulink to create larger systems by arranging and connecting them via drag and drop. *Hybrid* systems are supported with index-reduction for DAEs and discrete events and variables. Unfortunately MATLAB, Simulink, and Simscape are proprietary tools that are not *open* in any way, requiring license fees and prohibiting custom extensions.

MoDROGH languages: SBML

In systems biology, the SBML is a widely-used *open* language for describing biological models—mostly at the level of biochemical pathways[66]. SBML level 3 includes an optional language module for hierarchical composition, which allows building *modular* models via the import of components during which individual variables can also be overwritten or deleted. Because it uses a subset of MathML to describe equations, SBML is *declarative*, and *hybrid* and in theory allows the definition of arbitrary DAEs in explicit and implicit form. SBML is based on XML, which makes it highly machine-readable and in turn facilitates interoperability between tools, because support for model import or export can be implemented easily. Unit definition is possible but optional and tools are not required to interpret them. However, libSBML, the

most popular library for working with SBML models, can perform automatic unit consistency checks[67]. Support for discrete events is limited to reinitialization of continuous variables. The reliance on MathML and XML is also a drawback, because it limits the *human-readability* of model files and presents challenges for version control software that is not equipped to distinguish structural from semantical changes. Individual components can be annotated with textual notes, Systems Biology Ontology (SBO) terms, or Minimal Information Required In the Annotation of Models (MIRIAM) metadata. Using the SBML level 3 packages for layout and rendering, *graphical* annotations can be assigned to model components. The high interoperability between SBML tools resulting from its focus on machine-readability is a major advantage, because researchers can use a tool that is designed to fit their specific use case and can reuse models across tools. Due to the wide acceptance of SBML, it can be expected that most researchers will have at least one such tool available so that the visual clutter of the XML files is no issue for model reuse. However, most of these tools do not support all optional SBML packages with the consequence that in practice support for modularity, graphical annotations, and DAEs in implicit or explicit form may be limited to specific tools.

MoDROGH languages: CellML

CellML is similar to SBML, but focuses on building more general component-based models[68]. It is also *open*, *declarative* and *hybrid* with the same considerations for being based on XML and MathML. In contrast to SBML, it does not only support units, but enforces that every variable in a valid CellML model must have a unit definition. Modelers can still choose the special value `dimensionless` to designate that a variable does not have a unit, but they have to make this choice consciously and explicitly. The language itself does not require tools to check the consistency of these units, but OpenCOR, one of the main tools for creating and simulating CellML models, does perform automated consistency checks when a model is loaded or saved[69]. OpenCOR can also somewhat alleviate the downside in *human-readability*, because it defines a so called "CellML Text" language, which can be used to view and manipulate the model in a more human-readable text format[69]. However, "CellML Text" has limited expressiveness only allowing the definition of explicit and not implicit equations and it is only used for viewing and editing and not for model storage. It also does not contain annotations, which can be defined in CellML through embedded metadata files in Resource Description Framework (RDF) format, which can also contain ontological annotations. Since version 1.1, *modular* CellML models can be hierarchically composed of sub-models[24,70]. To support the graphical representation of models, CellML provides constructs for referencing externally-stored graphical files and formatting figure captions. This feature allows modelers to link models with an associated image and is used by the curators of the Physiome Model Repository—the primary clearinghouse for CellML models—to display relevant figures on a model's webpage. However, as there is no semantic link between figure elements and model code, it is the responsibility of the modeler to keep the figure up to date when the model is changed.

MoDROGH languages: Python

Python is an open-source programming language that is popular in data science (http://www.python.org). The language itself is imperative, but it can be extended with some declarative features for special purposes. In systems biology, notable efforts include PySB[33] and the PySCeS[31]. These packages define their own declarative domain-specific languages (DSLs) within Python to tackle specific biological use cases. PySB focuses on rule-based reaction models while PySCeS focuses on ODE, structural analysis, and metabolic control analysis. There also exist general-purpose packages, such as SimuPy[71] and PyDSTool[32], that allow users to create and analyze models built with ODE, DAEs, and discrete events.

All aforementioned python-based solutions are *open* and *declarative* and Python itself focuses on *human-readability*. However, the modeling packages mainly rely on the modeler to use the features of Python to implement *modularity* concepts and to document their models by themselves. Also, none of them support any *graphical* representation of models. Notably, SimuPy and PyDSTool lack slightly in human-readability and declarativeness because they require a very specific and low-level technical format for defining equations. Exceptions in terms of modularity are SimuPy's block diagrams and PySB's macros. The fact that the environment is not declarative in itself leads to the drawback that only PySB supports ontological annotations and only PySCeS supports the definition (but not consistency checks) of units. Regarding the *hybrid* characteristic, the differences are most pronounced since PySB is not hybrid at all, featuring only specific biochemical rules without events, while PySCeS and SimuPy allow discrete events as well as ODE and only PyDSTool is able to also handle DAEs. None of these packages support the explicit declaration of discrete variables.

MoDROGH languages: Antimony

Antimony[35] is a *declarative* modeling language with an emphasis on *human-readability* used by the *open* Python-based environment Tellurium[34], which can be used for model building, simulation, and analysis. Since Tellurium version 2, Antimony also supports the structural annotation of models with terms from the SBO or general MIRIAM metadata. Antimony is *modular* by design, allowing the definition of components that can be imported in other models. As in SBML, individual variables and equations can be overwritten or deleted during import. It is *hybrid* in the sense that it allows discrete events, but it only supports explicit ODE and not DAEs and it lacks support for declaring discrete variables. Like SBML, Antimony focuses on models on the level of biochemical pathways by providing a special syntax for reactions. It has no support for embedding any form of *graphical* model representations.

MoDROGH languages: Modelica

Modelica is an open-source declarative modeling language primarily used in engineering[37]. It is very similar to MATLAB's Simulink environment and the Simscape language. In fact, Simulink was developed before Modelica and Modelica before Simscape, which suggests some influence between the languages in both directions. Modelica supports *modularity* via object orientation including the overwriting of variables and explicit equations, and, in contrast to Simscape, multiple inheritance. Most Modelica tools support the FMI, allowing the reuse of models across different languages. Like Simscape, it also allows grouping of interface variables to connectors, which can be used to connect models *graphically* via drag and drop. It is *human-readable* and *declarative* allowing to define a model with a mix of explicit and implicit DAEs. Models can be annotated with documentation strings for individual components, a full HTML documentation for classes and machine-readable annotations, which do not support ontologies by default but have a flexible extension mechanism with so-called vendor-specific annotations. As in Simscape, "is-a" relationships between model components and ontological terms can also be implemented via a type hierarchy. While this introduces design restrictions in Simscape, Modelica supports multiple inheritance and therefore allows maintaining ontological type hierarchies in addition to generic base classes. Units are supported and optional consistency checks can be performed. Modelica is also fully *hybrid* with support for discrete events and variables as well as arbitrary DAEs. With additional, open-source

Table 1. Evaluation of language candidates with respect to the desirable MoDROGH characteristics established in this paper.

	MATLAB[a]	SBML	CellML	pySB	PySCeS	SimuPy	PyDSTool	Antimony	Modelica	Julia[b]
Modular	✓	(✓)	✓	(✓)	×	(✓)	×	✓	✓	(✓)
Declarative	✓	✓	✓	(✓)	(✓)	(✓)	(✓)	✓	✓	(✓)
Readable	×	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	✓	(✓)
Open	×	✓	✓	✓	✓	✓	✓	✓	(✓)	✓
Graphical	✓	✓	(✓)	×	×	×	×	×	✓	×
Hybrid	✓	(✓)	(✓)	×	(✓)	(✓)	✓	(✓)	✓	✓

A check mark in parentheses means the language has the respective characteristic in principle, but not to its full extent or with noticeable drawbacks. A more detailed version of this table with regard to individual language features can be found in Supplementary Table 1.
[a]Using the Simulink environment and the Simscape language.
[b]Using macro packages that extend the language.

libraries, it is also capable to express models, e.g., as bond graphs, Petri nets and finite state machines. The Modelica ecosystem is a mix of *open* academic tools and commercial tools for use in industry while most libraries are open-source. There are two actively maintained open-source Modelica compilers called JModelica and OpenModelica, the latter including a fully-fledged integrated development environment (IDE)[72,73]. However, Dymola, the most widely-used Modelica IDE, is proprietary and not fully compatible with open-source alternatives (https://www.3ds.com/products-services/catia/products/dymola/). Therefore, Dymola models may need to be adjusted slightly to run with open-source compilers or vice versa.

MoDROGH languages: Julia

Julia is an *open* programming language that is mainly used for data science[36]. It is imperative by nature, but the language can be extended with macros, which are more powerful than the respective capabilities of Python, to allow *declarative* modeling. Elmqvist et al. use this feature for Modia, an implementation of the Modelica syntax within Julia[74]. Modia is currently still experimental. It is *modular*, *hybrid*, and focuses on *human-readability*, but lacks, for example, the *graphical* features of Modelica.

In general, Julia offers strong support for differential equations with packages such as DifferentialEquations.jl which supports *hybrid* systems including DAEs, partial differential equations (PDEs) and discrete events, but not the declaration of discrete variables[75]. The syntax of this package focuses on *human-readability* and is *declarative*, but only allows either a fully-implicit or semi-explicit formulation of the whole system of DAEs with a mass matrix. Like Python-based solutions, annotation and *modularization* in this package is up to the modeler using the features of the language Julia which supports a multiple dispatch mechanism, which can be used to accomplish the same functionality as object orientation except for encapsulation. However, units with automated consistency checks can be supported though the Unitful.jl package[75]. Like Modia, DifferentialEquations.jl offers no *graphical* representation of models.

MoDROGH languages: comparison of existing candidates

A summary of the available languages and their features can be found in Table 1. Notably, Simscape and Modelica stand out by supporting full object-oriented design of models, explicit declaration of discrete variables, an integrated graphical representation, which allows biological drawings and manual arrangement, acausal connections between components, cross-language import of models via the FMI, grouping of interface variables as connectors, and unrestricted mixing of implicit and explicit equation formats. The feature-richness of these languages is not surprising, since both are established industry standards, which are used in multiple disciplines to build large and complex models. Between the two candidates, Modelica additionally provides a mostly open environment, multiple inheritance, overwriting of variables and some equations during instantiation and inheritance, export to other languages via the FMI, and machine-readable annotations, which can, in theory, be used to implement support for ontologies. On the downside, this ontology support must be implemented manually and is not included in major tools and while open source tools do exist, they only make up a part of the Modelica ecosystem and are not necessarily fully compatible with proprietary solutions.

Although it is certainly not the only option and it is as of now foreign to the systems biology ecosystem, we think that Modelica is a suitable choice to demonstrate the benefits of the MoDROGH characteristics since it implements them to the fullest extent among our selection of languages.

Modularizing a model of the human cardiac conduction system facilitates reuse

The Seidel-Herzel model (SHM) describes the autonomic control of the heart rate in humans at a high level of abstraction[10]. It was developed and implemented by Henrik Seidel in 1997 using the programming language C. We chose this model because preliminary versions, which have been published as individual peer-reviewed articles[11,76], have gained substantial research interest and are able to simulate several relevant disease conditions such as first and second degree atrioventricular block[10], carotid sinus hypersensitivity[11], congestive heart failure[77], and primary autonomic failure[77] as well as treatment options such as the administration of atropine or metoprolol[77]. It is also especially interesting with regard to its dynamical properties such as the emergence of Mayer waves[10], bifurcations[11], and cardio-espiratory synchronization[78]. In a previous paper, we translated the SHM to Modelica[12], and we recently also published our full model code as an open-source reference implementation[79]. The model is therefore freely available, able to produce physiologically relevant results, large enough to benefit from engineering methodology, and yet small enough to allow an in-depth analysis at the source code level. It is not representative of lower-level metabolism and cell signaling models, which are currently the most common type of models encountered in systems biology, but it is well suited to showcase what is needed for future multi-scale models, which inevitably have to leave these well-explored levels behind to generate new insights. In fact, the model can already be considered to span multiple scales of time since it includes effects at the sub-second level as well as on the level of multiple minutes[80].

To be more specific, the SHM can be classified as a hybrid (discrete and continuous), deterministic, quantitative, macro-level model. All effects in the model are described on the organ level, including the time course of systemic arterial blood pressure

generated by the pumping of the heart; the Windkessel effect of the expanding arteries dampening the initial rise in blood pressure; the arterial baroreceptors generating a neural signal depending on the absolute value and the increase in blood pressure; the autonomic nervous system emitting norepinephrine and acetylcholine as hormone and neurotransmitter based on signals from the baroreceptor and the lungs; and the cardiac conduction system with the sinoatrial node (SA node) as main pacemaker and the atrioventricular node (AV node) as a fallback system.

In the following, only the conduction system is examined. It takes an input signal from the SA node (based on norepinephrine and acetylcholine concentrations) and includes the refractory behavior of the SA node limiting the maximum signal frequency, the delay between a signal from the SA node and the actual ventricular contraction, and the AV node generating a signal if no signal has been received for a given period of time. There is a little confusion about the refractory behavior, because the wording in Seidel's thesis suggests that he wanted to model the refractory period of the ventricles, but in the code, the refractory state is checked *before* the delay between SA node and ventricles is applied. In the original model, these effects were tightly coupled within a single piece of code comprising five parameters, and 13 variables and equations—not counting additional parameters and variables for initial conditions. We found that this complexity makes it hard to understand and modify the model, which is why we translated it into a modular structure using Modelica. We will explain Modelica-specific language constructs as they appear in the code examples, but for a more complete introduction to Modelica in a biological context the reader is referred to the Lotka-Volterra examples in ref. [81] as well as our own implementation of the Hodgkin-Huxley model[82].

The modular version separates the code into the three components `RefractoryGate`, `Pacemaker` and `AVConductionDelay`. These components are connected via a unifying interface using a base class `UnidirectionalConductionComponent`, which takes a Boolean signal as an input and produces a Boolean output. These inputs and outputs are only `true` for the exact point in time when a signal is issued (i.e., they behave as a sum of Kronecker deltas). In Modelica, this behavior can be indicated by defining a type alias `InstantSignal`.

```
type InstantSignal = Boolean(quantity="sum of Kronecker deltas");
connector InstantInput = input InstantSignal annotation(...);
connector InstantOutput = output InstantSignal annotation(...);
```

The new type is functionally identical to the base type `Boolean`, but by overwriting the built-in variable `quantity` it includes additional information that is both human-readable and can be interpreted by graphical tools to enhance understandability. The next two lines achieve two separate goals: first, the keyword `connector` designates `InstantInput` and `InstantOutput` as part of the interface of a class to the outside world. Second, specifying the `input` and `output` causalities ensures that input signals can only be connected to output signals and vice versa. This distinction can also be reflected in the graphical representation, which is defined in `annotation()` statements, which are shown here without their content for the sake of brevity. The base model `UnidirectionalConductionComponent`, which has one input and one output, then becomes

```
partial model UnidirectionalConductionComponent
  InstantInput inp "input connector" annotation(Placement(...));
  InstantOutput outp "output connector" annotation(Placement(...));
  annotation(Icon(...));
end UnidirectionalConductionComponent;
```

Note that the model is declared as `partial` which indicates that it is only a template that cannot be used on its own but must be extended by defining other models that include the following declaration.

```
extends UnidirectionalConductionComponent;
```

This statement imports all variables and equations of the base class into the current model, which ensures that all components will have an input and output connector named `inp` and `outp` without the need to define these variables multiple times. Models can also inherit graphical annotations from base classes, which can define a common look and connector placement for the graphical representation.

The three main components `RefractoryGate`, `Pacemaker`, and `AVConductionDelay` all extend `UnidirectionalConductionComponent`. For the sake of brevity, we will only show the code for the `RefractoryGate` here while the code for the other two components can be found in the methods section. The `RefractoryGate` represents the refractory behavior of the SA node which cannot be excited for a certain time period after it has fired a signal. For our model this means that the output equals the input except that after each signal there is a time period `d_refrac` in which incoming signals are ignored. This results in the following definition:

```
model RefractoryGate "lets signal pass if refractory period has passed"
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Gate;
  import SI = Modelica.SIunits;
  parameter SI.Time t_first = 0 "time of first signal";
  parameter SI.Duration d_refrac = 1 "duration of refractory period";
  Boolean refrac_passed = time - pre(t_last) > d_refrac "not refractory?";
protected
  discrete SI.Time t_last(start=t_first, fixed=true) "time of last output";
equation
  outp = inp and refrac_passed;
  when outp then
    t_last = time;
  end when;
end RefractoryGate;
```

This model showcases several language features: It designates parameters with the `parameter` keyword, indicating that their value will not change during the simulation. It uses the `Modelica.SIunits` package, which contains types with unit definitions according to the SI. It documents each variable with a short informative explanation. It defines the helper variable `t_last` in a `protected` environment, which indicates that this variable is only relevant inside this component and should be hidden from other components. It contains an event using the `when` keyword, which can be used to assign values to discrete variables and to reinitialize continuous variables. It uses the `pre()` function to distinguish between the value of `t_last` *before* and *after* the event, which is required, because equations do not assume any causality. It explicitly marks `t_last` as `discrete`, which ensures that it must be defined within a `when` equation and indicates to the reader that it remains constant between events. It also employs multiple inheritance by including two `extends` statements: one for the base class containing the interface connectors, and one for an icon class containing the graphical annotation code. The latter is not strictly required, but it is convenient for readability, because it allows keeping verbose icon annotations in a separate file.

The other two components follow a similar design structure. The `Pacemaker` represents the capability of the AV node to generate spontaneous action potentials in the absence of a signal from the SA node. This means it lets incoming signals pass through but also issues a signal on its own when the output has been silent for the duration of its period `period`. To ensure that signals during the refractory period do not prematurely reset the pacemaker, it needs

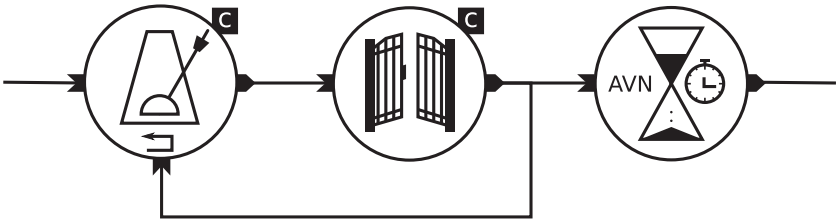


Fig. 5 Diagram of the modular conduction model with symbols for the components. Components from left to right: `Pacemaker` for the pacemaker effect of the AV node, `RefractoryGate` for the refractory behavior of the SA node and `AVConductionDelay` for the combined delay between the SA node and the ventricles. The C in a black box indicates that the main variable of the component is held constant while the stopwatch symbol for the delay should indicate that the duration is time-dependent. Components have their input on the left, their output on the right and the pacemaker has the additional reset input at the bottom.

a separate reset input, which is only triggered when the output signal has also passed the refractory gate. The `AVConduction-Delay` represents the time delay that occurs due to the slow conduction between AV node cells. It delays an incoming signal by a duration that depends on the elapsed time since the last output signal has been issued. As mentioned above, the code for both of these components can be found in the methods section.

To form the full model of the cardiac conduction system, the components have to be connected through their interface variables. In Modelica, this is usually done in graphical tools like OpenModelica through a drag and drop interface. For this, the aforementioned `annotation()` statements come into play. They define the icons and the placement of components and connection lines in a vector graphics format. An example for the placement of the `inp` connector may look as follows:

```
InstantInput inp "input connector" annotation(
  Placement(
    visible = true,
    iconTransformation(
      origin = {-108, 0}, extent = {{-10, -10}, {10, 10}}, rotation = 0
    )
  )
);
```

This ensures that the resulting diagram in Fig. 5 is not a separate image file that has to be maintained separately, but is instead directly tied to the actual model structure. To keep the model code simple and short we defined the icon annotations in separate classes whose code can be found in Supplementary Listing 23–27. As seen in Fig. 5 we chose an open fence gate for the refractory gate, a metronome for the pacemaker, and an hourglass for the delay. The components are simply connected in order with the exception that the reset of the pacemaker component is only triggered if the signal also passed the refractory component. The resulting composite Modelica model `ModularConduction` looks as follows:

```
model ModularConduction
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Heart;
  import SI = Modelica.SIunits;
  RefractoryGate refrac_av(d_refrac=0.364) "refr. time of AVN" annotation(...);
  Pacemaker pace_av(period=1.7) "AV node pacemaker behavior" annotation(...);
  AVConductionDelay delay_sa_v "delay from SAN to ventricles" annotation(...);
  discrete SI.Duration d_interbeat(start=initial_T, fixed=true);
  discrete SI.Time cont_last(start=0, fixed=true);
equation
  connect(inp, pace_av.inp) annotation(...);
  connect(pace_av.outp, refrac_av.inp) annotation(...);
  connect(refrac_av.outp, pace_av.reset) annotation(...);
  connect(refrac_av.outp, delay_sa_v.inp) annotation(...);
  connect(delay_sa_v.outp, outp) annotation(...);
  when outp then
    d_interbeat = time - pre(cont_last);
    cont_last = time;
  end when;
end ModularConduction;
```

Note that we do not show the content of the `annotation()` statements here for the sake of brevity. The full code can be found on GitHub and in Supplementary Listing 1–27. Since the model itself receives a Boolean input from the SA node and provides a Boolean output for the Ventricles, it is itself a `UnidirectionalConductionComponent`. Components are used by defining variables of the types `RefractoryGate`, `Pacemaker`, and `AVConductionDelay`. The definitions also overwrite the parameters `d_refrac` and `period` to adjust the general `Pacemaker` and `RefractoryGate` models to their specific use case in this model. The inputs and outputs of the components are connected via `connect()` equations. In this case, `connect(a, b)` is synonymous with the equation `a = b`, but more complex connectors can connect multiple variables within a single statement and can also handle conservation laws. The model also introduces the additional variable `d_inter-beat`, which allows using the interbeat intervals as a higher-level feature.

The structure defined in this model (and seen in Fig. 5) deviates from the original SHM because the refractory behavior is situated at the AV node instead of the SA node. Additionally, the delay component models the complete delay from the SA node to the ventricles but is actually applied *after* the components for the AV node. To remain closer to physiology, one could split the delay component into two delays—one before and one after the AV node—and similarly add another refractory gate for the SA node. However, in Supplementary Note 1 we show that this simplified structure closely replicates the behavior of the SHM and even reveals some minor inconsistencies in the original model.

We also used our modular version to implement the trigger for PVC, which initially uncovered the problems with the monolithic version. It turned out that this extension now becomes possible without much effort, since it is easier to determine the effect of a PVC on the individual components one by one than to describe its effect on the whole system at once. A complete discussion of the extension can be found in Supplementary Note 2 and a diagram of the resulting model can be seen in Fig. 6.

DISCUSSION

The model that we chose to demonstrate the benefit of the MoDROGH characteristics is quite small as compared to, e.g., current whole-cell models, which can involve 28 or more individual interconnected components[83]. It can be argued that one needs to look at models of this scale to really assess the impact of model engineering decisions and language choice. However, we think that more than the size or structure of the model, the context of its reuse is the most important factor that allows us to generalize our findings to different areas of mathematical modeling. To extend the SHM, we needed to identify the correct integration points for the new effect in the model, which in turn required us to first separate the model into

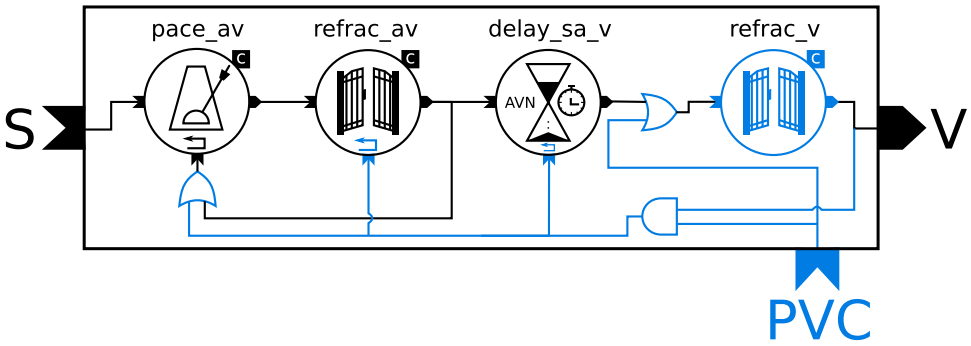


Fig. 6 Diagram of the extension of the modular conduction model with a trigger for (PVCs). The components are the same as in Fig. 5 with additional components and connections highlighted in blue: reset inputs, second `RefractoryGate` (right) for the refractory period of the ventricles, two logical OR gates and one AND gate. The letters on the outside of the rectangle represent the connections of the model to the outside world: the input from the SA node (S), the output to the ventricles (V) and the trigger signal for PVCs (PVC).

modules that each represent only a single physiological effect. We think that this requirement to understand and break up existing models for reuse in a different context represents one of the main challenges of multi-scale modeling in general. Additionally, larger models would not allow an in-depth discussion of their code within a single research article, since there would be simply too much interrelated code to discuss. We, therefore, chose the cardiac conduction system of the SHM as a "minimal working example", which is just large enough to show the effects that we want to discuss but still small enough to cover the whole code in this article. This is in accordance with common practices in computer science textbooks where general design patters for the construction of large software systems are discussed based on small examples[84]. It is also important to note that the language Modelica and the techniques that we discuss here can be, and to some extent have been, applied to build large models. Examples include the Physiolibrary, a library to build multi-organ or whole-body circulatory models[38], the Guyton model of physiological regulation[85] or even larger examples from industrial settings such as end-to-end simulation of launch vehicles[86] or electrical power systems with thousands of components[87]. For our specific example, an application to a model of relevant size is also tangible, because our model of the cardiac conduction system can be seamlessly integrated in our Modelica implementation of the full SHM, which features 15 interconnected components and also utilizes the MoDROGH characteristics[12].

This leads to our initial research question to assess how the modular, declarative, readable, open, graphical, and hybrid nature of a MoDROGH language helped in the modeling process of the conduction model. We discuss this for each individual characteristic (highlighted in **bold**) and then sum up the impact on the model design goals of reproducibility, understandability, reusabilty, and extensibility and reflect on our choice to use the modeling language Modelica.

Regarding the **modular** characteristic, the first noteworthy observation is a reduction in the amount of items that a researcher has to process simultaneously to understand the model. The modular implementation of the cardiac conduction system consists of small components with at most three parameters and seven variables and equations including only two to three interface variables. This stands in contrast to the five parameters and 13 variables and equations of the monolithic version. This can be seen as an indicator for increased understandability[88]. The `Pacemaker`, `RefractoryGate` and `ConductionDelay` models all are quite generic and it is easy to imagine that they could be reused in a different model that requires these effects. This also facilitated the extension of the model with a trigger for PVCs, which required the incorporation of a second `RefractoryGate` to model not only the refractory behavior of the SA node but also of the ventricles (see Fig. 6). In this case, the component could be reused without modification.

This extension was our initial goal, which sparked the discussion about language characteristics and design guidelines for mathematical models. When we originally tried to implement this behavior in the monolithic version, we found it extremely hard to pinpoint the lines of code that would need to change. Now, with the modular version, the question was not "Which variables do I have to change?" but "Which influence does a PVC have on physiological component X?". The discussion shifted from technological considerations to physiological ones, which made the extension possible without much effort. In the original model, several variables and equations would have to be added, making the already complicated system almost unmanageable. The benefits of modularity become even more apparent when moving from the conduction model to the whole SHM.

Since the whole `ModularConduction` model is also encapsulated with a simple interface consisting of an input, an output, the interbeat interval and the timestamp of the last contraction it can seamlessly be integrated into our modular version of the SHM. In fact, switching between the monolithic and modular implementation becomes as simple as changing the type of a variable from `MonolithicConduction` to `ModularConduction`. This is in stark contrast to the original implementation by Seidel in C, where the variables and equations for the conduction system were scattered throughout the code of the whole model. As a welcome side effect, the separation of the model into individual physiological effects also revealed some design flaws in the original. For example, it was not quite clear if the refractory term referred to the refractory behavior of the SA node, the AV node, or the ventricles and in Supplementary Note 1, we discovered that the C implementation introduces a seemingly unphysiological time-dependence in the effective duration of the refractory period.

It has to be said that the components we developed are only reusable within their physiological context. In contrast to, e.g., reaction equations in metabolism models, physiology is not yet standardized enough to have a unifying theory that allows building libraries of components that can be used in multiple tissue or organ models. However, the Physiolibrary can be seen as a first approach in this direction, which also uses Modelica[38].

Regarding the **human-readable** characteristic, all variables, parameters, and components in our model have human-readable labels that clearly specify which physiological quantity they represent. The full version of the model code, which can be found in Supplementary Listing 1–27 and at https://github.com/CSchoel/shm-conduction, also contains additional documentation.

This is important for the understandability of the model but also for reuse and extension. Reuse requires the identification of possible connection points between variables in different models based on their semantics. Extension could, for example, involve the replacement of one variable or component with a more complex representation, which models the same concept in more detail.

The new model also removed an undocumented technical workaround from the original where the interaction between refractory time, spontaneous beats by the AV node, and the time delay were resolved indirectly: a scheduling system kept track of the next time a beat would be issued, giving precedence to beats that enter the schedule at a later time but would take effect earlier. A diagram of this system can be seen in Supplementary Fig. 1. This indirect implementation was hard to understand, because the schedules have no direct physiological equivalent. The AV node, for example, does not signal the sinus node ahead of time to indicate when it will issue the next beat. This system was therefore replaced by an explicit, more readable version by only considering actual signals and no schedules.

Modelica focuses on human-readability over machine-readability, which enabled us to discuss code-level details in this article and makes it possible to quickly review changes in a version control tool like Git (https://git-scm.com/). This made it easy for us to spot bugs by tagging working versus broken versions and identifying the lines that changed between the working and the broken state. In contrast, editing software for XML-based formats with less focus on human-readability may reorder lines from one version to another without consequence for the overall functioning of the model and without these changes being apparent to the modeler. This can result in long line change messages in popular version control systems such as Git, which complicates locating the individual line that introduced an error.

Another advantage of fully human-readable code can be seen in the experiment setup, which we show in the methods section. This model contains the code `__OpenModelica_simulationFlags (s = "dassl")`, which tells the OpenModelica compiler to use the Differential Algebraic System SoLver (DASSL)[89] to solve the equation system. This information can only be interpreted by OpenModelica and not by other tools, which might not support this solver. However, since Modelica is designed to be written by humans directly, researchers who inspect the model can easily find this information without knowing that it is there. In contrast, if the model was written in an XML-based language, researchers would probably not look at the raw code, but load the model in a tool that uses an intermediary language or a graphical user interface to display the model content. It is likely that such a tool would just discard information that it cannot process, making it possible that details like these tool-dependent solver settings might be overlooked in reproduction attempts.

However, our model also has a downside with regard to its human-readability: the visual annotations, which are only helpful within a tool like OpenModelica, do introduce visual clutter when the model is only viewed in text form. To some extent, we could alleviate this issue by separating icon definitions into separate files and base classes and including them via multiple inheritance. Yet still the model `ModularConduction` has to include verbose `annotation()` statements for the placement and connection of components.

Moving on to the **hybrid** characteristic, the example model is purely discrete and therefore not hybrid in itself. However, it is important to note that we could use the same language to describe this discrete model that we also used for the rest of the SHM, which is mainly continuous. It was, for example, not needed to explicitly set the derivatives of the discrete variables to zero, which would introduce visual clutter and therefore reduce understandability. While our example cannot directly show the benefits of DAEs and acausality, these features *are* included in the implementation of the SHM that we published previously. One example of this are the acetylcholine kinetics, which use the following connector interface.

```
connector SubstanceConcentration
  Real concentration "concentration of the substance";
  flow Real rate "rate of concentration change";
end SubstanceConcentration;
```

The keyword `flow` indicates that the variable `rate` is subject to a conservation law: At each connection point in the system, the sum of acetylcholine flow from and to all connected components must be zero. In the SHM, the acetylcholine concentration is only determined by a single `NeurotransmitterRelease` component, which is connected to the parasympathetic system, but it is easy to imagine an extension that includes multiple uptake sites. In such an example, the automatic generation of the conservation law by Modelica would allow to separate the effects of all connected components, which only have to declare their individual contribution to the acetylcholine concentration. This both leads to better encapsulation, making the model more understandable, and it facilitates extension since additional components that have an influence on the concentration can be added without changing any of the equations of the existing components.

The **openness** of both the Modelica language and the OpenModelica IDE ensures that readers can easily run simulations themselves. They simply have to download the latest release from the Github repository at https://github.com/CSchoel/shm-conduction, download and install OpenModelica and load the models using the "Load library" option in the “File” menu. This means that our results can be easily reproduced regardless of available licenses or of the user’s operating system and that researchers who might want to reuse the model or components can quickly run simulations to assess the usefulness of the model for their use case.

However, there may still be some barriers. First, Modelica is not yet widely known in systems biology, which makes it likely that researchers will have to become familiar with a new tool and language in order to reproduce our findings. Second, engineers that use Modelica for industrial applications mostly turn toward proprietary solutions like Dymola (https://www.3ds.com/products-services/catia/products/dymola/), which can be more feature-rich than and not fully compatible with OpenModelica.

One area where the **declarative** characteristic comes into play in our example are unit definitions. Even though the example model only contains time-related variables and no other physical quantities, the fact that we introduced proper SI units is still helpful for understanding. For example, it avoids errors, if the components are reused in another model that measures time in milliseconds instead of seconds. Unfortunately, Modelica does not enforce unit definitions or unit consistency checks. However, when these optional unit declarations are used consistently, they help to quickly identify and solve such order of magnitude errors.

In his C implementation, Seidel implemented a fourth order Runge-Kutta method himself, making this the only numerical method available. By using a declarative language, it becomes p-

ossible to easily switch between different solvers which can improve numerical accuracy. Not being tied to a specific numerical method also increases interoperability between models and thus reusability.

Another benefit of the declarative specification that becomes apparent in our example is the increase in mathematical soundness and clarity. The C implementation contained some design choices that were convenient for programming, but neither for understanding nor physiological plausibility. For example, the original model mixed variables that represent actual signals and time stamp variables that schedule signals for the future (see Supplementary Fig. 1). To comprehend these formulas a context switch from the physiological meaning to the technical representation is required. Another hurdle for understanding the model is the unclear causality. In the SHM, every effect is triggered by the contraction, even if there is no actual signal feedback from the ventricles to the AV node on a physiological level. By separating the model into smaller physiologically meaningful modules with a unified interface, the Modelica compiler automatically hinted at these concerns, e.g., because variables were missing.

One downside of choosing Modelica is that we could not demonstrate the benefit of augmenting a model with semantic information using ontologies. However, the use of the `SIunits` package and the definition of the `InstantSignal` type to indicate Kronecker delta behavior of in- and outputs show how this could be achieved through a type hierarchy: The variables in the model do not only have units, but we also distinguish between the type `Time` for a point in time and `Duration` for a difference between two points in time. Similarly, `InstantSignal` is technically equivalent to the type `Boolean`, but carries additional semantic information about the shape of the signal. In much the same way, one could build large type hierarchies containing all terms of an ontology like the SBO, Chemical Entities of Biological Interest (ChEBI)[90] or Ontology of Physics for Biology (OPB)[91]. Another way to implement ontology support in Modelica would be so called vendor-specific annotations of the form `annotation(__VendorName(key=value, ...))`, which could be added to components and variables. Since ontology terms are typically identified through Uniform Resource Identifiers (URIs), which are not human-readable, and because there are currently no graphical Modelica tools that support such ontologies, the first approach using the type system seems preferable for now.

Regarding the **graphical** characteristic, the diagram in Fig. 5 helps to understand the model at first glance. It can both be used as an entry point for understanding and for communicating the model to a domain expert who is not familiar with mathematical modeling or the language Modelica. The same would not be possible with more detailed SBGN graphs or automatically generated graphs of variable interactions, which would also include internal variables of components and helper variables. At the same time, the diagram is not just a separate image file but it is generated from `annotation()` statements in the individual components themselves. This means that it will remain up to date if components are added or removed or new connectors are included so that other researchers can rely on the accuracy of the diagram if they want to understand, reuse, or reproduce the model. The annotations also allow building more complex models or small test cases using drag and drop in a graphical tool like OpenModelica[72], which can facilitate reuse and extension. For example, the PVC extension required very little changes in the code. Most of the changes could be applied by adding a `RefractoryGate` component and three logical AND and OR gates to the diagram which can be seen in Fig. 6.

On the downside, it can be argued that the connection in Fig. 5 which points back from the refractory component to the pacemaker component is unintuitive and may be confusing when the model is interpreted physiologically. This can be remedied by introducing another layer of abstraction, which combines the components `Pacemaker` and `RefractoryGate` to a single component `RefractoryPacemaker`. We did not do this in our implementation to keep the model code as simple as possible, but in a larger model such an intermediary component may be advisable.

As of now, our discussion was focused on Modelica, but there are multiple languages with MoDROGH characteristics. Additionally, our example revealed some shortcomings of Modelica with respect to modeling biological systems. We, therefore, want to recapitulate which features of the language were especially beneficial for our model design, which features were lacking, and what are the trade-offs that have to be made when switching to another language.

Our Modelica implementation made heavy use of object orientation, including multiple inheritance; it featured discrete variables with human-readable labels; it relied on the graphical representation for creating and communicating the toplevel model structure; it provided minimal interfaces by encapsulating helper variables and defining explicit connectors; and it used the built-in support for SI units. Unfortunately, unit definition are not enforced in Modelica, which means that it is up to the modeler to ensure their reliable use. Modelica also does not support semantic annotation of model components with ontological terms. We showed how this can be achieved with a type hierarchy or with vendor-specific annotations, but still ontology support would have to be added to open-source Modelica tools to be of practical use.

As an alternative, MATLAB with the Simulink environment and the Simscape language is the only other language presented in the results section that supports full object orientation, declaration of discrete variables, and integrated graphical annotations. In contrast to Modelica, it also enforces unit checks at the interfaces between components. The only technical downside of this language is that Simscape classes do not support multiple inheritance. However, this is probably no issue since we only used multiple inheritance to import the annotation code for icons. Simulink only supports icons as links to separate image files and not as verbose vector graphics code. This has the benefit of not cluttering the code and, therefore, removing the requirement for multiple inheritance, but it also has the drawback that it is not possible to define a common appearance for all components by inheriting parts of the graphical annotation from the base class. Apart from the technical aspects, the biggest drawback of MATLAB is that it is not open. If we had used MATLAB instead of Modelica, researchers who want to repeat our experiments could still download the code from GitHub, but they would need licenses for Matlab, Simulink, and Simscape to run the simulations.

From the open alternatives, Julia with the Modia package comes closest to the features we used for our example model. It has the drawback of not supporting any graphical representation and still being in an experimental stage. If one instead makes the switch to the more stable DifferentialEquations.jl, the support for object-orientation, declaration of discrete variables, variable labels, and encapsulation is lost.

To compensate the shortcomings of Modelica, CellML might be the best fit, since it is the only language from those presented in the results section that enforces unit definitions. Additionally, it also supports model annotation with terms from an ontology and as an accepted COMBINE standard it is part of an established

ecosystem for mathematical modeling in systems biology. With SemGen, there even exists a tool for semantics-based annotation and composition of CellML (and SBML) models[92]. On the downside, CellML does not support full object orientation for composing models. This means that base classes that define an interface need to be imported as *components* of the model, which requires a more verbose syntax. It also does not support the explicit declaration of discrete variables but does support variable reinitialization due to discrete events. To implement a variable like the interbeat interval `d_interbeat`, which stays constant between events, an additional equation would have to be added to the model, which sets the derivative of this variable to zero. This both introduces unnecessary code and makes the model less understandable as there is no clear distinction between discrete and continuous parts apart from the labels assigned to the variables. CellML also follows a different philosophy for graphical annotations, providing basic support for referencing and captioning externally-stored images as part of the documentation for the whole model. Finally, the language is not designed to be written directly by humans and instead relies on the use of appropriate tools to view and edit models.

It becomes clear that there is no single "best" language. Further development is needed to obtain a language that supports all the MoDROGH characteristics to their fullest extent. This development could start with Modelica, leveraging a flexible and industry-proven general-purpose modeling language and extending it to fit the needs of the systems biology community. It could also start with CellML or SBML, which are already proven languages with widespread support in the systems biology community, and which could borrow some software engineering features and standards from languages like Modelica or Simscape. Other approaches and foundations are also possible and it may even make sense to not pursue the "perfect" language at all, but to focus more on interoperability between languages that fit the specialized needs of smaller modeling domains.

In conclusion, using a modeling language that is Modular, Descriptive, human-Readable, Open, Graphical, and Hybrid (MoDROGH) can make models more reproducible, understandable, reusable, and extensible. Because there is no single best language, modelers have to decide which features are most important for them and which trade-offs they are willing to make. They should be aware of the beneficial characteristics of the language and use them consistently as we described in our guidelines and showed in our modular example model of the cardiac conduction system. The situation that a model needs to be dissected, modified, and extended to be used in a different context is common in multi-scale, multi-level, and multi-class models and therefore it is likely that our findings translate to large areas of systems biology. Mathematical modeling in systems biology has become an engineering challenge that requires engineering solutions. Models should no longer be implemented with only a single purpose in mind, but as reliable parts of larger systems. We hope that this article can spark a discussion in the community to put more emphasis on these engineering aspects of mathematical modeling in the development, selection, and application of modeling languages.

METHODS

Material

We used Mo|E version 0.6.3[93] to write the code of our models and OpenModelica version 1.13.0[72] as well as Inkscape version 0.91 (https://inkscape.org/) to add the component icons. OpenModelica version 1.17.0-dev.344+gc8233fa62a was used for all simulations in conjunction with Julia version 1.5.3 and the packages OMJulia (v0.1.0) and ModelicaScriptingTools (v1.1.0-alpha.4). For plotting we used Python version 3.9.1 with the packages matplotlib (v3.3.4), numpy (v1.20.0), and pandas (v1.2.1).

In the following we will show and explain our Modelica code for the models and simulations. To keep it short, we do not show the code of the original monolithic version and of the graphical annotations. We also do not include most of the documentation strings, which are present in the full version. They can be found in Supplementary Listing 1–27 and at https://github.com/CSchoel/shm-conduction. Please also note that this article was previously published as a preprint[94].

Modular conduction model

The fundamental part of the modular model of the human cardiac conduction system is the interface component `UnidirectionalConductionComponent`, which serves as a base class for all other components. It has already been shown in the results section. It defines the input and output connectors `inp` and `outp`, which are Booleans that are wrapped in a custom type `InstantSignal` to indicate that they behave as a sum of Kronecker deltas, meaning that they are only true for the exact instants in time when events occur:

```
type InstantSignal = Boolean(quantity="sum of Kronecker deltas");
connector InstantInput = input InstantSignal annotation(...);
connector InstantOutput = output InstantSignal annotation(...);

partial model UnidirectionalConductionComponent
  InstantInput inp "input connector" annotation(Placement(...));
  InstantOutput outp "output connector" annotation(Placement(...));
  annotation(Icon(...));
end UnidirectionalConductionComponent;
```

The keyword `connector` designates the types `InstantInput` and `InstantOutput` as part of the interface of a class and allows the assignment of a basic icon representation in the form of an `annotation()` statement. The content of these annotation statements can be quite verbose, which is why we only show them in the full code in Supplementary Listing 1–27 as well as on GitHub. The model `UnidirectionalConductionComponent` is defined as `partial` to designate that it is not designed to be used as a finished component but has to be extended in some way—in this case by defining the relationship between the input and the output.

The `RefractoryGate` has already been shown in the results section. The component passes on its input signal as output signal, but only when the elapsed time since the last signal left the component is larger than the refractory period:

```
model RefractoryGate "lets signal pass if refractory period has passed"
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Gate;
  import SI = Modelica.SIunits;
  parameter SI.Time t_first = 0 "time of first signal";
  parameter SI.Duration d_refrac = 1 "duration of refractory period";
  Boolean refrac_passed = time - pre(t_last) > d_refrac "not refractory?";
protected
  discrete SI.Time t_last(start=t_first, fixed=true) "time of last output";
equation
  outp = inp and refrac_passed;
  when outp then
    t_last = time;
  end when;
end RefractoryGate;
```

The function `pre()` is used here to denote the value right before an event instead of the value right after the event. The `when` statement describes an event which can define states of discrete variables and can reinitialize continuous variables. This model also introduces a `protected` section which contains variables and parameters that should not be visible from the outside.

The `Pacemaker` model propagates incoming signals, but also adds its own signal if there was no input for a certain period of time. Additionally, this component, too, has to ignore incoming signals during the refractory period. This can be implemented by decoupling the reset of the pacemaker timer from the output of the component and instead treating the reset signal as an additional input. It is assumed that this reset signal is only triggered if the signal passes not only the pacemaker but also the subsequent `RefractoryGate` component. The pacemaker component itself still resets when a spontaneous output signal is generated to maintain the invariant that the output signal will not be true for a prolonged period of time:

```
model Pacemaker "pacemaker eliciting spontaneous signals"
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Metronome;
  InstantInput reset "resets internal clock";
  import SI = Modelica.SIunits;
  parameter SI.Period period = 1 "pacemaker period";
protected
  discrete SI.Time t_next(start=period, fixed=true)
    "scheduled time of next spontaneous beat";
  InstantSignal spontaneous_signal = time > pre(t_next)
    "signal generated spontaneously by this pacemaker";
equation
  outp = inp or spontaneous_signal;
  when spontaneous_signal or pre(reset) then
    t_next = time + period;
  end when;
end Pacemaker;
```

The `ConductionDelay` model puts incoming signals on hold and releases them after a certain time has passed. Physiologically the duration of the delay for each signal depends on the time that has passed between the last signal leaving the component and the current input signal. The original model silently assumed that there will never be a second input signal while a signal is put on hold. Therefore, this assumption is kept, but made more explicit by using the helper variable `delay_passed` in the `when` condition:

```
partial model ConductionDelay "delay depending on prev. cycle duration"
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Hourglass;
  import SI = Modelica.SIunits;
  discrete SI.Duration d_delay "delay duration";
  Boolean delay_passed(start=false, fixed=true) = time > t_next
    "if false, there is still a signal currently put on hold";
protected
  discrete SI.Duration d_outp_inp(start=0, fixed=true)
    "time between last output and following signal";
  discrete SI.Time t_last(start=0, fixed=true) "time of last output";
  discrete SI.Time t_next(start=-1, fixed=true)
    "scheduled time of next output";
equation
  outp = edge(delay_passed);
  when inp and pre(delay_passed) then
    d_outp_inp = time - pre(t_last);
    t_next = time + d_delay;
  end when;
  when outp then
    t_last = time;
  end when;
end ConductionDelay;
```

Modelica does already have support for explicit delays, but this feature is tailored toward continuous variables. Therefore, we use a scheduling solution with the variable `t_next`, which indicates the time when the next signal should leave the component. This is similar to the approach in the original C implementation of the SHM, but here this scheduling system is encapsulated in a single component and the respective helper variables are defined in a `protected` environment so that they do not show up in the simulation output.

Note that this is again only a `partial` model, which does not specify the behavior of the variable `d_delay`. This allows the separation of the general delay logic from the physiological equation for the AV node which is modeled in the `AVConductionDelay`:

```
model AVConductionDelay "conduction delay between SA node and ventricles"
  extends ConductionDelay;
  import SI = Modelica.SIunits;
  parameter SI.Duration k_avc_t = 0.78 "maximum increase in delay duration";
  parameter SI.Duration d_avc0 = 0.09 "minimal delay duration";
  parameter SI.Duration tau_avc = 0.11 "reference time for delay duration";
  parameter SI.Duration initial_d_avc = 0.15 "initial value for delay";
initial equation
  d_delay = initial_d_avc;
equation
  when inp and pre(delay_passed) then
    d_delay = d_avc0 + k_avc_t * exp(-d_outp_inp/tau_avc);
  end when;
end AVConductionDelay;
```

Currently, this separation is only performed to increase readability and to not further complicate the already complex `ConductionDelay` component. Additionally, if the delay was split into two components, as discussed in the results section, the second delay component could also inherit the base equations from `ConductionDelay`, which would avoid code duplication.

Finally, the model `ModularConduction` combines the aforementioned components using `connect()` equations to connect the input and output variables. These equations are represented as lines in the graphical representation which are again defined in `annotation()` statements:

```
model ModularConduction
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Heart;
  import SI = Modelica.SIunits;
  RefractoryGate refrac_av(T_refrac=0.364)
    "refractory component for AV node" annotation(...);
  Pacemaker pace_av(period=1.7)
    "pacemaker effect of AV node" annotation(...);
  AVConductionDelay delay_sa_v
    "delay between SA node and ventricles" annotation(...);
  discrete SI.Duration d_interbeat(start=1, fixed=true)
    "duration of last heart cycle";
  discrete SI.Time cont_last(start=0, fixed=true)
    "time of last contraction";
equation
  connect(inp, pace_av.inp) annotation(...);
  connect(pace_av.outp, refrac_av.inp) annotation(...);
  connect(refrac_av.outp, pace_av.reset) annotation(...);
  connect(refrac_av.outp, delay_sa_v.inp) annotation(...);
  connect(delay_sa_v.outp, outp) annotation(...);
  when outp then
    d_interbeat = time - pre(cont_last);
    cont_last = time;
  end when;
end ModularConduction;
```

As already mentioned in the results section, this model also shows how parameters like `d_refrac` and `period` can be adjusted when the components are imported. Note also that the model `ModularConduction` is again an `UnidirectionalConductionComponent` and can therefore be used as a component in a larger model such as the SHM.

Modular contraction experiment setup

Simulation experiments can also be defined directly in Modelica syntax. The following code was used to produce Supplementary Fig. 2:

```
model ModularExample
  ModularConduction modC;
  MonolithicConduction monC;
equation
  modC.inp = monC.inp;
  if time < 5 then
    monC.inp = sample(0,1);
  elseif time < 15 then
    monC.inp = sample(0,3);
  elseif time < 20 then
    monC.inp = sample(0,0.05);
  elseif time < 30 then
    monC.inp = sample(0,0.8);
  elseif time < 40 then
    monC.inp = sample(0,0.2);
  else
    monC.inp = sample(0,1.8);
  end if;
  annotation(
    experiment(
      StartTime = 0, StopTime = 50,
      Tolerance = 1e-6, Interval = 0.002
    ),
    __OpenModelica_simulationFlags(s = "dassl")
  );
end ModularExample;
```

Here, the built-in function `sample(start, interval)` is used to issue signals from the SA node at a precise interval. The interval length is switched every five to ten seconds using an `if` statement. In addition to the experiment setup, the experiment protocol is also given by the `experiment()` annotation, which defines the start and stop times of the interval, the requested step size for the output and the tolerance used in the solver settings. The vendor-specific annotation `__OpenModelica_simulationFlags` is used to define the DASSL[89] as the default solver. For Supplementary Fig. 2, the variables `monC.d_interbeat` and `modC.d_interbeat` were plotted against simulation time.

DATA AVAILABILITY

The data for figures in the data supplement can be generated from the model code available on GitHub (https://github.com/CSchoel/shm-conduction), Zenodo[95], and BioModels (https://www.ebi.ac.uk/biomodels/MODEL2103050002). No other datasets were generated or analyzed during the current study.

CODE AVAILABILITY

The full code of the models, experiments, and plots used in this article can be found on GitHub, Zenodo, and BioModels. The model of the cardiac conduction system is uploaded as https://github.com/CSchoel/shm-conduction[95], and https://www.ebi.ac.uk/biomodels/MODEL2103050002. The identifiers for the Modelica implementation of the SHM are https://github.com/CSchoel/shm-conduction[79], and https://www.ebi.ac.uk/biomodels/MODEL2101280001, respectively.

Received: 18 October 2019; Accepted: 19 April 2021;
Published online: 03 June 2021

REFERENCES

1. Hodgkin, A. L. & Huxley, A. F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol.* **117**, 500–544 (1952).
2. Bardini, R., Politano, G., Benso, A. & Di Carlo, S. Multi-level and hybrid modelling approaches for systems biology. *Comput. Struct. Biotechnol. J.* **15**, 396–402 (2017).
3. Uhrmacher, A. M., Degenring, D. & Zeigler, B. Discrete event multi-level models for systems biology. In *Transactions on Computational Systems Biology I*, (ed Priami, C.), 66–89 (Springer, 2005).
4. Dada, J. O. & Mendes, P. Multi-scale modelling and simulation in systems biology. *Integr. Biol.* **3**, 86 (2011).
5. Yu, J. S. & Bagheri, N. Multi-class and multi-scale models of complex biological phenomena. *Curr. Opin. Biotech.* **39**, 167–173 (2016).
6. Waltemath, D. & Wolkenhauer, O. How modeling standards, software, and initiatives support reproducibility in systems biology and systems medicine. *IEEE Trans. Biomed. Eng.* **63**, 1999–2006 (2016).
7. Medley, J. K., Goldberg, A. P. & Karr, J. R. Guidelines for reproducibly building and simulating systems biology models. *IEEE Trans. Biomed. Eng.* **63**, 2015–2020 (2016).
8. Tiwari, K. et al. Reproducibility in systems biology modelling. *Mol. Syst. Biol.* **17**, e9982 (2021).
9. Topalidou, M., Leblois, A., Boraud, T. & Rougier, N. P. A long journey into reproducible computational neuroscience. *Front. Comput. Neurosci.* **9**, 1–2 (2015).
10. Seidel, H. *Nonlinear Dynamics of Physiological Rhythms*. PhD thesis, (Technische Universität Berlin, Berlin, Germany, 1997).
11. Seidel, H. & Herzel, H. Bifurcations in a nonlinear model of the baroreceptor-cardiac reflex. *Physica D: Nonlinear Phenomena* **115**, 145–160 (1998).
12. Schölzel, C., Goesmann, A., Ernst, G. & Dominik, A. Modeling biology in Modelica: The human baroreflex. In *Proceedings of the 11th International Modelica Conference*, 367–376 (Versailles, France, 2015).
13. Sarma, G. P. et al. Unit testing, model validation, and biological simulation. *F1000Research* **5**, 1946 (2016).
14. Miskovic, L., Tokic, M., Fengos, G. & Hatzimanikatis, V. Rites of passage: requirements and standards for building kinetic models of metabolic phenotypes. *Current Opin. Biotechnol.* **36**, 146–153 (2015).
15. Hicks, J. L., Uchida, T. K., Seth, A., Rajagopal, A. & Delp, S. L. Is my model good enough? Best practices for verification and validation of musculoskeletal models and simulations of movement. *J. Biomech. Eng.* **137**, 020905 (2015).
16. Grimm, V. & Railsback, S. F. Pattern-oriented modelling: a 'multi-scope' for predictive systems ecology. *Philos. Trans. R. So. B: Biol. Sci.* **367**, 298–310 (2012).
17. Zhao, P., Rowland, M. & Huang, S.-M. Best practice in the use of physiologically based pharmacokinetic modeling and simulation to address clinical pharmacology regulatory questions. *Clin. Pharmacol. Ther.* **92**, 17–20 (2012).
18. Smith, N. P., Crampin, E. J., Niederer, S. A., Bassingthwaighte, J. B. & Beard, D. A. Computational biology of cardiac myocytes: proposed standards for the physiome. *J. Exp. Biol.* **210**, 1576–1583 (2007).
19. Goldberg, A. P. et al. Emerging whole-cell modeling principles and methods. *Curr. Opin. Biotechnol.* **51**, 97–102 (2018).
20. Bartocci, E. & Lió, P. Computational modeling, formal analysis, and tools for systems biology. *PLOS Comput. Biol.* **12**, e1004591 (2016).
21. Walpole, J., Papin, J. A. & Peirce, S. M. Multiscale computational models of complex biological systems. *Ann. Rev. Biomed. Eng.* **15**, 137–154 (2013).
22. Hucka, M. et al. Promoting coordinated development of community-based information standards for modeling in biology: the COMBINE initiative. *Front. Bioeng. Biotechnol.* **3,** 19 (2015).
23. Wolstencroft, K. et al. SEEK: a systems biology data and model management platform. *BMC Syst. Biol.* **9**, 33 (2015).
24. Cooling, M. T., Hunter, P. & Crampin, E. J. Modelling biological modularity with CellML. *IET Syst. Biol.* **2**, 73–79 (2008).
25. Neal, M. L. et al. A reappraisal of how to build modular, reusable models of biological systems. *PLOS Comput. Biol.* **10**, e1003849 (2014).
26. Waltemath, D. et al. The first 10 years of the international coordination network for standards in systems and synthetic biology (COMBINE). *J. Integr. Bioinform.* **17**, 20200005 (2020).
27. Malik-Sheriff, R. S. et al. BioModels—15 years of sharing computational models in life science. *Nucleic Acids Res.* **48**, D407–D415 (2019).
28. Cooling, M. T. et al. Standard virtual biological parts: a repository of modular modeling components for synthetic biology. *Bioinformatics* **26**, 925–931 (2010).
29. Clerx, M., Collins, P., de Lange, E. & Volders, P. G. Myokit: a simple interface to cardiac cellular electrophysiology. *Prog. Biophys. Mol. Biol.* **120**, 100–114 (2016).
30. Mulugeta, L. et al. Credibility, replicability, and reproducibility in simulation for biomedicine and clinical applications in neuroscience. *Front. Neuroinform.* **12**, 18 (2018).
31. Olivier, B. G., Rohwer, J. M. & Hofmeyr, J.-H. S. Modelling cellular systems with PySCeS. *Bioinformatics* **21**, 560–561 (2005).

32. Clewley, R. Hybrid models and biological model reduction with PyDSTool. *PLoS Comput. Biol.* **8**, e1002628 (2012).
33. Lopez, C. F., Muhlich, J. L., Bachman, J. A. & Sorger, P. K. Programming biological models in Python using PySB. *Mol. Syst. Biol.* **9**, 646 (2013).
34. Choi, K. et al. Tellurium: An extensible python-based modeling environment for systems and synthetic biology. *Biosystems* **171**, 74–79 (2018).
35. Smith, L. P., Bergmann, F. T., Chandran, D. & Sauro, H. M. Antimony: a modular model definition language. *Bioinformatics* **25**, 2452–2454 (2009).
36. Bezanson, J., Edelman, A., Karpinski, S. & Shah, V. B. Julia: A fresh approach to numerical computing. *SIAM Rev.* **59**, 65–98 (2017).
37. Mattsson, S. E. & Elmqvist, H. Modelica—an International effort to design the next generation modeling language. In *7th IFAC Symposium on Computer Aided Control Systems Design, CACSD'97*, 51–155 (Gent, Belgium, 1997).
38. Mateják, M. et al. Physiolibrary—Modelica library for physiology. In *Proceedings of the 10th International Modelica Conference*, 499–505 (Lund, Sweden, 2014).
39. Maggioli, F., Mancini, T. & Tronci, E. SBML2Modelica: Integrating biochemical models within open-standard simulation ecosystems. *Bioinformatics* **36**, 2165–2172 (2019).
40. Hellerstein, J. L., Gu, S., Choi, K. & Sauro, H. M. Recent advances in biomedical simulations: A manifesto for model engineering. *F1000Research* **8**, 261 (2019).
41. Blochwitz, T. et al. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference*, 105–114 (Dresden, Germany, 2011).
42. Blochwitz, T. et al. Functional mockup interface 2.0: the standard for tool independent exchange of simulation models. In *Proceedings of the 9th International Modelica Conference*, 173–184 (Munich, Germany, 2012).
43. Zhu, X.-G. et al. Plants *in silico*: Why, why now and what?-An integrative platform for plant systems biology research. *Plant, Cell Environ.* **39**, 1049–1057 (2016).
44. Mirschel, S., Steinmetz, K., Rempel, M., Ginkel, M. & Gilles, E. D. ProMoT: Modular modeling for systems biology. *Bioinformatics* **25**, 687–689 (2009).
45. Kell, D. The virtual human: Towards a global systems biology of multiscale, distributed biochemical network models. *IUBMB Life* **59**, 689–695 (2007).
46. Hasenauer, J., Jagiella, N., Hross, S. & Theis, F. J. Data-driven modelling of biological multi-scale processes. *J Coupled Syst. Multiscale Dyn.* **3**, 101–121 (2015).
47. Oliveira, A., Kohwalter, T., Kalinowski, M., Murta, L. & Braganholo, V. XChange: a semantic diff approach for XML documents. *Information Systems* **94**, 101610 (2020).
48. Dräger, A. et al. SBML2LaTeX: conversion of SBML files into human-readable reports. *Bioinformatics* **25**, 1455–1456 (2009).
49. Lincoln, P. & Tiwari, A. Symbolic systems biology: hybrid modeling and analysis of biological networks. In *Hybrid Systems: Computation and Control* (eds Alur, R. & Pappas, G. J.), 660–672 (Springer, 2004).
50. Bortolussi, L. & Policriti, A. Hybrid systems and biology. In *Formal Methods for Computational Systems Biology* (eds Bernardo, M., Degano, P. & Zavattaro, G.) 424–448 (Springer, 2008).
51. Rejniak, K. A. & Anderson, A. R. A. Hybrid models of tumor growth. *Wiley Interdiscip. Rev. Syst. Biol. Med.* **3**, 115–125 (2011).
52. Yu, T. et al. The physiome model repository 2. *Bioinformatics* **27**, 743–744 (2011).
53. Bassingthwaighte, J. B. Strategies for the physiome project. *Anna. Biomed. Eng.* **28**, 1043–1058 (2000).
54. Holzhütter, H.-G., Drasdo, D., Preusser, T., Lippert, J. & Henney, A. M. The virtual liver: a multidisciplinary, multilevel challenge for systems biology. *Wiley Interdiscip. Rev. Syst. Biol. Med.* **4**, 221–235 (2012).
55. Zhu, H., Huang, S. & Dhar, P. The next step in systems biology: simulating the temporospatial dynamics of molecular network. *BioEssays* **26**, 68–72 (2004).
56. Loew, L. M. & Schaff, J. C. The virtual cell: a software environment for computational cell biology. *Trends in Biotechnol.* **19**, 401–406 (2001).
57. Butterworth, E., Jardine, B. E., Raymond, G. M., Neal, M. L. & Bassingthwaighte, J. B. JSim, an open-source modeling system for data analysis. *F1000Research* **2**, 288 (2013).
58. Yan, K. & Cui, W. Visualizing the uncertainty induced by graph layout algorithms. In *2017 IEEE Pacific Visualization Symposium (PacificVis)*, 200–209 (Seoul, South Korea, 2017).
59. Kerren, A. & Schreiber, F. Network visualization for integrative bioinformatics. In *Approaches in Integrative Bioinformatics* (eds Chen, M. & Hofestädt, R.), 173–202 (Springer, 2014).
60. Le Novère, N. et al. The systems biology graphical notation. *Nat. Biotechnol.* **27**, 735–741 (2009).
61. Gonçalves, E., Iersel, M. & Saez-Rodriguez, J. CySBGN: a cytoscape plug-in to integrate SBGN maps. *BMC Bioinformatics* **14**, 17 (2013).
62. Gauges, R., Rost, U., Sahle, S., Wengler, K. & Bergmann, F. T. The Systems Biology Markup Language (SBML) level 3 package: Layout, version 1 core. *J. Integr. Bioinform.* **12**, 550–602 (2015).
63. Bergmann, F. T., Keating, S. M., Gauges, R., Sahle, S. & Wengler, K. SBML level 3 package: Render, version 1, release 1. *J. Integr. Bioinform.* **15** https://doi.org/10.1515/jib-2017-0078 (2018).
64. Alves, R., Antunes, F. & Salvador, A. Tools for kinetic modeling of biochemical networks. *Nature Biotechnol.* **24**, 667–672 (2006).
65. Mangourova, V., Ringwood, J. & Van Vliet, B. Graphical simulation environments for modelling and simulation of integrative physiology. *Comput. Methods Programs Biomed.* **102**, 295–304 (2011).
66. Keating, S. M. et al. SBML Level 3: an extensible format for the exchange and reuse of biological models. *Mol. Syst. Biol.* **16**, e9110 (2020).
67. Bornstein, B. J., Keating, S. M., Jouraku, A. & Hucka, M. LibSBML: An API library for SBML. *Bioinformatics* **24**, 880–881 (2008).
68. Cuellar, A. A. et al. An overview of CellML 1.1, a biological model description language. *Simulation* **79**, 740–747 (2003).
69. Garny, A. & Hunter, P. J. OpenCOR: A modular and interoperable approach to computational biology. *Front. Physiol.* **6**, 26 (2015).
70. Nickerson, D. & Buist, M. Practical application of CellML 1.1: The integration of new mechanisms into a human ventricular myocyte model. *Prog. Biophys. Mol. Biol.* **98**, 38–51 (2008).
71. Margolis, B. W. L. SimuPy: a Python framework for modeling and simulating dynamical systems. *J. Open Source Softw.* **2**, 396 (2017).
72. Fritzson, P. et al. The OpenModelica modeling, simulation, and development environment. In *Proceedings of the 46th Conference on Simulation and Modelling of the Scandinavian Simulation Society* (Trondheim, Norway, 2005).
73. Åkesson, J. R., Gäfvert, M. & Tummescheit, H. J. Modelica—An open source platform for optimization of modelica models. In *Proceedings of the 6th Vienna International Conference on Mathematical Modelling* (Vienna, Austria, 2009).
74. Elmqvist, H., Henningsson, T. & Otter, M. Systems modeling and programming in a unified environment based on Julia. In *ISoLA 2016: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, 198–217 (Corfu, Greece, 2016).
75. Rackauckas, C. & Nie, Q. DifferentialEquations.jl—A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *Journal of Open Research Software* **5**, 15 (2017).
76. Seidel, H. & Herzel, H. Modelling heart rate variability due to respiration and baroreflex. In *Modelling the Dynamics of Biological Systems* (eds Mosekilde, E. & Mouritsen, O. G.), 205–229 (Springer, 1995).
77. Kotani, K., Struzik, Z., Takamasu, K., Stanley, H. & Yamamoto, Y. Model for complex heart rate dynamics in health and diseases. *Phys. Rev. E* **72**, 041904 (2005).
78. Kotani, K., Takamasu, K., Ashkenazy, Y., Stanley, H. & Yamamoto, Y. Model for cardiorespiratory synchronization in humans. *Phys. Rev. E* **65**, 051923 (2002).
79. Schölzel, C. Modelica implementation of the Seidel-Herzel model of the human baroreflex. Zenodo https://doi.org/10.5281/ZENODO.3855126 (2020).
80. Duggento, A., Toschi, N. & Guerrisi, M. Modeling of human baroreflex: considerations on the Seidel–Herzel model. *Fluct. Noise Lett.* **11**, 1240017 (2012).
81. Tiller, M. *Modelica by Example* https://mbe.modelica.university/. E-book (Michael Tiller, 2020).
82. Schölzel, C., Blesius, V., Ernst, G. & Dominik, A. An understandable, extensible, and reusable implementation of the Hodgkin-Huxley equations using Modelica. *Front. Physiol.* **11**, 583203 (2020).
83. Karr, J. R. et al. A whole-cell computational model predicts phenotype from genotype. *Cell* **150**, 389–401 (2012).
84. Freeman, E., Robson, E., Bates, B. & Sierra, K. *Head First Design Patterns* (O'Reilly, Sebastopol, CA, 2004).
85. Kofránek, J., Rusz, J. & Matoušek, S. Guyton's Diagram brought to life—From graphic chart to simulation model for teaching physiology. In *Technical Computing Prague 2007: 15th Annual Conference Proceedings*, 1–13 (Prague, Czech Republic, 2007).
86. Briese, L. E., Klöckner, A. & Reiner, M. The DLR environment library for multidisciplinary aerospace applications. In *Proceedings of the 12th International Modelica Conference*, 929–938 (Prague, Czech Republic, 2017).
87. Casella, F., Bartolini, A., Pasquini, S. & Bonuglia, L. Object-oriented modelling and simulation of large-scale electrical power systems using Modelica: A first feasibility study. In *Proceedings of the IECON 2016—42nd Annual Conference of the IEEE Industrial Electronics Society*, 6298–6304 (Florence, Italy, 2016).
88. Sweller, J. Cognitive Load Theory. In *Advances in Cognitive Load Theory: Rethinking Teaching* (eds Tindall-Ford, S., Agostinho, S. & Sweller, J.) 1st edn., 1–11 (Routledge, 2019).
89. Petzold, L. R. Description of DASSL: A differential/algebraic system solver. Sandia Report SAND82-8637 (Sandia National Laboratories, Albuquerque, New Mexico; Livermore, California, 1982).
90. Hastings, J. et al. ChEBI in 2016: improved services and an expanding collection of metabolites. *Nucleic Acids Res.* **44**, D1214–D1219 (2016).
91. Cook, D. L., Bookstein, F. L. & Gennari, J. H. Physical properties of biological entities: an introduction to the ontology of physics for biology. *PLoS ONE* **6**, e28708 (2011).
92. Neal, M. L. et al. Semantics-based composition of integrated cardiomyocyte models motivated by real-world use cases. *PLoS ONE* **10**, e0145621 (2015).

93. Justus, N., Schölzel, C., Dominik, A. & Letschert, T. Mo|E—a communication service between Modelica compilers and text editors. In *Proceedings of the 12th International Modelica Conference*, 815–822 (Prague, Czech Republic, 2017).
94. Schölzel, C., Blesius, V., Ernst, G. & Dominik, A. Characteristics of mathematical modeling languages that facilitate model reuse in systems biology: a software engineering perspective. *bioRxiv*. Preprint at https://doi.org/10.1101/2019.12.16.875260v6 (2021).
95. Schölzel, C. CSchoel/shm-conduction: Release v1.1.1. Zenodo https://doi.org/10.5281/zenodo.4585654 (2021).

ACKNOWLEDGEMENTS

We thank our four anonymous reviewers for their much valued input which greatly improved the quality of this article. C. Schölzel would like to thank Denis Noble, Peter Hunter, James Bassingthwaighte, Maxwell Neal, and Herbert Sauro for insightful discussions about the IUPS and NSR Physiome projects, CellML, and present and future challenges for systems biology. We also thank Alexander Goesmann for his advice regarding the focus of the article and Jochen Blom, Björn Pfarr and Annina Hofferberth for proofreading the manuscript.

AUTHOR CONTRIBUTIONS

C.S., V.B., and A.D. conceived the project, C.S. implemented the models and performed the experiments, V.B. and G.E. provided physiological consultation and criticism, C.S. drafted the manuscript and V.B., G.E., and A.D. revised it.

FUNDING

Open Access funding enabled and organized by Projekt DEAL.

COMPETING INTERESTS

The authors declare no competing interests.

ADDITIONAL INFORMATION

Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s41540-021-00182-w.

Correspondence and requests for materials should be addressed to C.S.

Reprints and permission information is available at http://www.nature.com/reprints

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/.

© The Author(s) 2021

8.2. An understandable, extensible and reusable implementation of the Hodgkin-Huxley equations using Modelica

Title	An understandable, extensible, and reusable implementation of the Hodgkin-Huxley equations using Modelica
Authors	C. Schölzel, V. Blesius, G. Ernst, and A. Dominik
Publication type	Journal article
Journal	Frontiers in Physiology
Publisher	Frontiers Media
Year	2020
Volume	11
Article number	583203
DOI	10.3389/fphys.2020.583203

Contributions by dissertation author (using the Contributor Roles Taxonomy):

Conceptualization	Conceived project, formulated research goals
Data curation	Maintained model code on GitHub, Zenodo, and BioModels
Investigation	Performed all experiments
Methodology	Applied cognitive load theory to model code
Software	Implemented the HH-modelica model
Validation	Implemented test script for HH-modelica model
Visualization	Created all figures and tables
Writing — original draft	Wrote initial draft
Writing — review & editing	Revised manuscript due to feedback of other authors and reviewers

frontiers in Physiology

TECHNOLOGY AND CODE
published: 02 October 2020
doi: 10.3389/fphys.2020.583203

An Understandable, Extensible, and Reusable Implementation of the Hodgkin-Huxley Equations Using Modelica

Christopher Schölzel[1]*, *Valeria Blesius*[1], *Gernot Ernst*[2,3] and *Andreas Dominik*[1]

[1] Life Science Informatics, Technische Hochschule Mittelhessen - University of Applied Sciences, Gießen, Germany, [2] Vestre Viken Hospital Trust, Kongsberg, Norway, [3] Psychological Institute, University of Oslo, Oslo, Norway

The Hodgkin-Huxley model of the squid giant axon has been used for decades as the basis of many action potential models. These models are usually communicated using just a list of equations or a circuit diagram, which makes them unnecessarily complicated both for novices and for experts. We present a modular version of the Hodgkin-Huxley model that is more understandable than the usual monolithic implementations and that can be easily reused and extended. Our model is written in Modelica using software engineering concepts, such as object orientation and inheritance. It retains the electrical analogy, but names and explains individual components in biological terms. We use cognitive load theory to measure understandability as the amount of items that have to be kept in working memory simultaneously. The model is broken down into small self-contained components in human-readable code with extensive documentation. Additionally, it features a hybrid diagram that uses biological symbols in an electrical circuit and that is directly tied to the model code. The new model design avoids many redundancies and reduces the cognitive load associated with understanding the model by a factor of 6. Extensions can be easily applied due to an unifying interface and inheritance from shared base classes. The model can be used in an educational context as a more approachable introduction to mathematical modeling in electrophysiology. Additionally the modeling approach and the base components can be used to make complex Hodgkin-Huxley-type models more understandable and reusable.

Keywords: understandability, cognitive load theory, Modelica, mathematical modeling, software engineering, model engineering, Hodgkin-Huxley, action potential

OPEN ACCESS

Edited by:
Sanjay Ram Kharche,
University of Western Ontario, Canada

Reviewed by:
Dominic G. Whittaker,
University of Nottingham,
United Kingdom
Bradley John Roth,
Oakland University, United States

***Correspondence:**
Christopher Schölzel
christopher.schoelzel@mni.thm.de

Specialty section:
This article was submitted to Computational Physiology and Medicine, a section of the journal Frontiers in Physiology

Received: *14 July 2020*
Accepted: *31 August 2020*
Published: *02 October 2020*

Citation:
Schölzel C, Blesius V, Ernst G and Dominik A (2020) An Understandable, Extensible, and Reusable Implementation of the Hodgkin-Huxley Equations Using Modelica. Front. Physiol. 11:583203. doi: 10.3389/fphys.2020.583203

1. INTRODUCTION

Since 1952, when Alan Hodgkin and Andrew Huxley published their conductance-based model of the action potential generation in the squid giant axon, the Hodgkin-Huxley (HH) model has been the basis of countless research projects to further the understanding of ionic currents and action potentials in neurons and cardiac myocytes (Hodgkin and Huxley, 1952). Today's models feature more types of ion channels and pumps than the three channels identified by Hodgkin and Huxley, but they still use the same electrical analogy and equation structure, which is why we will call them HH-type models in the following (Courtemanche et al., 1998; Inada et al., 2009; Fabbri et al., 2017; Bai et al., 2018). Although other types of ion channel models, such as Markov

Models, are emerging, HH-type models are still the gold standard (Winslow et al., 1999; Fink and Noble, 2009). The descriptions of these HH-type models usually follow one of three explanatory approaches: Either the differential equations are given directly with a short biological explanation of the major variables or a diagram of the electrical analogy is shown and explained in biological terms or a combination of both. Often a biological drawing of the cell is also provided, but it is only used to explain the modeled concepts and not tied to the model equations themselves. This holds for research articles (Courtemanche et al., 1998; Inada et al., 2009), simulation toolkits (Hines and Carnevale, 1997; Jordan et al., 2020) and textbooks (Voit, 2013; Gerstner et al., 2014). However, these approaches pose significant challenges for novices and limit the productivity of experts: A novice has to become familiar with the formalism of differential equations or with circuit diagrams at the same time as they try to understand the model itself. Experts will have overcome this barrier already, but they are also faced with much more complex HH-type models that can easily grow to over 100 equations. These equations all interact with each other in a multitude of feedback loops, making it extremely difficult to spot small errors or to reproduce and extend the model. The risk associated with these barriers is 2-fold: On the one hand, students may choose not to specialize in systems biology or electrophysiology, because they perceive the field to be too difficult. On the other hand, a published model that is described in this way may generate new insights, but prove to be too hard to reuse and extend. For example, the latter seems to be the case for a model by Inada et al. (2009) (116 equations) which has been labeled as "groundbreaking" (Noble et al., 2012) but has only been used for simulations by two other research groups in 10 years. It becomes apparent that there is room for improving the understandability of HH-type models and that this should be a goal of both the initial model design and its presentation in scientific and instructional material.

One area from which such an improvement may originate is software engineering, because software development faces similar problems of understandability: The building blocks of source code are easy to grasp, but creating and maintaining projects with millions of lines of code requires additional organization. A widely established solution to handle this complexity is modularization. Instead of overseeing the whole project at once, software engineers identify individual functions and parts of a system and create small modules to represent them. Each component only has a few lines of code and a limited number of connections to the outside world which makes it understandable. To form the whole system, the modules can be connected at a higher level of abstraction, where each of them can be considered to be a single entity.

In recent years, multiple researchers have advocated to borrow concepts from software engineering for systems biology, culminating in the formulation of the term "model engineering" (Hellerstein et al., 2019). In accordance with this movement, we found that the modeling language and consistent application of relevant language characteristics can have a significant impact on the model quality (Schölzel et al., 2020). In this paper we therefore present a novel modular implementation of the original HH model that is based on the electrical analogy, but explains and visualizes each component in biological terms. The model is written in the modeling language Modelica and makes heavy use of the features of this language and of software engineering techniques.

Due to the aforementioned anticipated benefits of these techniques we pose the following research questions:

RQ1 Can the understandability of the HH model be improved by a modular implementation that bridges the gap between biological meaning and electrical analogy?

RQ2 Can a modular implementation of the HH model serve as a unifying basis for extensions and therefore facilitate the creation of more complex HH-type modules?

For the investigation of these questions, the term *understandability* is central, as our model does not differ from other solutions in terms of its output, but only its presentation. We may find our assessment of what makes a model more or less understandable intuitive, but in a scientific context it is not sufficient to rely on intuition alone. This is especially true, when it involves reasoning about the experiences of other people which may have quite a different background and perspective. Therefore, a model for understandability is needed that is based on scientific evidence. For this task we use cognitive load theory (CLT), a popular and well-validated theory in cognitive psychology which frames understandability in terms of the amount of items that have to be kept in active working memory and the degree of interactivity between them.

CLT is introduced in more detail in section 2.1 along with model engineering, the language Modelica, and the biological basis of the Hodgkin-Huxley model followed by an overview over the software engineering concepts that we apply and the resulting model structure. Section 2 describes our rationale for reducing cognitive load, for the component hierarchy, and the design of the individual model components. Section 3 then shows and explains the resulting model code including the graphical representation followed by a discussion of the implications of the new model structure for understandability and extensibility. Finally, section 4 sums up the answers to our research questions and discusses possible alternative approaches, limitations and future work.

2. MATERIALS AND METHODS

2.1. Background

2.1.1. Model Engineering

To this day, many models are still built for a single purpose without guidelines regarding code quality. However, when models grow beyond a certain point, the modeling process becomes an engineering task and the goal should not only be to produce a model that is working and mathematically sound, but also to build it with an architecture that facilitates the anticipated use cases and makes the code maintainable and accessible to other researchers (Hellerstein et al., 2019). This includes documentation, testing, naming of variables, and the use of established design patterns. In a previous work we have found the consistent use of an appropriate modeling language that is modular, descriptive, (human)-readable, open, graphical,

and hybrid (MoDROGH) to be a major driving factor for model quality in terms of understandability and reproducibility (Schölzel et al., 2020).

2.1.2. Modelica and Object-Oriented Programming

There are a few established and emerging languages that exhibit MoDROGH-characteristics, such as the systems biology markup language (SBML) (Hucka et al., 2003), CellML (Cuellar et al., 2003), Simscape (The MathWorks, Inc., 2020), or embedded domain-specific languages (DSLs) written in Python or Julia (Olivier et al., 2005; Lopez et al., 2013; Elmqvist et al., 2016; Rackauckas et al., 2020)[1]. In our opinion, the modeling language Modelica (Mattsson and Elmqvist, 1997) is a particularly interesting example, because it is an industrial standard that emphasizes the engineering aspect of model design. In contrast to SBML, CellML, as well as Python- and Julia-based DSLs, Modelica supports full object oriented design (e.g., through model inheritance), discrete variables for the seamless integration of continuous and discrete model parts, the graphical composition of models via drag and drop, implicit differential/algebraic equations for acausal connections between components via conservation laws, cross-language export and import via the Functional Mockup Interface (Blochwitz et al., 2012), grouping of interface variables to connectors, and unrestricted mixing of implicit and explicit equation formats. It shares these features with Simscape, but unlike Simscape, Modelica provides an open environment, more flexible mechanisms for model inheritance including multiple inheritance and overwriting of variables and equations, and extensible annotations, which could, e.g., be used to implement support for ontological terms to the language. Weak points of Modelica are the lack of existing support for biological terminology and ontologies and the fact that, while the OpenModelica integrated development environment (IDE) (Fritzson et al., 2005) is open-source, many users prefer the proprietary IDE Dymola (Dassault Systèmes, 2020), which has its own compiler that is not always fully compatible with OpenModelica.

In Modelica, modularity is realized by the principles of object-oriented programming (Gamma et al., 1994). Code is structured in classes that can be instantiated to reuse the same code at different positions in a project and that can inherit behavior and interfaces from abstract base classes. This reuse is not only encouraged from one project to the following but also within one project. This corresponds with one of the guiding principles in software engineering called "don't repeat yourself" (DRY) suggesting that one should avoid writing duplicated code with only very small differences, such as constant values or variable names (Hunt and Thomas, 2000). Both DRY and object orientation are most effective, when the implemented system can be broken down into structurally similar components, which is the case for the HH model.

[1]For a comparison of different language candidates see the supplementary note in Schölzel et al. (2020).

2.1.3. The Biological Basis of the Hodgkin-Huxley Model

The Hodgkin-Huxley model explains the time course of the membrane potential of the squid giant axon during an action potential by means of three ion channels: A sodium channel lets Na^+ cations enter the cell, which increases the potential. A delayed-rectifier potassium channel permits K^+ cations to leave the cell, lowering the potential back toward the resting state. Finally, a leak channel is responsible for maintaining the resting potential while the other channels are closed. Both the sodium and the potassium channel have voltage-dependent gates—molecules that change their conformation with the membrane potential to activate or inactivate the channel. The sodium channel has both a fast activation gate and a slightly slower inactivation gate, allowing the channel to open for a short period of only a few milliseconds. The delayed-rectifier potassium channel only has an activation gate with slower kinetics while the leak channel is assumed to be always active.

2.1.4. Cognitive Load Theory

As mentioned in the introduction, we use cognitive load theory (CLT) as a model for understandability (Sweller, 2019). In short, CLT is based on the architecture of the human brain, which has a very limited capacity for new information in the working memory, but can easily transfer stored information from long-term memory to working memory. The amount of items that have to be kept in working memory to process an information is called the cognitive load. The main driving factor of this metric is element interactivity. Independent elements can be processed one by one, but when elements have high interactivity they have to be kept in working memory simultaneously. Hence, cognitive load can be reduced in two ways: First, expertise can allow a person that has understood a concept to further on process it as a single item instead of the several items it comprises. Second, a part of cognitive load does not originate from the complexity of the taught concept itself, but from the way it is presented and can therefore be reduced by choosing appropriate methods to present the information and instruct the learner.

2.2. Model Design and Structure

For the sake of simplicity, we will consider the amount of variables, parameters, and equations that constitute a model or model component as an indicator of its cognitive load and thus its understandability. Our goal is therefore to produce small model components that have a low amount of elements and to reduce element interactivity by introducing clearly defined interfaces between these components that allow the learner to view one component as a single item to be kept in working memory once they understand it.

For this task we identify the following components in the HH model: The lipid bilayer that separates the ionic concentrations and therefore electrical charge on the outside and the inside of the cell; the sodium, potassium, and leak channels; the voltage-dependent gates inside the sodium and potassium channels; and the current clamp that holds the current constant in order to measure the voltage with reference to a ground electrode. All of

these components reside on the same hierarchical layer, except for the gates which are components of an ion channel class.

To build the full model, two kinds of connections between non-gate components are required: First, each component has a positive electrical pin on the extracellular side and a negative electrical pin on the intracellular side, allowing current to flow through the component as positive outward current. Second, the voltage-dependent gating molecules of the sodium and potassium channels react more quickly at higher temperatures, which establishes the need for an input-output temperature connection from the lipid bilayer to the ion channels.

In the design of the Modelica model we follow our guidelines established in Schölzel et al. (2020), i.e., implementing small self-contained modules; describing only the "what" of the model, not the "how"; keeping the code human-readable with speaking names and additional documentation strings for variables and parameters; using only open-source tools (namely OpenModelica) for ease of reproduction; and adding a graphical representation for each component. We also published the full code of the model on GitHub as well as in the **Supplementary Data Sheet 1** including every information required to reproduce our results. As mentioned in section 2.1.2, graphical annotations are part of the Modelica code. This encompasses connection annotations that define the coordinates of the line connecting two components and more complex icon annotations that define a component icon as vector graphic. As the latter can be quite large and tend to clutter the otherwise human-readable code, we define all icons in separate classes and include them via inheritance with the `extends` statement.

The model was implemented using OpenModelica version 1.16.0 (Fritzson et al., 2005) and Mo|E version 0.6.3 (Justus et al., 2017) as well as Inkscape version 0.91 (Inkscape Developers, 2020) to add the component icons. Simulations can be replicated with OpenModelica on Windows, Linux, and macOS. The code is available on GitHub under the MIT license at https://github.com/CSchoel/hh-modelica.

3. RESULTS

3.1. Model Code

The first thing to consider in a software engineering task are the required interfaces. Hence, we start our implementation of the HH model by defining the following basic connectors:

```
connector TemperatureInput = input Real(unit="degC");
connector TemperatureOutput = output Real(unit="degC");
connector ElectricalPin
  flow Real i(unit="uA/cm2");
  Real v(unit="mV");
end ElectricalPin;
```

Here, `TemperatureInput` and `TemperatureOutput` follow a simple input-output relationship. All components that have a `TemperatureInput` will be connected to a single component with a `TemperatureOutput` that determines the global temperature value. `ElectricalPin`s that are connected to each other will all have the same voltage `v`, but can have different currents `i` (indicated by the keyword `flow`). During compilation, Modelica will generate an equation following Kirchhoff's current law that ensures that the sum of all connected currents equals zero. This allows to connect an arbitrary number of components without having to determine the direction of the flow. For the sake of terminology and for a visual distinction `PositivePin` and `NegativePin` are introduced as subclasses without any functional difference from the base class.

```
connector NegativePin
  extends ElectricalPin;
  annotation(...);
end NegativePin;

connector PositivePin
  extends ElectricalPin;
  annotation(...);
end PositivePin;
```

Annotation code that defines the connector icons is not given here for the sake of brevity and will be completely omitted for further code examples. The same is true for most of the documentation strings. These details can be viewed on GitHub and the resulting visual design can be seen in **Figure 1**.

Since most components will have an electrical connection both to the inside and the outside of the cell, it is beneficial to introduce another base class for those two-pin components:

```
partial model TwoPinComponent
  PositivePin p "positive extracellular pin";
  NegativePin n "negative intracellular pin";
  Real v(unit="mV") "potential difference between pins";
  Real i(unit="ua/cm2") "current flowing through comp.";
equation
  0 = p.i + n.i;
  v = p.v - n.v;
  i = p.i;
end TwoPinComponent;
```

This base class already introduces three small equations that connect the positive (extracellular) and negative (intracellular) pins. The first equation again follows Kirchhoff's current law to ensure that the sum of all currents entering and leaving the components is zero. The other equations just introduce two helper variables: The variable `v` can be used to measure (or define) the voltage at this component as a difference between the potential at the positive and the negative pin. The variable `i` measures or defines the current flowing through the component from the negative to the positive pin. The model is declared as `partial` since the number of equations and variables is not balanced. It does not yet specify the current-voltage relationship but leaves it open for implementation in specific subclasses.

The simplest `TwoPinComponent` that specifies this relationship is the `LipidBilayer`:

```
model LipidBilayer
  extends TwoPinComponent;
  extends HHmodelica.Icons.LipidBilayer;
  TemperatureOutput temp = temp_m;
  parameter Real temp_m(unit="degC") = 6.3 "temperature";
  parameter Real c(unit="uF/cm2") = 1 "capacitance";
  parameter Real v_init(unit="mV") = -90 "initial stim.";
initial equation
  v = v_init;
equation
  der(v) = 1000 * i/c;
end LipidBilayer;
```

This model inherits the variables and equations of `TwoPinComponent` and the graphical annotations from the icon `LipidBilayer`. It therefore only has to introduce one additional equation that describes the component as a capacitor, which separates the inside from the outside and is charged when a current is applied. A factor of 1,000 has to be introduced to measure the derivative `der(v)` in millivolt instead of volt per second. Since the voltage `v` only enters the equation as a derivative, the initial value has to be determined. Hodgkin and Huxley used this degree of freedom to simulate a "short initial stimulation" of the Membrane, assuming that it has been kept at a constant $V = 0$ until the time $t = 0$ where the voltage is suddenly changed to $V = V_{\text{init}}$ and is then left to develop under a constant current. In this implementation this behavior is reflected by the parameter `v_init`. Additionally, the membrane temperature is defined through the parameter `temp_m` and propagated via the output connector `temp`.

The temperature is passed on to the `Gate`s, which describe the conformation changes of gating molecules in an ion channel:

```
model Gate "molecule that opens/closes an ion channel"
  replaceable function fopen = expFit(sx=1, sy=1);
  replaceable function fclose = expFit(sx=1, sy=1);
  Real n(start=fopen(0)/(fopen(0) + fclose(0){}},
    fixed=true);
  input Real v(unit="mV");
  TemperatureInput temp;
protected
  Real phi = 3^((temp-6.3)/10);
equation
  der(n) = phi * (fopen(v) * (1 - n) - fclose(v) * n);
end Gate;
```

In contrast to the `LipidBilayer` the `Gate` does not inherit from `TwoPinComponent`, since it is not a component in the electric circuit itself but only a part of the `IonChannel`. Here, `n` is the gating variable that determines the ratio of gating molecules that are in "open" conformation. The rates with which molecules change formations depend on the current voltage `v` through the functions `fopen`, which gives the rate of change from closed to open, and `fclose`, which gives the rate of change from open to closed. Instead of having variables α and β that change with an equation as in the original formulation by Hodgkin and Huxley, `fopen` and `fclose` actually can be seen as variables that store the whole fitting functions. This allows us to keep the code DRY by reusing these functions to determine the starting value for `n` as the steady state that would be achieved by holding the membrane voltage constant as $V = 0\,\text{mV}$. In the original model, a change in one of the fitting parameters for the equation for α would also require a change in the stating value for n which is not immediately transparent by the description. The functions `fopen` and `fclose` are explicitly declared as `replaceable` so that each ion channel can redefine them as required. For this the three fitting functions `expFit`, `logisticFit` and `goldmanFit` are required in the original HH model. For the sake of simplicity we will only discuss `expFit` here:

```
function expFit "exponential fitting function"
  input Real x "input value";
  input Real sx "scaling factor for x axis";
  input Real sy "scaling factor for y axis";
  output Real y "result";
algorithm
  y := sy * exp(sx * x);
end expFit;
```

In function definitions, Modelica switches from the usual declarative implementation style to an imperative style as in C or MATLAB. In an `algorithm` section, equations are variable assignments where an expression on the right-hand side is evaluated and stored in the variable on the left-hand side. This is also indicated by the assignment operator `:=` which has a direction in contrast to the equals sign used for normal equations. During compilation, an algorithm section is transformed to a single equation that depends on all input variables and determines the value of all output variables of the function definition. Here, a simple exponential relationship is defined between the main input `x` and the output `y` that can be scaled by the additional fitting parameters `sx` and `sy`. These fitting parameters are fixed to a constant value when the function is instantiated as, for example, `function fclose = expFit(sx=1, sy=1)`. The resulting function now has only `x` left as the single mandatory input and can therefore be called as `fclose(x)` for any real value `x`. The functions `logisticFit` and `goldmanFit` follow the same general structure, but realize different fitting functions with additional fitting parameters.

As mentioned before, the `Gate` component is part of an `IonChannel` component. As there are three different ion channels in the HH model, it again makes sense to introduce a common base class:

```
partial model IonChannel "ionic current through membrane"
  extends TwoPinComponent;
  extends HHmodelica.Icons.IonChannel;
  Real g(unit="mmho/cm2") "ion conductance";
  parameter Real v_eq(unit="mV") "equilibrium potential";
  parameter Real g_max(unit="mmho/cm2") "max conduct.";
equation
  i = g * (v - v_eq);
end IonChannel;
```

This component is a two-pin component and inherits a graphical annotation from an icon component. It adds the missing relationship between current and voltage by introducing a conductance variable `g`. If `g` is constant, the `IonChannel` behaves as a simple electrical conductor with the only exception that the voltage is relative to the equilibrium potential for the ions transported by this channel. This is true for the `LeakChannel` which only introduces the additional equation `g = g_max`. The sodium and potassium channels, however, have voltage- and temperature-dependent gates. Therefore, `GatedIonChannel` is introduced as another base class that is the same as `IonChannel` but with an additional `TemperatureInput` called `temp`. The delayed-rectifier potassium channel, which lets K^+ cations pass through the membrane when it is open, then becomes:

```
model PotassiumChannel "channel selective for K+ cations"
  extends GatedIonChannel(g_max=36, v_eq=12);
  extends HHmodelica.Icons.Activatable;
  Gate gate_act(
  redeclare function fopen=
      goldmanFit(x0=-10, sy=100, sx=0.1),
    redeclare function fclose= expFit(sx=1/80, sy=125),
    v=v, temp=temp
```

```
    )"activation gate";
 equation
   g = g_max * gate_act.n ^ 4;
 end PotassiumChannel;
```

This component inherits variables and equations from `GatedIonChannel` and at the same time changes the values for the maximum conductance and the equilibrium potential. It uses a `Gate` component and redeclares the appropriate fitting functions to use for `fopen` and `fclose`. The only additional equation introduced is the dependency between the conductance and the gating variable `n`. As Hodgkin and Huxley determined, four gating molecules have to be in the open conformation simultaneously in order to allow ion transfer through a delayed-rectifier potassium channel. This is realized by taking the fourth power of the gating variable `n`.

The sodium channel, which lets Na^+ cations pass through the membrane, looks similar but a little more complex:

```
model SodiumChannel "channel selective for Na+ cations"
  extends GatedIonChannel(g_max=120, v_eq=-115);
  extends HHmodelica.Icons.Activatable;
  extends HHmodelica.Icons.Inactivatable;
  Gate gate_act(
  redeclare function fopen=
      goldmanFit(x0=-25, sy=1000, sx=0.1),
    redeclare function fclose=
      expFit(sx=1/18, sy=4000),
    v=v, temp=temp
    )"activation gate";
  Gate gate_inact(
  redeclare function fopen=
      expFit(sx=1/20, sy=70),
    redeclare function fclose=
      logisticFit(x0=-30, sx=-0.1, y_max=1000),
    v=v, temp=temp
    )"inactivation gate";
 equation
  g = g_max * gate_act.n ^ 3 * gate_inact.n;
 end SodiumChannel;
```

Here we have three molecules that form an activation gate and one molecule that forms an inactivation gate. The gates again only differ in the choice of fitting functions and values for their fitting parameters.

With this we already have all individual components that constitute the cell membrane. To measure and to perform experiments, however, we still need a model of the current clamp which keeps the current through the membrane constant in order to measure the voltage relative to a ground electrode:

```
  model ConstantCurrent
    extends TwoPinComponent;
    parameter Real i_const(unit="ua/cm2");
  equation
    i = i_const;
  end ConstantCurrent;

  model Ground
    PositivePin p;
  equation
    p.v = 0;
  end Ground;

  model CurrentClamp
    extends HHmodelica.Icons.CurrentClamp;
    PositivePin p "extracellular electrode";
    NegativePin n "intracellular electrode(s)";
    parameter Real i_const(unit="ua/cm2") = 40;
    ConstantCurrent cur(i=i_const);
    Ground g "reference electrode";
    Real v = -n.v "measured membrane potential";
  equation
    connect(p, cur.p);
    connect(n, cur.n);
    connect(g.p, p);
  end CurrentClamp;
```

This is the first part in the model where we use Modelica's `connect` equation to connect smaller components to one large component. The `Ground` component simply sets the potential of the extracellular compartment to zero while the `ConstantCurrent` component ensures that the cell has a constant positive outward current. The additional variable `v` captures the actual membrane potential that would be measured by a real current clamp experiment.

Now that all components are defined, putting together the whole HH model becomes as simple as just placing them side by side, connecting positive with positive and negative with negative pins of neighboring components as well as connecting the `TemperatureOutput` of the `LipidBilayer` to all `TemperatureInput` connectors:

```
model HHmodular
  PotassiumChannel c_pot;
  SodiumChannel c_sod;
  LeakChannel c_leak;
  LipidBilayer l2;
  CurrentClamp clamp;
equation
  connect(l2.p, c_pot.p);
  connect(c_pot.p, c_sod.p);
  connect(c_sod.p, c_leak.p);
  connect(c_leak.p, clamp.p);
  connect(clamp.n, c_leak.n);
  connect(c_leak.n, c_sod.n);
  connect(c_sod.n, c_pot.n);
  connect(c_pot.n, l2.n);

  connect(l2.temp, c_pot.temp);
  connect(c_pot.temp, c_sod.temp);
end HHmodular;
```

Annotations can be used to place the components on a coordinate system and to give the connections a graphical representation. Usually these annotations are not written manually but generated by an IDE like OpenModelica, where the components can be placed on a diagram view via drag and drop. Due to restrictions in space we do not show all the annotations here but only an example for the placement of the `LipidBilayer` and the connection between the `LipidBilayer` and the `PotassiumChannel`:

```
...
  LipidBilayer l2 annotation(
    Placement(visible = true, transformation(
      origin = {-67, 3},
      extent = {{-17, -17}, {17, 17}},
     rotation = 0
      ))
     );
...
   equation
    connect(l2.p, c_pot.p) annotation(
     Line(
      points = {{-66, 20},{-66, 40},{-33, 40},{-33, 20}},
```

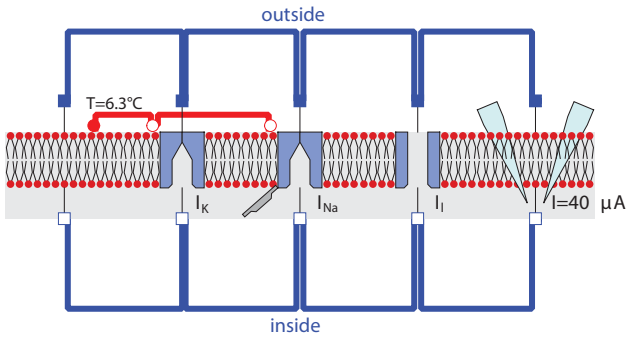


FIGURE 1 | Diagram view of the modular Modelica implementation of the Hodgkin-Huxley model. Each component has a positive electrical pin on the top/outside and a negative electrical pin on the bottom/inside. From left to right the components are: `LipidBilayer`, `PotassiumChannel`, `SodiumChannel`, `LeakChannel`, and `CurrentClamp`. Extra connections in red represent temperature dependence of gating variables.

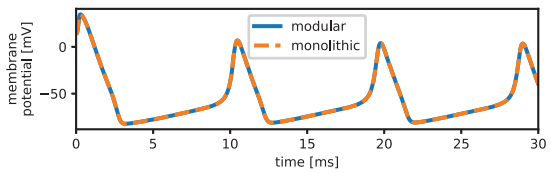


FIGURE 2 | Comparison of the monolithic and modular versions of the HH model. The plot shows perfect alignment of the voltage curves. Note that we plot the membrane potential V_m as difference between the potential on the inside and the potential on the outside of the cell. This conforms with current standards, but is in contrast to the original equations by Hodgkin and Huxley, which define V as the displacement from the resting potential with opposite sign. We used the equation $V_m = E_r - V$, assuming a resting potential E_r of −75 mV, which is also used in the HH-implementation in the BioModels database (Le Novère, 2020) and corresponds to the resting potential of the squid giant axon *in vivo* (Moore and Cole, 1960).

```
      color = {0, 0, 255}
       )
      );
...
```

The resulting diagram is shown in **Figure 1**. The lipid bilayer is represented similar to diagrams in biology textbooks with circles on the outside and curved black lines pointing to the inside of the membrane. The channels are displayed as pores that are either open for the leak channel or closed for channels that have to be activated for ions to pass. The sodium channel additionally has a hinged lid to represent the inactivation gate. Finally, the current clamp is represented by two electrodes piercing the membrane.

3.2. Model Validation

When compiled, the modular version of the HH model reduces to the exact same equation system as the original monolithic version. Due to the modularization there are multiple versions of one variable, but this only leads to the addition of a few trivial equations of the form $x = y$ or the form $x = -y$. **Figure 2** shows a plot of the monolithic and modular version to ensure that both are functionally equivalent. This means that there are now two very different implementations with the same functionality which can and have to be analyzed for their suitability according to the research questions we established in our introduction.

3.3. Assessment of Understandability (RQ1)

With RQ1, we asked if the understandability of the HH model can be improved by a modular implementation that bridges the gap between biological meaning and electrical analogy. Using cognitive load theory as a framework, we will assume that a model is more understandable if it requires fewer items to be kept in working memory simultaneously. Although the modular implementation has more lines of code in total, it separates the model into small digestible parts. When a novice, for example, wants to know what a `Gate` is, they only have to process five variables and one equation, each of which are documented with their physiological meaning. When they understand this component, they know that its purpose is to produce a value between zero and one which is based on voltage and temperature and represents the ratio of gating molecules in open conformation. Once this concept is stored in long-term memory it can be recalled as a single item into working memory. This means that, when the learner moves on, the two gates in `SodiumChannel` can be processed as two items instead of twelve[2]. Moving from component to component, the modular version presents the reader with at most two equations and five variables or parameters at the same time. This constitutes a very low cognitive load compared to the 15 equations and 33 variables and parameters of the monolithic version that are presented all at once or in loose groups without clearly defined interfaces.

One way to facilitate the transfer of new concepts in long-term memory is to anchor them to existing knowledge. Our implementation does this by annotating each component and each variable or parameter in that component with biological terms[3]. The implementation also uses speaking variable names wherever possible to keep a close link between the biological and the mathematical representation—a common best practice in software engineering (that could also be applied to a monolithic version). Some parts of the model, such as the seemingly arbitrary fitting function `goldmanFit`, require more explanation which can be given in Modelica by adding an HTML string to the component for a detailed documentation.

Finally, on the highest hierarchical level, the component `HHmodular` has still only five variables but ten connect equations. For this model, however, a novice does not need to read any code at all to understand it, because they can use the diagram in **Figure 1** instead. Since it is defined directly in the code and tied to the individual components it is not only a simplified documentation but an accurate graphical reflection of

[2]This benefit is even more pronounced considering that the formulas for the α and β variables in the original model each consist up to seven seemingly arbitrary mathematical operations that are now given a meaning by introducing the named and documented fitting functions `expFit`, `logisticFit`, and `goldmanFit`.
[3]Due to spatial limitations we do not show all these annotations in this paper, but they can be found at https://github.com/CSchoel/hh-modelica/.

the underlying implementation. This means that understanding the model on this level of abstraction is not any harder than understanding a corresponding biological drawing in a textbook. Assuming that the learner is already familiar with *what* is modeled, the implementation can again be anchored easily to that existing knowledge.

3.4. Assessment of Extensibility and Reusability (RQ2)

With RQ2, we asked if a modular implementation of the HH model can serve as a unifying basis for extensions and therefore facilitate the creation of more complex HH-type modules. The current implementation already could reduce some duplicate equations in the original model by reusing already existing code. The current-voltage relationship of the three ion channels, the rate of conformation change in the three different gates, and the fitting functions `expFit` and `goldmanFit` each had to be defined only once. Furthermore, the introduction of the replaceable functions `fopen` and `fclose` eliminated the need to define starting values for the gating variables. When new components are added to the model, it is highly probable that some of these existing components can be used to reduce the implementation effort. Even more importantly, the argument for the reduction of cognitive load by modularization gets more weight as the model grows in size. In the modular version, the only point where cognitive load may increase due to extensions and therefore make the model less understandable is when there are too many individual models at the highest hierarchical layer. However, even then it is possible to form groups of components (e.g., a group for all potassium-sensitive channels or for all ion pumps) that are then connected on a new even higher hierarchical level. Conversely, the cognitive load associated with a monolithic model will grow with each variable and each equation that is added to the model.

To give one specific example, a reasonable extension could be the inclusion of slow inactivation of sodium channels. In contrast to fast inactivation, that stops the influx of Na^+ cations after a few milliseconds, slow inactivation takes place over seconds or even minutes of prolonged or high frequency depolarizations, reducing the number of sodium channels available for activation (Payandeh, 2018). This could be realized by simply adding another `Gate` component to the `SodiumChannel` and introducing a ratio `p_slow` that determines how much of the total current is attributed to slow as opposed to fast inactivation. The conductance equation would then change from

```
g = g_max * gate_act.n ^ 3 * gate_inact.n;
```

to:

```
g = g_max * gate_act.n ^ 3
  * (p_slow * gate_inact_slow.n + (1-p_slow) * gate_inact_fast
  .n);
```

Apart from choosing appropriate fitting functions for the new gate, this would be the only change required. Arguably a monolithic model would not require more changes, but it would be more difficult to first identify which equations have to change and thus it would be easier to make a mistake by missing or interchanging an equation or variable. We encountered this problem in a previous work with a model of the human cardiac conduction system (Schölzel et al., 2020).

Other extensions might involve defining an alternative `Gate` that uses fitting functions to determine the steady state n_∞ and time constant τ instead of α (`fopen`) and β (`fclose`) (Destexhe and Huguenard, 2000; Goldman et al., 2001) or new components based on `TwoPinComponent`, such as channel formulations based on the Goldman-Hodgkin-Katz flux equation (Huguenard and McCormick, 1992; Destexhe and Huguenard, 2000) or models of ionic pumps (Di Francesco and Noble, 1985; Matsuoka and Hilgemann, 1992). Even in these cases the underlying interfaces can stay the same and parts like the current clamp formulation, common base classes, and fitting functions can be reused.

4. DISCUSSION

We showed that a modular version of the HH model that uses software-engineering techniques to manage complexity is beneficial both for novices and for experts, answering both of our research questions in the affirmative.

RQ1 asked whether the understandability of the HH model can be improved by a modular implementation. We showed that this is the case using CLT as framework and demonstrating a drop of the cognitive load by a factor of 6. The biological concepts can be explained and understood one at a time with an accurate graphical representation at the highest level of abstraction instead of having to navigate through a multitude of equations and variables with high element interactivity. In summary this means that with our implementation a deeper understanding of the HH model can be achieved in less time and it is likely that novices learning the model in this way will make fewer errors when recalling the learned concepts at a later time.

RQ2 asked whether the modular implementation can also serve as a unifying basis for extensions and facilitate the creation of more complex HH-type models. We showed that, in contrast to the monolithic version, adding new components does not significantly increase the cognitive load associated with the model. We also demonstrated that many components of our model are easily reusable which reduces development time and increases interoperability of solutions.

Similar results like ours would have been possible, for example, using CellML, or SBML with the SBML-comp package. In fact, Wimalaratne et al. (2009) also used the Hodgkin-Huxley model as an example to promote the support for hierarchical composition of CellML models. However, we used some Modelica features for our design that do not exist in these other languages. This includes the graphical composition of models, object-oriented programming with multiple inheritance, acausal connections between electrical and chemical components, the grouping of interface variables to connectors, and the annotation of the experiment setup within the model file itself.

One limitation of this approach is that some experts might be much more familiar with the formalism of differential equations than with object-oriented software design. It might be easier for them to reduce a group of equations to a single item in

their working memory than it is to do the same with a piece of code that represents a class. This means that, for models of small to moderate size, the navigation through different classes according to a modular design structure might actually be detrimental to their understanding of the model and to their productivity when working with it. A solution to this problem could be to provide a third, equation-based representation of the respective model in parallel to the existing graphical view and the raw code. Authoring tools like OpenCOR for CellML (Garny and Hunter, 2015) or COPASI for SBML (Hoops et al., 2006) already provide these equation-based views. However, they do not provide an overview of all equations in the whole model, but only of the parts that are currently selected. For Modelica we are only aware of a similar approach to OpenCOR and COPASI in the proprietary IDE MapleSim (Maplesoft, 2020). Implementing such a representation in open-source tools, such as OpenModelica would be possible due to the fact that, like CellML and SBML, Modelica is declarative. OpenModelica already has a feature to "instantiate" a model, which reduces its structure to a "flat" format consisting of a single class with a list of parameters, variables, and equations. Additionally, OpenModelica models can be exported in an XML format that contains all parameters, variables, and equations in a machine-readable form. Based on these existing features, an "equation view" could be implemented in the OpenModelica IDE OMEdit that would allow experts to understand a model at first glance based on the differential equations and without having to traverse its hierarchical structure. Alternatively, such a representation could be part of a documentation website associated with a model or model library. As an added benefit a tool that provides such an automated representation as typeset equations could also provide an export as LaTeX or Word documents, which can then be inserted in articles to guarantee that the published version of the equations is exactly the same as the equations used to simulate the model. We have implemented a first prototype of such an equation-based web documentation using the Julia package Documenter.jl (Piibeleht et al., 2020) and our own package ModelicaScriptingTools.jl (Schölzel, 2020). The resulting experimental documentation for the Hodgkin-Huxley model presented in this article can be found at https://cschoel.github.io/hh-modelica/dev/.

Another direction for future research is the application of our techniques to larger and more complex models. We are already using the model developed in this paper as a basis to reproduce a large 116 equation model of the atrioventricular node (Inada et al., 2009). We hope that this and other projects based on the same methodology, be it with Modelica or another MoDROGH language like CellML, may help to increase the quality and speed of scientific progress in systems biology.

DATA AVAILABILITY STATEMENT

The original contributions presented in the study are publicly available. This data can be found here: https://github.com/CSchoel/hh-modelica (archived as https://doi.org/10.5281/zenodo.3947848).

AUTHOR CONTRIBUTIONS

CS and AD conceived the project. CS implemented the models and performed the experiments. VB and GE provided the physiological consultation and critique. CS drafted the manuscript. All authors contributed to manuscript revision, read, and approved the submitted version.

FUNDING

Funding for article processing fees was provided by the central publishing fund of the Technische Hochschule Mittelhessen - University of Applied Sciences.

ACKNOWLEDGMENTS

We thank our two reviewers for their valuable comments. We also thank Annina Hofferberth for proofreading the manuscript.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fphys.2020.583203/full#supplementary-material

REFERENCES

Bai, J., Gladding, P. A., Stiles, M. K., Fedorov, V. V., and Zhao, J. (2018). Ionic and cellular mechanisms underlying TBX5/PITX2 insufficiency-induced atrial fibrillation: insights from mathematical models of human atrial cells. *Sci. Rep.* 8:15642. doi: 10.1038/s41598-018-33958-y

Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., et al. (2012). "Functional mockup interface 2.0: the standard for tool independent exchange of simulation models," in *Proceedings of the 9th International Modelica Conference* (Munich), 173–184. doi: 10.3384/ecp12076173

Courtemanche, M., Ramirez, R. J., and Nattel, S. (1998). Ionic mechanisms underlying human atrial action potential properties: insights from a mathematical model. *Am. J. Physiol. Heart Circ. Physiol.* 275, H301–H321. doi: 10.1152/ajpheart.1998.275.1.H301

Cuellar, A. A., Lloyd, C. M., Nielsen, P. F., Bullivant, D. P., Nickerson, D. P., and Hunter, P. J. (2003). An overview of CellML 1.1, a biological model description language. *Simulation* 79, 740–747. doi: 10.1177/0037549703040939

Dassault Systèmes (2020). *Dymola*. Available online at: https://www.3ds.com/products-services/catia/products/dymola/ (accessed August 20, 2020).

Destexhe, A., and Huguenard, J. R. (2000). Nonlinear thermodynamic models of voltage-dependent currents. *J. Comput. Neurosci.* 9, 259–270. doi: 10.1023/A:1026535704537

Di Francesco, D., and Noble, D. (1985). A model of cardiac electrical activity incorporating ionic pumps and concentration changes. *Philos. Trans. R. Soc. Lond. B Biol. Sci.* 307, 353–398. doi: 10.1098/rstb.1985.0001

Elmqvist, H., Henningsson, T., and Otter, M. (2016). "Systems modeling and programming in a unified environment based on Julia," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion,*

Dissemination, Applications, ISoLA 2016, Volume 9953 of Lecture Notes in Computer Science (Corfu), 198–217. doi: 10.1007/978-3-319-47169-3_15

Fabbri, A., Fantini, M., Wilders, R., and Severi, S. (2017). Computational analysis of the human sinus node action potential: model development and effects of mutations. *J. Physiol.* 595, 2365–2396. doi: 10.1113/JP273259

Fink, M., and Noble, D. (2009). Markov models for ion channels: versatility versus identifiability and speed. *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci.* 367, 2161–2179. doi: 10.1098/rsta.2008.0301

Fritzson, P., Aronsson, P., Lundvall, H., Nyström, K., Pop, A., Saldamli, L., et al. (2005). "The OpenModelica modeling, simulation, and development environment," in *Proceedings of the 46th Conference on Simulation and Modelling of the Scandinavian Simulation Society* (Trondheim).

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series*. Reading, MA: Addison-Wesley.

Garny, A., and Hunter, P. J. (2015). OpenCOR: a modular and interoperable approach to computational biology. *Front. Physiol.* 6:26. doi: 10.3389/fphys.2015.00026

Gerstner, W., Kistler, W. M., Naud, R., and Paninski, L. (2014). *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge: Cambridge University Press.

Goldman, M. S., Golowasch, J., Marder, E., and Abbott, L. F. (2001). Global structure, robustness, and modulation of neuronal models. *J. Neurosci.* 21, 5229–5238. doi: 10.1523/JNEUROSCI.21-14-05229.2001

Hellerstein, J. L., Gu, S., Choi, K., and Sauro, H. M. (2019). Recent advances in biomedical simulations: a manifesto for model engineering. *F1000Research* 8:261. doi: 10.12688/f1000research.15997.1

Hines, M. L., and Carnevale, N. T. (1997). The NEURON simulation environment. *Neural Comput.* 9, 1179–1209. doi: 10.1162/neco.1997.9.6.1179

Hodgkin, A. L., and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol.* 117, 500–544. doi: 10.1113/jphysiol.1952.sp004764

Hoops, S., Sahle, S., Gauges, R., Lee, C., Pahle, J., Simus, N., et al. (2006). COPASI–a COmplex PAthway SImulator. *Bioinformatics* 22, 3067–3074. doi: 10.1093/bioinformatics/btl485

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., et al. (2003). The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics* 19, 524–531. doi: 10.1093/bioinformatics/btg015

Huguenard, J. R., and McCormick, D. A. (1992). Simulation of the currents involved in rhythmic oscillations in thalamic relay neurons. *J. Neurophysiol.* 68, 1373–1383. doi: 10.1152/jn.1992.68.4.1373

Hunt, A., and Thomas, D. (2000). *The Pragmatic Programmer: From Journeyman to Master*. Reading, MA: Addison-Wesley.

Inada, S., Hancox, J. C., Zhang, H., and Boyett, M. R. (2009). One-dimensional mathematical model of the atrioventricular node including atrio-nodal, nodal, and nodal-his cells. *Biophys. J.* 97, 2117–2127. doi: 10.1016/j.bpj.2009.06.056

Inkscape Developers (2020). *Inkscape-Draw Freely*. Available online at: https://inkscape.org/ (accessed August 20, 2020).

Jordan, J., Mørk, H., Vennemo, S. B., Terhorst, D., Peyser, A., Ippen, T., et al. (2020). *NEST*. Zenodo. Available online at: https://doi.org/10.5281/zenodo.2605422 (accessed August 20, 2020).

Justus, N., Schölzel, C., Dominik, A., and Letschert, T. (2017). "Mo|E–a communication service between Modelica compilers and text editors," in *Proceedings of the 12th International Modelica Conference* (Prague), 815–822. doi: 10.3384/ecp17132815

Le Novère, N. (2020). *BioModels: Hodgkin-Huxley Squid-Axon 1952*. Available online at: https://www.ebi.ac.uk/biomodels/BIOMD0000000020 (accessed August 20, 2020).

Lopez, C. F., Muhlich, J. L., Bachman, J. A., and Sorger, P. K. (2013). Programming biological models in Python using PySB. *Mol. Syst. Biol.* 9:646. doi: 10.1038/msb.2013.1

Maplesoft (2020). *MapleSim–Advanced System-Level Modeling and Simulation*. Available online at: https://www.maplesoft.com/products/maplesim/ (accessed August 20, 2020).

Matsuoka, S., and Hilgemann, D. W. (1992). Steady-state and dynamic properties of cardiac sodium-calcium exchange: ion and voltage dependencies of the transport cycle. *J. Gen. Physiol.* 100, 963–1001. doi: 10.1085/jgp.100.6.963

Mattsson, S. E., and Elmqvist, H. (1997). "Modelica–an international effort to design the next generation modeling language," in *7th IFAC Symposium on Computer Aided Control Systems Design, CACSD'97* (Gent), Vol. 30, 151–155. doi: 10.1016/S1474-6670(17)43628-7

Moore, J. W., and Cole, K. S. (1960). Resting and action potentials of the squid giant axon *in vivo*. *J. Gen. Physiol.* 43, 961–970. doi: 10.1085/jgp.43.5.961

Noble, D., Garny, A., and Noble, P. J. (2012). How the Hodgkin-Huxley equations inspired the cardiac physiome project. *J. Physiol.* 590, 2613–2628. doi: 10.1113/jphysiol.2011.224238

Olivier, B. G., Rohwer, J. M., and Hofmeyr, J.-H. S. (2005). Modelling cellular systems with PySCeS. *Bioinformatics* 21, 560–561. doi: 10.1093/bioinformatics/bti046

Payandeh, J. (2018). Progress in understanding slow inactivation speeds up. *J. Gene. Physiol.* 150, 1235–1238. doi: 10.1085/jgp.201812149

Piibeleht, M., Hatherly, M., and Ekre, F. (2020). *Documenter.jl*. Available online at: https://github.com/JuliaDocs/Documenter.jl (accessed August 20, 2020).

Rackauckas, C., Ma, Y., Widmann, D., Ranocha, H., Levien, E., Short, T., et al. (2020). *DifferentialEquations.jl Documentation*. Available online at: http://docs.juliadiffeq.org/latest/ (accessed August 20, 2020).

Schölzel, C. (2020). *THM-MoTE/ModelicaScriptingTools.jl: V1.1.0-alpha.1*. Zenodo.

Schölzel, C., Blesius, V., Ernst, G., and Dominik, A. (2020). The impact of mathematical modeling languages on model quality in systems biology: a software engineering perspective. *bioRxiv*. doi: 10.1101/2019.12.16.875260

Sweller, J. (2019). "Cognitive load theory," in *Advances in Cognitive Load Theory: Rethinking Teaching, 1st Edn*., eds S. Tindall-Ford, S. Agostinho, and J. Sweller (Abingdon: Routledge), 1–11. doi: 10.4324/9780429283895-1

The MathWorks, Inc. (2020). *Simscape*. Available online at: https://www.mathworks.com/products/simscape.html (accessed August 20, 2020).

Voit, E. O. (2013). *A First Course in Systems Biology, 1st Edn*. New York, NY: Garland Science.

Wimalaratne, S. M., Halstead, M. D. B., Lloyd, C. M., Cooling, M. T., Crampin, E. J., and Nielsen, P. F. (2009). Facilitating modularity and reuse: guidelines for structuring CellML 1.1 models by isolating common biophysical concepts. *Exp. Physiol.* 94, 472–485. doi: 10.1113/expphysiol.2008.045161

Winslow, R. L., Rice, J., Jafri, S., Marbán, E., and O'Rourke, B. (1999). Mechanisms of altered excitation-contraction coupling in canine tachycardia-induced heart failure, II: model studies. *Circ. Res.* 84, 571–586. doi: 10.1161/01.RES.84.5.571

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Copyright © 2020 Schölzel, Blesius, Ernst and Dominik. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) and the copyright owner(s) are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

8.3. Countering reproducibility issues in mathematical models with software engineering techniques: A case study using a one-dimensional mathematical model of the atrioventricular node

Title	Countering reproducibility issues in mathematical models with software engineering techniques: A case study using a one-dimensional mathematical model of the atrioventricular node
Authors	C. Schölzel, V. Blesius, G. Ernst, A. Goesmann, and A. Dominik
Publication type	Journal article
Journal	PLOS ONE
Publisher	Public Library of Science
Year	2021
Volume	16
Issue	7
Article number	e0254749
DOI	10.1371/journal.pone.0254749

Contributions by dissertation author (using the Contributor Roles Taxonomy):

Conceptualization	Conceived project, formulated research goals
Data curation	Maintained model code on GitHub, Zenodo, and BioModels
Formal analysis	Analyzed and modularized model equations
Investigation	Performed all experiments, literature research, and other evidence collection
Methodology	Applied SE techniques to model code
Software	Implemented the InaMo model and the MoST.jl
Validation	Implemented unit tests and CI/CD scripts for InaMo model
Visualization	Created all figures and tables
Writing — original draft	Wrote initial draft
Writing — review & editing	Revised manuscript due to feedback of other authors and reviewers

PLOS ONE

RESEARCH ARTICLE

Countering reproducibility issues in mathematical models with software engineering techniques: A case study using a one-dimensional mathematical model of the atrioventricular node

Christopher Schölzel[1,4]*, **Valeria Blesius**[1,4], **Gernot Ernst**[2,3], **Alexander Goesmann**[4], **Andreas Dominik**[1]

1 Technische Hochschule Mittelhessen—THM University of Applied Sciences, Giessen, Germany, **2** Vestre Viken Hospital Trust, Kongsberg, Norway, **3** University of Oslo, Oslo, Norway, **4** Justus Liebig University Giessen, Giessen, Germany

* christopher.schoelzel@mni.thm.de

Check for updates

OPEN ACCESS

Citation: Schölzel C, Blesius V, Ernst G, Goesmann A, Dominik A (2021) Countering reproducibility issues in mathematical models with software engineering techniques: A case study using a one-dimensional mathematical model of the atrioventricular node. PLoS ONE 16(7): e0254749. https://doi.org/10.1371/journal.pone.0254749

Editor: Roger A. Bannister, University of Maryland School of Medicine, UNITED STATES

Received: March 11, 2021

Accepted: July 3, 2021

Published: July 19, 2021

Peer Review History: PLOS recognizes the benefits of transparency in the peer review process; therefore, we enable the publication of all of the content of peer review and author responses alongside final, published articles. The editorial history of this article is available here: https://doi.org/10.1371/journal.pone.0254749

Copyright: © 2021 Schölzel et al. This is an open access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Data Availability Statement: All model files, simulation data, and scripts are available from Zenodo (https://doi.org/10.5281/ZENODO.4775302),

Abstract

One should assume that in silico experiments in systems biology are less susceptible to reproducibility issues than their wet-lab counterparts, because they are free from natural biological variations and their environment can be fully controlled. However, recent studies show that only half of the published mathematical models of biological systems can be reproduced without substantial effort. In this article we examine the potential causes for failed or cumbersome reproductions in a case study of a one-dimensional mathematical model of the atrioventricular node, which took us four months to reproduce. The model demonstrates that even otherwise rigorous studies can be hard to reproduce due to missing information, errors in equations and parameters, a lack in available data files, non-executable code, missing or incomplete experiment protocols, and missing rationales behind equations. Many of these issues seem similar to problems that have been solved in software engineering using techniques such as unit testing, regression tests, continuous integration, version control, archival services, and a thorough modular design with extensive documentation. Applying these techniques, we reimplement the examined model using the modeling language Modelica. The resulting workflow is independent of the model and can be translated to SBML, CellML, and other languages. It guarantees methods reproducibility by executing automated tests in a virtual machine on a server that is physically separated from the development environment. Additionally, it facilitates results reproducibility, because the model is more understandable and because the complete model code, experiment protocols, and simulation data are published and can be accessed in the exact version that was used in this article. We found the additional design and documentation effort well justified, even just considering the immediate benefits during development such as easier and faster debugging, increased understandability of equations, and a reduced requirement for looking up details from the literature.

BioModels (https://www.ebi.ac.uk/biomodels/MODEL2102090002), and GitHub (https://github.com/CSchoel/inamo).

Funding: The author(s) received no specific funding for this work.

Competing interests: The authors have declared that no competing interests exist.

1 Introduction

Mathematical modeling in systems biology, along with many other fields, is facing a reproducibility crisis [1, 2]. According to Stodden *et al.* [3], only an estimated 26 Curators of the BioModels database recently found that of 455 ordinary differential equation (ODE) models, only 51 As a single extreme case, Topalidou *et al.* [4] reported requiring three months to reproduce a neuroscientific model of the basal ganglia. The situation is similar to the reproducibility issues in wet-lab experiments, but it is less understandable, since *in silico* experiments only involving mathematical models are inherently free of the biological variations that complicate their wet-lab counterparts.

When talking about reproducibility, it is important to clearly define this term [5]. We follow the terminology of Goodman *et al.* [6], with the following modeling-specific adaptations: *Methods reproducibility* is achieved if the same code can be used with the same simulation tools and settings to produce the same results as the original study. *Results reproducibility* is achieved if the model can be rebuilt in a different language, with a different architectural structure, or simulated with different simulation tools using the same experiment protocol to achieve results that closely match those of the original study. *Inferential reproducibility* does not concern the reproduction of simulation data, but the reproduction of the conclusions drawn from the analysis of that data and the properties of the model. For the most part of this article we will not talk about inferential reproducibility, as our focus lies on model design and not on biological findings.

A lack in methods and results reproducibility can have direct consequences for the usefulness of a model. One example of this is the one-dimensional mathematical model of the atrioventricular (AV) node by Inada *et al.* [7]. It has been labeled as "ground-breaking" [8], because it was the first detailed model of the AV node, and it is still the only AV node model among over 600 models in the Physiome Model Repository [9]. We chose it for our research, because it is able to simulate many important phenomena of the cardiac conduction system including AV nodal reentry while still remaining manageable in its own complexity. Moreover, the article comes with a supplement that contains not only simulation results of the full model but also a set of figures that show the characteristics of the individual ion channel and ion pump models with up to eight individual plots for a single channel. Despite these indicators for a high quality article, the methods of Inada *et al.* are unfortunately not reproducible as there is no executable code available that can produce the results of the original study and reproducing the results with a reimplementation in another language took us more than four months. It seems intuitive that such difficulties in reproducibility may lead to fewer reproduction attempts and therefore less scientific impact. The issues we encountered with the Inada model may have prevented its widespread application, and thus, to some extent, hindered scientific progress in cardiovascular modeling. Given the number of such cases, we believe that it is unlikely that they arise out of a lack of scientific rigor. In contrast, it seems that the inherent complexity of such models inevitably opens the door to human error and that new tools and workflows are required to manage this complexity.

Researchers have already proposed several approaches to increase reproducibility in mathematical modeling. The most pressing and obvious suggestion is to publish the full simulation code, including executable scripts that produce the simulation results and plots that appear in the corresponding article [1–3, 10–15]. Many also advocate the use of literate programming in the form of electronic notebooks that mix textual descriptions and code as publication format [1, 4, 10–13, 16]. However, Medley *et al.* [16] also note that electronic notebooks can be too rigid for the creation of large and complex models and pose some difficulties for version control. Along with the code, data used for plots in the article should also be published, including

the simulation output of the published model [1–3, 10, 14]. Both data and code could be stored in specialized model databases that allow model discovery via semantic information [1, 2, 12]. Workflow systems such as Galaxy [17] or KNIME [18] can be used to publish simulation procedures in a format that ensures methods reproducibility through the use of standardized components [1, 10, 16].

Other suggestions concern the role that academic journals can play in ensuring and promoting reproducibility. Publication checklists [1, 11] or a "seal of approval" [1, 11, 14, 15] could provide missing incentives for researchers to put more focus on all forms of reproducibility. A few journals even already experiment with publication workflows that aim to guarantee methods reproducibility of published models. *PLOS Computational Biology* recently started a promising pilot project in collaboration with the Center for Reproducible Biomedical Modeling [19], which extends the peer review with an additional step in which reviewers specifically evaluate the methods reproducibility of the computational modeling aspects of a submission [15]. The journal *Physiome* takes a similar approach by publishing articles that demonstrate the consistency and reproducibility of mathematical models already described in other publications. Here, too, the actual methods reproducibility is assessed by independent *Physiome* curators.

Apart from these suggestions, which are specific for mathematical modeling and/or systems biology, researchers also advocate for the application of common best practices from software engineering. This includes structured documentation [1, 3, 11, 12], version control [10–13, 16], unit testing [2, 12, 13, 16], the use of open standards [1, 2, 12, 14], human-readable code with style guides [2, 13], modularity [11, 13], object-orientation [12, 13], the use of virtual machine specifications [1, 10, 11], and the long-term archival of code [2].

Borrowing software engineering concepts for improving the methods and results reproducibility of mathematical models seems natural, since these models are, after all, software. As models grow in size and complexity towards examples such as the *Mycoplasma genitalium* whole-cell model by Karr *et al.* [20] or the central metabolism of *E. coli* by Millard *et al.* [21], they face the same kind of issues that software faces when it evolves from a single script of a few lines of code to a complex system with thousands or millions of lines of code. While these issues started to appear only fairly recently in systems biology, they are known for decades in software engineering and efficient solutions have been and are still being developed. Hellerstein *et al.* [13], therefore, argue that modelers should rethink their work as "model engineering" by applying software engineering techniques to the domain of mathematical modeling.

In our attempt to make the Inada model more reproducible, we build on the ideas of model engineering and our own previous work. Most importantly, we found that languages that are modular, descriptive, human-readable, open, graphical, and hybrid (MoDROGH) can help to increase both methods and results reproducibilty as well as reusability, extensibility and understandability [22]. We verified the effectiveness of the consistent use of these characteristics by creating and analyzing a modular version of the Hodgkin-Huxley (HH) model of the squid giant axon [23]. Since the Inada model mainly consists of HH-type ion channels, it is highly likely that this model can also benefit from our design approach. While implementing the HH model, we also developed a workflow with unit tests that are run automatically on an online server every time the code is updated. This concept is called continuous integration (CI) in software engineering and was developed precisely to ensure that software can be installed and run in an environment that is completely separate from the development environment [24]. It is already used, for example, in the bioinformatics framework NF-CORE [25], and in the OpenWorm project, which aims to model *Caenorhabditis elegans* [26]. We expect that this, combined with a model architecture that follows the MoDROGH guidelines, and regression

tests, which ensure that changes to a model do not affect the simulation output, can solve many if not all the reproducibility issues present in the Inada model.

We believe that results from this case study will be applicable to a large set of systems biology models for several reasons: The Inada model is a good example of the range of difficulties and pitfalls one faces when trying to ensure the reproducibility of methods and results of an *in silico* study. Inada *et al.* certainly tried to make their work as transparent as possible. Yet still, the model exhibits all the common reproducibility issues identified by the BioModels reproducibility study [27]—"recoverable" issues like sign errors, missing equations, order of magnitude, and unit errors, as well as "non-recoverable" issues such as missing parameter values, missing initial values and errors in equations. It is also a representative example for challenges in reproducing models in a multi-scale context. On the one hand, the full one-dimensional model of the AV node is in itself a multi-scale model, since it covers cell and organ scales with observed effects ranging from milliseconds to seconds. On the other hand, all 6 published reproductions of the results of Inada *et al.* include the single AV cell model in a larger multi-scale context, be it a 3D heart model [28–30], the cardiac conduction system [31], or a one-dimensional ring model of the sinoatrial (SA) node [32, 33]. Finally, none of the techniques and guidelines that we apply are specific to the Inada model or electrophysiological models in general. The MoDROGH criteria were already applied to an organ-level model of the human baroreflex [22] and both CI and regression tests are concepts borrowed from software engineering, which are applicable to any piece of software. This should also allow to transfer our results to other MoDROGH languages like the Systems Biology Markup Language (SBML) [34] or CellML [35].

We therefore address the following research questions:

RQ1 What are the factors that hinder the reproduction of the methods and results of the Inada model?

RQ2 Are software engineering techniques (in particular a MoDROGH design, regression tests, and CI) suited to overcome the issues identified in RQ1?

We will answer these questions by first giving an overview of the Inada model, the resources available for reproduction, and our design philosophy for the reimplementation in Section 2. We then describe all reproducibility issues along with our solutions in Section 3. In Section 4 we discuss the answers to our research questions as well as the general applicability of the techniques that we presented and their limitations. Finally, we draw our conclusion in Section 5.

2 Materials and methods

2.1 The Inada model

The one-dimensional mathematical model of the atrioventricular node (AV node) by Inada *et al.* [7], which we simply call the Inada model in the following, consists of a one-dimensional chain of different cell types: For the sinoatrial node cells and the atrial cells, preexisting models are used, but for the atrionodal (AN), nodal (N), and nodal-His (NH) cells, the authors developed own formulations. In total these three new cell types are composed of eight ion channels, two ionic pumps, and four compartments with variable Ca^{2+} concentrations:

- ion channels
  - background channel (I_b)
  - L-type calcium channel ($I_{Ca,L}$)

  - rapid delayed rectifier channel ($I_{K,r}$)
  - inward rectifier channel ($I_{K,1}$)
  - sodium channel (I_{Na})
  - transient outward channel (I_{to})
  - hyperpolarization-activated channel (I_f)
  - sustained outward channel (I_{st})
- ion pumps
  - sodium calcium exchanger (I_{NaCa})
  - sodium potassium pump (I_p)
- compartments containing variable Ca^{2+} concentrations
  - cytoplasm ($[Ca^{2+}]_i$)
  - junctional sarcoplasmic reticulum (JSR) ($[Ca^{2+}]_{jsr}$)
  - network sarcoplasmic reticulum (NSR) ($[Ca^{2+}]_{nsr}$)
  - "fuzzy" subspace ($[Ca^{2+}]_{sub}$), which is the "functionally restricted intracellular space accessible to the Na^+/Ca^{2+} exchanger as well as to the L-type Ca^{2+} channel and the Ca^{2+}-gated Ca^{2+} channel in the SR" [7, 36]
- concentrations assumed to be constant
  - extracellular calcium concentration ($[Ca^{2+}]_o$)
  - intra- and extracellular sodium concentrations ($[Na^+]_i$, $[Na^+]_o$)
  - intra- and extracellular potassium concentrations ($[K^+]_i$, $[K^+]_o$)

2.2 Available material

In the first stages of our reimplementation of the Inada model, we relied only on publicly available data. This included the article by Inada *et al.*, the supplementary data for this article in PDF format, and the CellML version of the model, which was created by Lloyd [37] and published in the Physiome Model Repository [9]. The CellML implementation contained code that was not in the paper and did not produce simulation output that resembled any of the plots in the original article. We therefore used it as a reference, but did not rely on its correctness. We supervised two Bachelor's theses that reimplemented the CellML model in Octave and in Modelica. Both projects were able to reproduce some but not all the reference plots in [7]. Before we implemented the current version, we therefore attempted to obtain the original C++ implementation of Inada *et al.* by contacting the authors themselves and the editors of the Biophysical Journal. Unfortunately, we did not receive an answer and our attempt to contact Lloyd for comments on the CellML model was equally unsuccessful. In a second attempt at a later stage of the development, we reached out to the production team of the Biophysical Journal and to the general help address of the Physiome Model Repository. The former finally allowed us to obtain the C++ code and the latter clarified some questions about the CellML implementation and improved our confidence in this code.

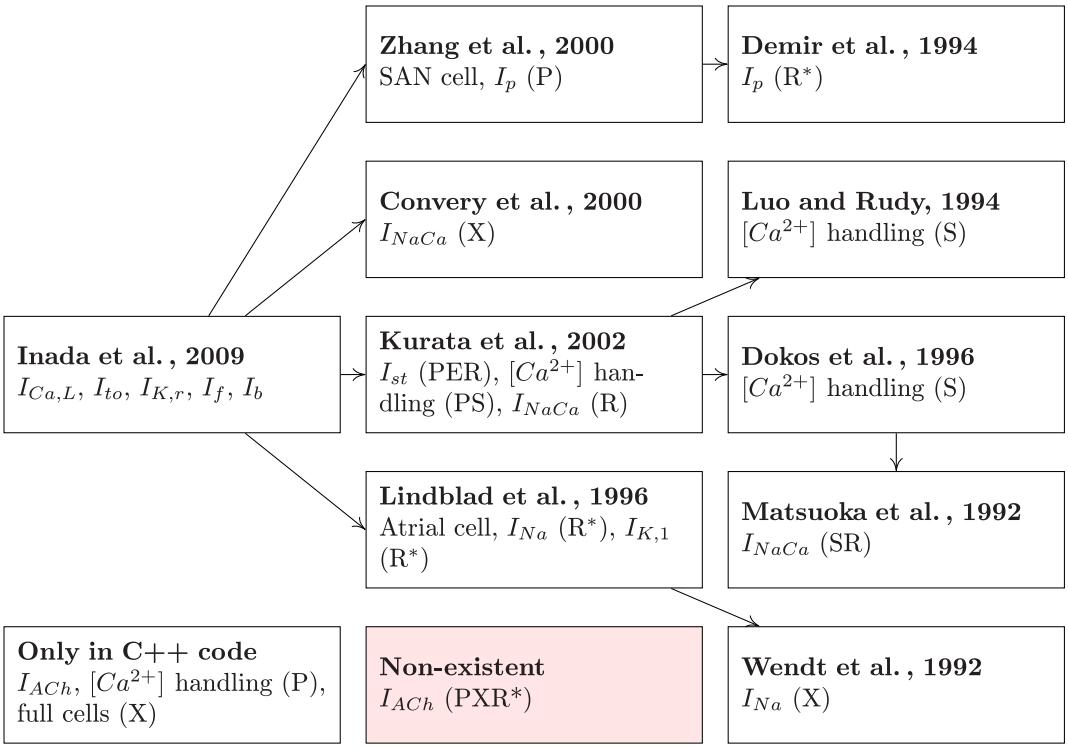


Fig 1. Tree of references that we traversed to obtain all relevant information to understand and test the model. Arrows between nodes indicate that the article at the beginning of the arrow cites the article at the end of the arrow. Each node contains a list of model parts that could only be reproduced by using this reference. This is further specified by distinguishing if the reference was needed to determine parameter values (P), correct errors (E), untangle equation semantics for modularization (S), obtain an additional (R) or the only available (R*) reference plot, or to reproduce the experiment protocol (X).

https://doi.org/10.1371/journal.pone.0254749.g001

2.3 Implementation process

Due to the discrepancies between the article, the CellML code, and the C++ code, we decided to implement the components of the Inada model one by one, testing each component before moving on to the next. In order to obtain reference plots, experiment protocols and parameter values as well as to understand the equations deeply enough to bring them into a modular structure, we needed to examine a total of nine additional articles that were cited directly or indirectly in the Inada model. The full tree of references can be seen in Fig 1. Additionally, an estimation of the time spent on research, implementation, testing, bug fixing, and refactoring and documentation can be seen in Fig 1 in S1 Text.

2.4 Model design

Our design philosophy was based on our own guidelines established for using the MoDROGH criteria of suitable modeling languages for systems biology, which can improve the methods and results reproducibility, understandability, reusability, and extensibility of models [22, 23]. In short, this includes the following design goals: The model should follow a modular design with small self-contained modules with clearly defined, minimal interfaces. Each module should only represent a single physiological compartment or effect. The code should be DRY (for "don't repeat yourself"), meaning that parts of the code that have similar structure are only implemented once and then reused at the respective position. Equations structure and variable names should convey their meaning, and should not be adjusted for brevity or

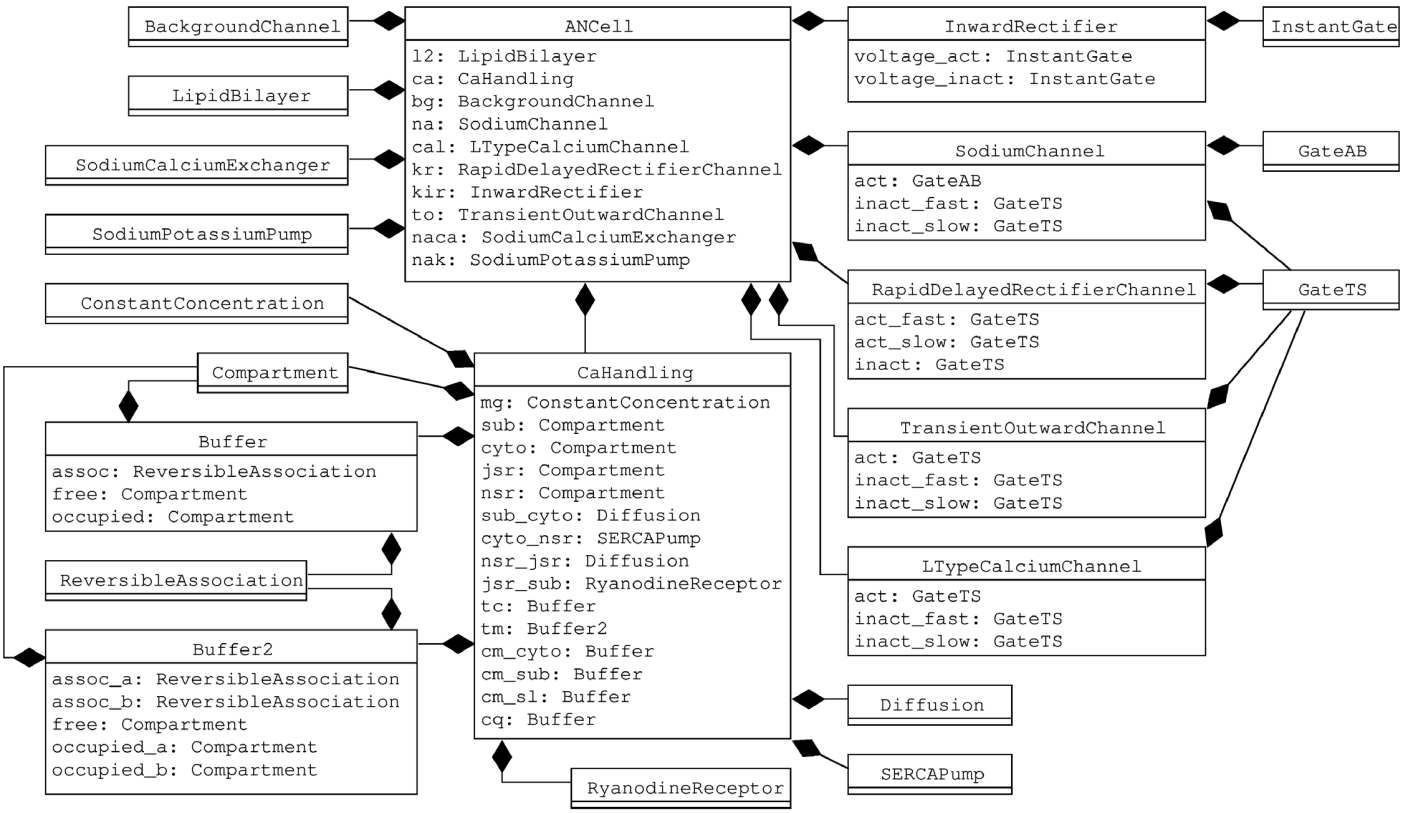


Fig 2. Hierarchical composition of AN cell model in InaMo version 1.4.1 as UML diagram. The composition arrow ——◆ indicates that the model at the beginning of the line is a part of the model at the end of the line, where the diamond shape is located.

https://doi.org/10.1371/journal.pone.0254749.g002

perceived ease of implementation. All variables and parameters should have International System of Units (SI) units and should be documented with at least a short sentence that explains what they represent. The model should also have a graphical representation that explains the modeled system using symbols that reflect the biological appearance or function of components. This representation should be adjusted manually for understandability, and it should be tied to the model equations to guarantee correctness and completeness. Additionally, the model code should be available in an online repository. In our case the repositories are GitHub (https://github.com/CSchoel/inamo), Zenodo [38], and BioModels (https://www.ebi.ac.uk/biomodels/MODEL2102090002).

We use the same basic model structure as in a previous work, where we tested the viability and benefits of building an electrophysiological model with these guidelines by implementing the Hodgkin-Huxley model [23]. The cell models consist of a model of the lipid bilayer, a number of ion channels with a common base class, and separate models for voltage or current clamp experiment protocols. The ion channel models again contain smaller modules that represent individual gating variables. All parts of the model are connected with a basic interface for components in an electrical circuit diagram with the convention that the positive pin resides on the outside of the cell while the negative pin is on the inside. An example of the full composition structure of the AN cell model can be found in Fig 2. Like in the HH model, we also used the modeling language Modelica [39], since it implements the MoDROGH criteria to a large extent and lends itself well to the application of software engineering techniques

because it is an established industry standard. However, similar results could also be achieved using, for example, SBML [34], CellML [35], or Antimony [40].

2.5 Software versions

For our implementation we used OpenModelica version 1.16.0 [41] as Modelica compiler and integrated development environment (IDE) and also version 0.7.2 of Mo—E [42] with the corresponding plugin for Atom version 1.49.0. For our test scripts we used Julia version 1.4.2 [43] with version 1.1.0-alpha.3 of our own library ModelicaScriptingTools.jl [44]. We keep the code under version control using Git [45] version 2.28.0 in a repository hosted on GitHub [46]. We also use GitHub Actions [47] to run our CI scripts. The plots were produced using Python 3.8.3 [48] with the plotting library matplotlib version 3.1.2 [49, 50]. Icons were created using Inkscape version 1.0 [51] with our extension MoNK version 0.2.0 that converts the Inkscape vector graphics to Modelica annotation format [52]. The CellML model was analyzed using OpenCOR version 0.6 [53].

3 Results

This section is structured according to the issues that hindered our reproduction of the results of the Inada model. For each issue we first explain the problem in detail and then show how it is solved in our Modelica reimplementation, which we will call InaMo in the following.

3.1 Missing equations and parameters

3.1.1 Problem description. The first and most obvious issue with results reproducibility of the Inada Model are parameter values and equations that are missing in the article, which are listed in Table 1. An example is the acetylcholine-sensitive potassium channel. The whole channel equations, as well as the influence of acetylcholine on I_f and $I_{Ca,L}$, only exist in the

Table 1. Missing information in the Inada model including all parameters, equations and starting values that cannot be found in the original article.

Component	Affected part	Recoverable from
I_{ACh}	all equations	C++ code
I_{ACh}	parameter $[ACh]_i$	not recoverable
I_f	ACh-sensitive term in equation	C++ code
$I_{Ca,L}$	ACh-sensitive term in equation	C++ code
$I_{Ca,L}$	parameter `ach_l`	not recoverable
$[Ca^{2+}]$ handling	equation for V_{cell} w.r.t. C_m	C++ code
$[Ca^{2+}]$ handling	parameter SL_{tot}	C++ code
$[Ca^{2+}]$ handling	all parameters but SL_{tot} and V_{cell}	[54]
I_{st}	parameter E_{st}	[54]
$I_{K,r}/I_{K,1}/I_{to}$	parameter E_K	calculated from $[K^+]_i$, $[K^+]_o$
I_p	parameters $K_{m,Na}$, $K_{m,K}$	[55]
I_p	parameter $\bar{I}_p$	called $I_{p,max}$ in [7, S15]
I_{Na}	parameter P_{Na}	called g_{Na} in [7, S15]
I_{to}	starting values r, q_{fast}, q_{slow}	called q, r_{fast}, r_{slow} in [7, S16]
I_{NaCa}	parameters K_x [1] and Q_y [2]	[56, 57]

[1] $x = ci, cni, 1ni, 2ni, 3ni, co, 1no, 2no, 3no$

[2] $y = ci, co, n$

https://doi.org/10.1371/journal.pone.0254749.t001

C++ code. Neither the equations nor the parameters are mentioned in the article, and we are not aware of any description in subsequent articles of the authors. The C++ code also does not give a value for $[ACh]_i$, which is set to zero in the CellML version. It is possible that this was a planned extension, which was never realized and not used for the plots in the article, but with the available material it is impossible to tell whether that is the case. Other parameters are missing in the article, but could be recovered from cited literature or the C++ code. There were also parameters that were hard to find due to naming confusions. One example that caused severe errors for us was that the value given for g_{Na}, the conductivity of I_{Na}, is actually the value for the permeability P_{Na} that is used to calculate g_{Na}. As a last minor piece of missing information, the article does not specify how the avoidable discontinuities in equations 1 and 14 of I_{Na} in table S3 should be handled.

3.1.2 Solution 1: Continuous integration. To ensure that such omissions do not hinder reproduction of simulation results, it is not enough to rely on human diligence. With a total of 85 parameter values, there is a statistical argument to be made about the expected percentage of errors that a single author or reviewer might be able to spot. Regardless of how the actual numbers would turn out, it does not seem reasonable to expect or demand 100.

Such a guarantee is only possible, if the complete code that is required to run the simulation on a different machine is published alongside with the model. Inada *et al*. did publish parts of their code, but not the full version, which left us with some open questions regarding the acetylcholine-sensitive potassium channel. In contrast, the CellML model is complete, but based on errors that we found in the code one must assume that simulations were only performed with the N cell model and not with the other two cell types.

For the new implementation, we therefore not only publish the full model definition but also the scripts that we used for simulation and plotting. To ensure that the published code is complete and does also work on other machines, we used the CI service GitHub Actions [47], which is free for public open source projects. For each update of the code, a build in a fresh virtual machine is started on the GitHub servers, which downloads the new release and runs the simulation script. The current build status can be indicated to users with a small badge in the repository, and if a build fails, the programmer is informed via e-mail. This mechanism guarantees that the repository contains everything that is required to perform simulations on a machine that is physically separated from the original development environment, i.e. it guarantees methods reproducibility. The build scripts for CI services such as GitHub Actions are easy to write and provide the additional benefit that they have to contain a full description of the development environment including installation scripts and non-standard software dependencies. The build script that we used for our implementation of the Inada model can be found in Listing 1.

Listing 1. CI script for InaMo version 1.4.3 using GitHub Actions. The script creates a virtual machine running the Ubuntu operating system, installs Julia, OpenModelica, the Modelica Standard Library, and required Julia packages, and runs the unit tests defined in the file `scripts/unittests.jl`. It runs automatically whenever a new commit is pushed to the main branch of the Git repository.

```
on:
  push:
    branches: [ main ]
    tags: 'v*'
  pull_request:
    branches: [ main ]
jobs:
  build:
    runs-on: ubuntu-latest
```

```
steps:
- uses: actions/checkout@v2
  with:
    submodules: true
- uses: julia-actions/setup-julia@v1
  with:
    version: 1.6
- uses: THM-MoTE/setup-openmodelica@v1
  with:
    version: 1.17.0
- name: Install Modelica standard library
  run: sudo apt-get install omlib-modelica-3.2.3
- name: Install Julia requirements
  run: |
    export PYTHON=""
    julia --project=. -e 'using Pkg; Pkg.instantiate()'
- name: Run unit tests
  run: julia --project=. scripts/unittests.jl
```

3.1.3 Solution 2: Version control. Even if the complete code of a model is published, an exact reproduction of methods might still fail, because of changes that have been added to the code after the model was published. It might even be the case that a figure in an article was created with a newer or older version of the code than other figures. One such uncertainty about code versions is the question if the current I_{ACh} was included and activated in the simulations performed by Inada *et al.* The C++ code gives some clues as it contains a list of major changes with the date of the change. According to this information, I_{ACh} was added on 11/04/2008, which is before the initial submission to Biophysical Journal on 27/02/2009. However, this is still not enough to be sure that I_{ACh} was used for simulations, because another change—a rescue effect for $I_{Ca,L}$—was added on 23/10/2008, but the current parameter values used in the published version clearly disable it. If the code was under version control and the history was published, it would be possible to answer this question at least with some confidence by tracking the changes through time.

We therefore publish our reimplementation on GitHub [46], which uses the version control software Git [45]. Additionally, we keep a human-readable log of major changes in a Markdown-formatted [58] text file called CHANGELOG.md in the repository. The simulation results in S2–S32 Figs are tagged with the actual version used for the simulation.

Version control also has several other benefits beyond understanding when, how, and why a model has been changed. Most prominently, it allows researchers to work on a model collaboratively and to merge changes made by different authors, which will become more important in systems biology as models grow in size and models by different groups have to be integrated into a single project. Additionally, version control facilitates debugging by allowing to effortlessly roll back changes to identify the exact edit that introduced an error in the code. Finally, changes that may have to be reverted, like the rescue effect for $I_{Ca,L}$, can be developed on separate branches, which can then either be abandoned or merged into the main branch, depending on whether the feature was deemed beneficial or not.

3.2 Errors in equations and parameters

Apart from missing information there are also errors both in equations and parameter values, which can be seen in Table 2. These are typical oversights including sign errors and order of magnitude errors related to unit conversion. An example of a sign error is the erroneous negative sign for Q_n in equation 5 of I_{NaCa} in table S10. Order of magnitude errors can, for example, be found in the parameters k_x and k_{b_x} where $x = f_{TC}, f_{TMC}, \ldots$ in the equations for $[Ca^{2+}]$

Table 2. Errors in the published equations of the Inada model including wrong signs, shifted floating points, and missing unit conversions.

Component	Affected part	Kind of error/correction
$[Ca^{2+}]$ handling	table S12, eq. 5	f_{CMi} and f_{TC} must have negative sign
I_{NaCa}	table S10, eq. 5	Q_n must have positive sign
I_{st}	table S8, eq. 2	second occurrence of V must be negative
$[Ca^{2+}]$ handling	parameters[1] k_x, k_{b_x}	must be multiplied by 1000 ($ms^{-1} \rightarrow s^{-1}$)
I_{st}	variables τ_{qa}, τ_{qi}	must be divided by 1000 (ms → s)
I_{to}	variable $\tau_{q_{fast}}$	constant 0.1266 must be 0.01266 instead

[1] $x = f_{CM}, f_{CQ}, f_{TC}, f_{TMC}, f_{TMM}$

https://doi.org/10.1371/journal.pone.0254749.t002

handling, which all have the unit ms^{-1} in [54] but need to be multiplied by 1000 since [7] uses seconds as unit of time.

As a second type of errors, there are inconsistencies between parameter values in the article and in the C++ and CellML implementations, which can be found in Table 3. Some of them, like the value used for P_{rel}, seem to be undocumented changes that do have a large qualitative effect on simulations. Others, like the value of 227700 instead of 222700 for $k_{f_{TMC}}$, seem to be simple typing errors and oversights. Of these, it is notable that in the CellML model, the value of C_m is switched between the NH and the N cell model, which should have stood out if the model was verified in simulations.

These inconsistencies are even more pronounced in the initial values. Due to the large number of initial values to keep track of, we do not show them here but only in Tables 1–3 in S1 Text. It seems as though the initial states were chosen to resemble the near-steady state achieved right before a pulse during a long-term simulation with a current pulse protocol. However, this information is missing in the article. We only found this pattern through trial and error and by a hint in the version history of the CellML model. Even with this knowledge, still large inconsistencies remain between the article, the C++ model, and the CellML model with some open questions. For example, we suspect that there is an order of magnitude error in the reported initial value for the parameter f_{TC} in the N cell model in the article. While these errors do affect simulations for up to 20 seconds, all variables that need an initial value do

Table 3. Parameter values that differ between the published article and the C++ and CellML implementations.

Parameter	Cell type	Unit	Value in article	Value in C++	Value in CellML
P_{rel}	AN, NH	s^{-1}	5000	1805.6	1805.6
P_{rel}	N	s^{-1}	5000	1500.0	1500.0
V_{cell}	AN, NH	m^3	$3.500 \cdot 10^{-15}$	$4.398 \cdot 10^{-15}$	$4.400 \cdot 10^{-15}$
V_{cell}	N	m^3	$3.500 \cdot 10^{-15}$	$3.189 \cdot 10^{-15}$	$3.190 \cdot 10^{-15}$
C_m	AN	F	$4.0 \cdot 10^{-11}$	$4.0 \cdot 10^{-11}$	$4.0 \cdot 10^{-11}$
C_m	N	F	$2.9 \cdot 10^{-11}$	$2.9 \cdot 10^{-11}$	$4.0 \cdot 10^{-11}$
C_m	NH	F	$4.0 \cdot 10^{-11}$	$4.0 \cdot 10^{-11}$	$2.9 \cdot 10^{-11}$
$k_{f_{TC}}$	all	$\frac{1}{\text{mM} \cdot \text{s}}$	534	534	543
$k_{f_{TMC}}$	all	$\frac{1}{\text{mM} \cdot \text{s}}$	227700	222700	227700
E_{st}	N	mV	37.4	37.4	-37.4

For C_m the reference article is [7], for all other parameters it is [54].

https://doi.org/10.1371/journal.pone.0254749.t003

gravitate towards a steady state, meaning that these issues should not affect results or inferential reproduction.

3.2.1 Solution 1: Testing. Again, it becomes clear that neither authors nor reviewers can be expected to find every single oversight within well over a hundred equations and parameters. Like with missing information, automated tests are the only way to reliably ensure that a piece of code of the size of the Inada model is free of errors. These tests can be run in a CI environment, as described above, which means that each version of a model will be automatically evaluated for errors that may have been introduced accidentally. For InaMo, we use three kinds of automated tests, which are common in software engineering: unit tests, integration tests and regression tests.

Unit tests are fine-grained tests for small software modules. In order to create unit tests, these modules must be independent of the rest of the code. They are designed to pin down errors to a single module and therefore increase the confidence in the correctness of these modules. In mathematical modeling this is important, because often researchers do not want to reproduce the results of the full model but only a part of it. This becomes considerably easier if there is an existing test case for the part that should be reused. InaMo contains unit tests for each individual current and gating variable in the model as well as for the diffusion reactions, ryanodine receptor, SERCA pump, and buffer components in the $[Ca^{2+}]$ handling. Each of the experiment models used for this test import the exact same components used for the full cell model in an isolated environment where all external variables that influence the behavior of the component are carefully controlled. Almost all of these experiments correspond to plots in the original article or cited references (see Table 4). The only exception are the individual components of the $[Ca^{2+}]$ handling, because neither Inada *et al.* [7] nor Kurata *et al.* [54] provide plots at this level of detail.

Moving to the next category, integration tests help to ensure that there is no error in the connection between modules when they are combined to a larger system. They work much in the same way as unit tests, but are applied to the whole software instead of only individual modules. This helps to spot errors that only emerge from the interaction between the individual modules. In InaMo, there are integration tests for each cell type (AN, N, and NH cells) as well as both for constant and for varying $[Ca^{2+}]_i$. This prevents issues like in the CellML model, where probably only N cells were tested.

The third category of tests are regression tests, which ensure that the output produced by a piece of code does not change accidentally. They are typically used, when the output is large and has a complex structure that is otherwise hard to incorporate in unit tests. In mathematical modeling, these kinds of tests can serve three purposes. First, they ensure that changes to a part of a model do not have unforeseen consequences in other parts of the model. Second, they highlight these changes during the development process, which increases the probability that plots and formulas in the corresponding article will be changed accordingly. Third, the mere fact that regression tests require keeping reference output data in the repository helps to preserve results reproducibility in the future, when the software that produces the simulation results may not be available anymore. The repository for InaMo contains a separate sub-repository with reference data for all models used in unit and integration tests and the GitHub Actions script performs regression tests for them. If changes are found, the plotting script can be configured to output additional plots from the reference data so that these changes can be inspected visually. Since Modelica is a human-readable language and the reference data is stored in the human-readable CSV format, researchers in the far future will only need a simple text editor to reproduce the results in this article.

3.2.2 Solution 2: Unit consistency checks. As another category of tests, which is specific to mathematical models, automated unit consistency checks can help to avoid order of

Table 4. Summary of individual experiments that were reproduced with InaMo.

Part	Figure	Exact	Issues/Required changes
$I_{Ca,L}$	[7, S1]	(✓)	T_{hold} = 5s, $V_{\text{hold}} \leftarrow$ -70 mV, NH parameters, S1H: reference timescale must be multiplied by 0.75
I_{to}	[7, S2]	✖	T_{hold} = 20s, NH parameters, S2E: current too high, S2D: higher minimum in reference
$I_{K,r}$	[7, S3]	✖	T_{hold} = 5s, S3C: reference shifted towards higher voltages
I_f	[7, S4]	✖	T_{hold} = 20s, S4C: qualitative differences
I_{st}	[7, S5]	✖	T_{hold} = 15s, $g_{st} \leftarrow$ 0.27 nS, S5C: qualitative differences
I_{st}	[54, 4]	✓	T_{hold} = 15s
I_{NaCa}	[7, S6]	(✓)	$[Ca^{2+}]_o \leftarrow$2.5 mM, $[Ca^{2+}]_{sub}$ = 0.00015 mM
I_{NaCa}	[54, 17]	✓	k_{NaCa} given as $\frac{\text{pA}}{\text{pF}} \Rightarrow$ multiply by C_m
I_{NaCa}	[57, 19]	(✓)	0.25 nA $< k_{NaCa} <$ 1 nA chosen to fit plots
I_{Na}	[65, 2]	✓	T_{hold} = 2 s, T_{pulse} = 50 ms, $na_p \leftarrow 2.1 \frac{\text{pl}}{\text{s}}$
I_b	-		trivial, no need for test
I_{ACh}	-		no description or plot available
$I_{K,1}$	[65, 2]	✓	
I_p	[64, 12]	✖	reference mixes I_p with background currents
$[Ca^{2+}]_i$	[7, S7]	✖	only as part of full cell simulation (see row below)
Cell	[7, S7]	✖	T_{hold} = 300 ms, T_{pulse} = 1 ms, I_{pulse} = −1.2 nA (AN), I_{pulse} = −0.95 nA (AN), AN, NH: resting potential too high and action potential slightly too short, N: $[Ca^{2+}]_i$ too high

The table shows the part of the model that is tested with the experiment, a reference to the original figure, the information whether the plot could be reproduced exactly, and a list of issues and changes that were required to obtain a good agreement with the original plot. For a visual comparison of plots, see Figs 2–33 in S1 Text. For the exactness, ✓ means a near perfect reproduction was possible with minimal adjustments, (✓) means that significant changes or manual parameter tuning were required, and ✖ means that even after adjustments only qualitative agreement was achieved while some visible differences remain. In the last column, an equals sign (=) means that a parameter value was not given in the original article and had to be determined by us, whereas an arrow (←) means that the parameter value was given, but had to be changed. The entry "NH parameters" means that we had to use the parameters given for the NH cell model, while Inada *et al.* report that they used parameter settings of the AN cell model.

https://doi.org/10.1371/journal.pone.0254749.t004

magnitude errors. In declarative languages like Modelica, SBML, or CellML, variables can be annotated with unit definitions and software tools can track the conversion between units in an equation with symbolic mathematics. A unit consistency check then produces an error or a warning if an equation is found, where the right-hand side has a different unit than the left-hand side. In InaMo, variables have unit definitions according to the SI wherever possible and the tests in the CI script contain consistency checks, which are performed when loading the individual models.

3.2.3 Solution 3: Modular model structure using object orientation. Unit tests require model components to be defined in an independent modular structure. It must be possible to run a simulation using only a single component and a minimal experiment setup surrounding it. At the same time, the code of that component must be exactly the same code in the same file that is used in the full cell model, because otherwise the unit test cannot make assertions about the correctness of the full model.

With InaMo, we therefore consequently follow a modular design structure with minimal interfaces between components. Each component is defined in its own file, which is imported both in the unit test of that component and in the full cell model. The full hierarchical composition of the AN cell model can be seen in Fig 2. An example that shows how the component `SodiumChannel` is used both in the unit test `SodiumChannelIV` and in the full cell model `ANCell` is presented in Listing 2.

Listing 2. Example for construction of unit tests and full cell model out of the same components. The component `SodiumChannel` is used both in `SodiumChannelIV`, which defines a voltage clamp experiment to test the current-voltage relationship of I_{Na} that is used as a unit test, and in `ANCellBase`, which is the base for the full AN cell model. In both cases, an instance of `SodiumChannel` with the name `na` is defined and then connected to other components in the model using `connect()` equations. Additionally, in `ANCellBase`, the initial values of gating variables are adjusted. The ellipses (. . .) denote code that is not shown including inheritance from base classes, additional components and connections, and graphical annotations.

```
model SodiumChannelIV "IV relationship of I_Na (Lindblad 1996,
Fig. 2b)"
  ...
  InaMo.Currents.Atrioventricular.SodiumChannel na annotation(...);
  ...
equation
  connect(vc.p, na.p) annotation(...);
  connect(vc.n, na.n) annotation(...);
  ...
end SodiumChannelIV;
model ANCellBase "base model for AN cell type"
  ...
  InaMo.Currents.Atrioventricular.SodiumChannel na(
    act.n.start = 0.01227,
    inact_slow.n.start = 0.6162,
    inact_fast.n.start = 0.7170
  ) annotation(...);
  ...
equation
  ...
  connect(na.p, p) annotation(...);
  connect(na.n, n) annotation(...);
  ...
end ANCellBase;
```

A modular, object-oriented design is not only beneficial because it allows defining unit tests, but it also in itself can help to reduce possible sources of errors by reducing redundancy in the code. For example, the CellML implementation of the Inada model is split into three separate files for the AN, N and NH cells. This means that every error that is found in the model has to be corrected in all three files, leaving the opportunity open for additional oversights. Conversely, the C++ implementation handles all model types in a single file, but this also creates a problem. The code that sets parameter values uses conditional branches based on which cell type should be simulated. Because of the monolithic structure, values need to be defined for each current, even for those currents which are not present in the selected cell type. This led to an error that the parameter E_{st} for the current I_{st} had a wrong sign in the AN and NH cell setup. For the C++ implementation, this is no issue, since I_{st} is only used in the N cell model, where the sign was corrected. However, it appears that this error was accidentally transmitted to the CellML model, where all three cell types have the wrong sign for E_{st}.

In InaMo, we follow the DRY (don't repeat yourself) principle of software engineering: Each component and parameter is defined exactly once in the code, reusing common structures as much as possible to reduce redundancies. In an object-oriented language like Modelica, this can be achieved in two ways: First, components can be instantiated, which means that their code is imported into a model under a chosen name. Two instances of a component can have different parameter settings, allowing, for example, to use the same component `GateTS` both for the activation and inactivation gate of an ion channel as shown in Listing 3. Second, models can also inherit components, parameters, and equations from common base classes. This is similar to composition via instantiation, but has the added benefit that the inherited parts directly become a part of the model without the need to access them through a component name. For example, this is very useful for the ion channels in the Inada model, which almost all follow an electric analog. The base class `IonChannelElectric` defines the basic behavior of an ion channel as voltage-dependent resistor with an attached voltage source and can be reused for I_b, I_{ACh}, I_f, $I_{K,1}$, $I_{Ca,L}$, $I_{K,r}$, I_{st}, and I_{to}. This is shown in Listing 4.

3.2.4 Solution 4: Specialized testing library for Modelica models. There are currently not many solutions for automated tests that are specifically designed for mathematical models. To facilitate the creation of such tests as much as possible, new tools are required. One promising approach is to use libraries that can run simulations from within general purpose programming languages such as Python or Julia and to then use the existing capabilities for automated testing that exist in these languages.

We therefore developed the Julia library ModelicaScriptingTools.jl (MoST.jl) [44], which uses the library OMJulia.jl [59] developed by the OpenModelica project [41]. With essentially three relevant lines of code, which can be seen in Listing 5 and 7, the library establishes communication with the OpenModelica compiler (OMC), and then loads a given model, runs a simulation with it and performs a regression test. During model loading and simulation, checks for unit consistency as well as for compiler errors and warnings are performed and any issues are reported with human-readable error messages that include the original compiler message if possible. This means that modelers do not need an in-depth knowledge of the Julia language, or any other programming language, to benefit from thorough automated testing. As shown in Listing 1, they can also set up a CI pipeline for their Modelica project with just two calls to the `julia` executable. If required, however, more fine-grained tests and separate simulation scripts can be defined using the application program interfaces (APIs) of MoST.jl and OMJulia.jl for model inspection and simulation and the testing capabilities of Julia.

An experimental feature of MoST.jl also aims to solve the problem of errors occurring in equation and parameter lists in articles by automatically generating a human-readable documentation of a model. This is based on the function `dumpXMLDAE` in the OpenModelica scripting application programmer interface (API), which generates an eXtensible Markup Language (XML) file containing a flat list of all parameters, variables, functions, and equations in a composite model. The equations are not only listed as code but additionally as content Mathematical Markup Language (MathML), which allows to automatically translate them to presentation MathML, which can be, e.g., rendered in a web browser. Since MoST.jl is written in Julia, we can use the highly extensible documentation generator Documenter.jl to generate an HTML documentation of a model by simply inserting an annotated code-block in a Markdown-formatted text file as shown in Listing 6. This can, again, happen in a CI pipeline, ensuring that there is an accurate human-readable documentation for each version of the model. However, automatic generation of such a documentation from a composite model is not trivial, as variables and functions can have multiple aliases, which introduce clutter that has to be reduced. Additionally, variables have to be grouped to keep the list of equations clear and the variable names in the equations short enough for a visually pleasing presentation. The current

implementation state of this feature is enough to give an idea what is possible, but does not yet produce output that can be used as a supplement in an academic journal. An example for InaMo can be seen at https://cschoel.github.io/inamo/v1.4/models/examples/#Tests-for-I_K,1.

Listing 3. Example for composition via instantiation in InaMo. The gating model GateTS implements the generic Hodgkin-Huxley equation. The ion channel model `SodiumChannel` uses two instances of GateTS with different names `inact_fast` and `inact_slow` for fast and slow inactivation. Both instances use different fitting functions to replace the generic placeholder function ftau for the time constant of the gating variable. This reduces both the need to redundantly define the Hodgkin-Huxley equations and fitting functions like `genLogistic` and therefore keeps the code DRY.

```
model GateTS
  import InaMo.Functions.Fitting.*;
  ...
  replaceable function ftau = genLogistic;
  replaceable function fsteady = genLogistic;
  Real n(start = fsteady(0), fixed = true) "ratio of molecules in open
conformation";
  outer SI.ElectricPotential v_gate "membrane potential of enclosing
component";
  ...
equation
  der(n) = (fsteady(v_gate)-n) / ftau(v_gate);
annotation(...);
end GateTS;
model SodiumChannel
  ...
  GateAB act(...);
  function inact_steady = pseudoABSteady(...);
  GateTS inact_fast(
    redeclare function fsteady = inact_steady,
    redeclare function ftau = genLogistic(
      y_min = 0.00035, y_max = 0.03+0.00035, x0=-0.040, sx=-1000/6.0)
  );
  GateTS inact_slow(
    redeclare function fsteady = inact_steady,
    redeclare function ftau = genLogistic(
      y_min = 0.00295, y_max = 0.12+0.00295, x0=-0.060, sx=-1000/2.0)
  );
  Real inact_total = 0.635 * inact_fast.n + 0.365 * inact_slow.n;
equation
  open_ratio = act.n^3 * inact_total;
end SodiumChannel;
```

Listing 4. Example for ion channel in InaMo. Most ion channels share a base class `IonChannelElectric` that implements the base equations for the electrical analogy to a conductor coupled to a voltage source. Full ion channel models such as `SustainedInwardChannel` then only have to define the `open_ratio` that determines the opening and closing of the channel in dependence of the gating variables.

```
partial model IonChannelElectric "ion channel based on electrical
analog"
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
  parameter SI.ElectricPotential v_eq "equilibrium potential";
  parameter SI.Conductance g_max "maximum conductance";
  SI.Conductance g = open_ratio * g_max "ion conductance";
  Real open_ratio "ratio between 0 (fully closed) and 1 (fully open)";
equation
```

```
  i = open_ratio * g_max * (v - v_eq);
end IonChannelElectric;
model SustainedInwardChannel "I_st"
  extends IonChannelElectric(g_max = 0.1e-9, v_eq=-37.4e-3);
  GateTS act(...);
  GateAB inact(...);
equation
  open_ratio = act.n * inact.n;
end SustainedInwardChannel;
```

Listing 5. ModelicaScriptingTools.jl (MoST.jl) script that loads the model file `src/InaMo/Examples/FullCell/AllCells.mo`, performs simulations according to the simulation parameters read from that file (see Listing 7), places the outputs in the folder out, and performs regression tests if it finds reference data in the directory `regRefData`.

```
using ModelicaScriptingTools
using Test
withOMC("out", "src") do omc
  @testset "Example" begin
    testmodel(omc, "InaMo.Examples.FullCell.AllCells";
refDir="regRefData")
  end
end
```

Listing 6. Markdown-formatted text file that is used to generate a HTML documentation of InaMo including the HTML string from the model file itself, a list of all equations rendered as MathML, a list of all functions in Modelica syntax, and a table with all variables and parameters.

```
# InaMo
Documentation for InaMo.
```@modelica
InaMo.Examples.FullCell.AllCells
```
```

3.3 Availability of data files

3.3.1 Problem description. Inada *et al*. did originally upload their C++ code to the *Biophysical Journal* with the intent to make it available for download, which is to be commended. However, some unknown issue—maybe an update of the publishing platform—seems to have buried this information as there is currently no download link on the journal website. Only after multiple attempts of contacting both the authors and the journal, we were able to obtain the code from the production team of the journal. We asked them to add a download link to the article page so that other researchers would have easier access to the files but received no answer to our request. As mentioned above, information was missing from the article and some errors in equations and parameters were ultimately only recoverable from the C++ code. Without the code we might therefore not have been able to recreate the full cell models at all. Earlier access to the model code could also have reduced the time that was spent fixing bugs in the code.

Listing 7. Experiment annotation of the `AllCells` model, which contains full cell tests for all three model types (AN, N, and NH cells). The parameters `StartTime`, `StopTime`, `Tolerance`, and `Interval` are part of the Modelica language specification, the parameter s for the solver selection is a vendor-specific annotation of OpenModelica and the `variableFilter`, which controls which variables occur in the output file, is a vendor-specific annotation of MoST.jl.

```
model AllCells
  FullCellCurrentPulses an(redeclare ANCell cell);
```

```
  FullCellSpon n(redeclare NCell cell);
  FullCellCurrentPulses nh(redeclare NHCell cell);
annotation(
  experiment(StartTime = 0, StopTime = 2.5, Tolerance = 1e-12,
Interval = 1e-4),
  __OpenModelica_simulationFlags(s = "dassl"),
  __MoST_experiment(variableFilter="(an|n|nh)\\.cell\\.(v|ca\\.(sub|
cyto)\\.c\\.c)")
);
end AllCells;
```

3.3.2 Solution: Services for long-term archival of code and data. We believe that while the management of supplemental data is the responsibility of scientific journals, researchers should not solely rely on this system. Journals and their archival systems are more focused on text content than on data and—as this case shows—can fail to preserve this relevant information or to make it accessible for future research.

With InaMo, we therefore used multiple fail-safe options. First, we publish our code on GitHub, which has adopted a "pace layers" strategy [60] for archiving code in multiple redundant databases with the extreme of the GitHub Arctic Code Vault that is designed to store code for a thousand years [61]. Second, to make our code citable and more easily accessible for research purposes, we also use Zenodo, which assigns document object identifiers (DOIs) to archived code and data and stores it in the CERN Data Centre [62]. Although it does not extend to the same time spans as the GitHub Archive Program, Zenodo might be the most suitable solution for data uploads such as reference data for regression tests, as those are not fully covered by GitHub's program. With this setup, the availability of our data does not depend on a single academic journal but is in the hands of multiple institutions that specialize in keeping code and data available for future generations of researchers.

3.4 Non-executable code

3.4.1 Problem description. Even with both the C++ and CellML implementations of the Inada model available, we could not obtain reference simulation results that we could have used for debugging. The C++ code does not include any file with an executable `main()` function, but only function and variable definitions for the equations and variables of the model. The CellML code *is* executable using OpenCOR, but only the N cell model does produce an action potential with the settings given in the model file. As mentioned in Section 3.2, the N cell model has significant errors in the parameter values of C_m and E_{st}, which does not increase our confidence that the model is in a state that allows it to be used as a reference.

This already means that the methods of Inada *et al.* are not reproducible. Without executable code, there is no way to obtain simulation results in the same way as the authors did. Additionally, this also limits the results reproducibility of the model as there is no reference implementation or simulation data against which we could compare our reimplementation for testing and debugging purposes. We could use the plots as data source, but this is more error-prone as we will explain in the following subsection.

3.4.2 Solution: Continuous delivery. In software engineering, CI pipelines often also include a distribution stage that compiles an artifact which can be distributed to end users if and only if the testing stage did not produce any errors. This process is called continuous delivery (CD), and it can be used in mathematical modeling to ensure that the code that is submitted to a journal or stored in an archive is indeed both complete and correct. GitHub already automatically adds a ZIP archive including the whole repository content to each tagged version of a repository, which can be enough for small projects. In our case, however, we need an

additional step, since the ZIP archive generated by GitHub by default does not include the content of submodules, which we use to store the data files for the regression tests.

The additional work that is required to generate a distribution of a model, is performed by a CI script. In our case, this involves a call to the `zip` tool on the command line and the use of the predefined actions `create-release` and `upload-release-asset` as can be seen in Listing 8.

CD also provides the opportunity to distribute models not only as source code but also in dedicated exchange formats. Since version 1.4.3, InaMo releases contain an export of the main model `AllCells` as Functional Mock-up Unit (FMU). The FMU format is used in the Modelica ecosystem to increase the interoperability of models and to make models available across tool and language barriers. This means that the release version of the `AllCells` model is not only executable in OpenModelica, but also in any of the over 100 tools that support the Functional Mock-up Interface (FMI) [63].

3.5 Missing reference plots and experiment protocols

In order to use unit tests, as suggested in Section 3.2, reference data or plots are required that capture the behavior of a single part of the model and therefore provide target results, which can be reproduced. The data supplement of [7] does contain reference plots for $I_{Ca,L}$, I_{to}, $I_{K,r}$, I_f, I_{st}, and I_{NaCa} as well as voltage and $[Ca^{2+}]_i$ curves for the full cell models. However, reference plots for $I_{K,1}$, I_{Na}, I_p, and I_{ACh} as well as for the $[Ca^{2+}]$ handling are missing. As shown in Fig 1, reference plots for $I_{K,1}$, I_{Na} and I_p could be obtained from the sources that are cited in [7]. This still leaves the $[Ca^{2+}]$ handling and I_{ACh} without reference.

As shown in Table 4, a complete and error-free experiment protocol was only available for $I_{K,1}$. All other experiments required some form of adjustments and in roughly half of the cases no exact agreement with the original plots could be achieved. A common reason for this is that current-voltage relationships of ion channels are usually determined with a test pulse protocol, of which not all parameters were reported in the articles. In this protocol, the voltage is held at a holding potential V_{hold} for a period T_{hold}, after which it is immediately set to a pulse potential V_{pulse} for a duration T_{pulse} followed by another holding period and so on. V_{hold} is gradually increased after each pulse and then plotted against the maximum current obtained in the cycle duration. Inada *et al.* give values for V_{hold}, V_{pulse} and T_{pulse}, but not for T_{hold}. This is relevant, because due to the high time constants of slow activation and inactivation gates, some currents only arrive at a steady state after 20 seconds. If T_{hold} is smaller than this time period, the current during a cycle will also be affected by the previous cycle.

Listing 8. GitHub Actions script to automatically draft a GitHub release each time a new tag is encountered in the repository. After downloading the source code with the `checkout` action, the version number is saved in the `RELEASE_VERSION` variable, and a ZIP archive is created with the `zip` tool. Then, the `create-release` action is used to create the release draft on the GitHub website and the ZIP archive is attached to this draft with the `upload-release-asset` action. This version uses the whole content of the file `README.md` as body text for the release. The full script for InaMo version 1.4.3 parses only the recent changes from the `CHANGELOG.md` file and also contains additional code to attach an Functional Mock-up Unit (FMU) export of the model `InaMo.Examples.FullCell.AllCells` to the release, which is not shown here.

```
on:
  push:
    tags:
      - 'v*' # Push events to matching v*, i.e. v1.0, v20.15.10
jobs:
```

```
  release:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
        with:
          submodules: true
      - run: echo "RELEASE_VERSION=${GITHUB_REF#refs/*/}"
    >> $GITHUB_ENV
      - run: |
          zip -r inamo-${RELEASE_VERSION}.zip . -x \*.git/\* \*.git
      - uses: actions/create-release@v1
        id: create_release
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        with:
          tag_name: ${{ github.ref }}
          release_name: Release ${{ github.ref }}
          body_path: README.md
          draft: true
      - uses: actions/upload-release-asset@v1
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        with:
          upload_url: ${{ steps.create_release.outputs.upload_url }} #
reference previous output
          asset_path: ./inamo-${{ env.RELEASE_VERSION }}.zip
          asset_name: inamo-${{ env.RELEASE_VERSION }}.zip
          asset_content_type: application/zip
```

In other cases, reported parameters had to be adjusted manually to obtain a good agreement with the original plots. This includes simple oversights like wrong units, but also cases where it seems that different values were used than those that were reported, as for S1 Fig in [7], where V_{hold} is given as −40 mV, but we achieve much better results with a value of −70 mV. It also seems that Inada *et al.* used the parameter settings of the NH cell model for plots of $I_{Ca,L}$ and I_{to}, even though the article states that parameters of the AN cell model were used. Another example are the plots for I_{NaCa} by Matsuoka *et al.*, where it is stated that a scaling parameter was used for each of the individual plots, but the value of the scaling parameter is not given in the article.

As a final issue, we could not obtain isolated reference plots for some of the components as they were only used and reported in combination with other components: The only reference that we had for I_p reports the sum of I_p and three background channels, which are different from the background channel used in the Inada model [64]. Also, $[Ca^{2+}]_i$ was only reported in the context of the full cell model by Inada *et al.*

While our simulations are mostly in qualitative agreement with the reference plots, we could not always achieve an exact match. We assume that this is due to further unreported changes in parameter values. For example, for the current density time course of I_{st} in S5B Fig we had to set the parameter g_{st} to 0.27 nS instead of 0.1 nS as reported in [7]; the differences for I_{NaCa} in S6A and S6B Figs vanish when the current densities are multiplied by a scaling factor 1.18, which can be achieved by adjusting k_{NaCa} accordingly; and for I_{to} the current in S2E Fig is slightly lower than in our model, which could be explained if T_{hold} was too small to allow a full recovery to the steady state in [7]. Finally, the differences in the full cell models might be explained if I_{ACh} *was* actually used for simulations. There are only two instances of qualitative differences for I_f in S4C Fig, and for I_{st} in S5C Fig. We have no good explanations for these

differences, but it is unlikely that they are due to an error on our side, since they exist only in the I-V-plots, but not in other plots using the same data or reference plots from other sources.

3.5.1 Solution 1: Run experiments in continuous integration. The hurdles to results reproducibility posed by missing and erroneous information about reference plots and by missing plots themselves can also be solved by employing automated testing. The test cases in InaMo directly produce the simulation data required for a specific reference plot. The model code contains the full experiment protocol including all relevant simulation settings such as the solver and the step size. An example can be seen in Listing 7. The repository also contains a plotting script that reads the simulation output produced by the test script and generates plots for all examples. In consequence, reproduction of the methods of this article becomes possible with a few simple steps: Researchers have to install OpenModelica, the Python distribution Anaconda, and Julia with the single additional package `ModelicaScriptingTools`. Then they can download our code from GitHub and type the following two commands in a command prompt:

```
julia --project="." scripts/unittests.jl
python scripts/plot_validation.py
```

This should result in the creation of a directory called `plots` which contains a reproduction of all the reference plots listed in Table 4. Only I_{ACh} remains untested in InaMo, because we do not have any reference for the equations used in the C++ code of Inada *et al.*

3.5.2 Solution 2: Use dummy components in unit tests. While we did not have a reference plot for the $[Ca^{2+}]$ handling, we still wanted to create a unit test of the component as it was quite difficult to implement, and we wanted to isolate it from any feedback loops to facilitate bug fixing. In software engineering, it is a common issue that a piece of code that should be tested depends on a fairly complex and not fully predictable environment, such as a database or a web service. In these cases, dummy components are used, which provide the same interface as the required service, but actually contain no logic whatsoever and only return the results that are expected and needed for the unit test.

This technique can also be applied to mathematical modeling. For the unit test of the $[Ca^{2+}]$ handling, we approximated the time course of the currents $I_{Ca,L}$, and I_{NaCa} throughout an action potential in the full cell example very roughly with a sum of Gaussians. This leaves us with current signals that have a physiologically plausible shape and value and that do not depend on any other component. The resulting plot therefore shows the behavior of the $[Ca^{2+}]$ handling component in isolation, allowing to examine the effect of changes to this component in a controlled environment.

3.5.3 Solution 3: Publish simulation data used for regression tests. Since we only had plots as a reference, we initially only checked the exactness of our experiment results by comparing the plotted values at prominent sample points like extrema or zero crossings. For the full cell model, we invested the additional effort to reconstruct the simulation data from the plots using the vector graphics editor Inkscape and a small Python script. We then later extended this reconstruction procedure to all other reference plots. This allowed us to immediately assess whether a parameter change brought the simulation result closer to the original data or introduced additional deviations. However, this process is both tedious and inexact. In a first attempt, we underestimated the scale of the x-axis in the plot for the full cell model, which was only given as a small ruler-like segment of 50 milliseconds width. Additionally, we first assumed that the test pulse occurred exactly after 50 milliseconds for each cell type, but later found out that the position differed by a few milliseconds between plots. These errors and the reconstruction effort could have been eliminated, if the simulation data used for the original plots was available for download.

As mentioned above, we publish our simulation output for the regression tests, which includes all data required to reconstruct our reference plots. We also make the reconstructed simulation data from the plots in the original article available. Additionally, our plotting script can be easily configured to produce plots from the reference data instead of or in addition to the simulation output. Therefore, even if there should be some unforeseen issues with running one of our scripts in the future, an exact reproduction of the simulation results will still be possible, because the reference data allows to reliably quantify the error in a reproduction attempt.

3.6 Semantics lost in the chain of references

3.6.1 Problem description. The last problem that we encountered in our reproduction of the results of the Inada model was not so much concerned with correctness and completeness but with the understandability of the model. In an attempt to reproduce simulation results, it is unlikely that the goal is to reproduce the full code with the exact same structure as before. This was also the case for us, as we wanted to include the model in a high-level model of the human baroreflex [66, 67]. For this task, we also wanted to adhere to our MoDROGH guidelines [22]. This required us, among other changes, to bring the model into a modular structure that follows the biological structure as much as possible. For the HH-type ion channel formulations this was straightforward, although we sometimes struggled to understand the reasoning behind the choice of fitting functions.

The I_{Na} formulation, however, follows the Goldman-Hodgkin-Katz (GHK) flux equation, which—unless one is already familiar with this equation—only becomes apparent when reading the reference by Lindblad *et al.* [65]. This posed a problem, because understanding this equation was required for resolving an error in the article: The permeability P_{Na} was given in nl/s by Lindblad *et al.*, which is not a unit for permeability and also has the wrong order of magnitude since it should be pl/(s · m^2). This error can only be found and fixed, if one understands the semantics that P_{Na} is supposed to be the permeability term used in the GHK flux equation.

A similar but more severe problem occurred in the formulation for I_{NaCa}, where the main set of equations that defines the current is the following:

$$x_1 = k_{34}k_{41}(k_{23} + k_{21}) + k_{21}k_{32}(k_{43} + k_{41}) \tag{1}$$

$$x_2 = k_{43}k_{32}(k_{14} + k_{12}) + k_{41}k_{12}(k_{34} + k_{32}) \tag{2}$$

$$x_3 = k_{43}k_{14}(k_{23} + k_{21}) + k_{12}k_{23}(k_{43} + k_{41}) \tag{3}$$

$$x_4 = k_{34}k_{23}(k_{14} + k_{12}) + k_{21}k_{14}(k_{34} + k_{32}) \tag{4}$$

$$I_{NaCa} = k_{NaCa}(k_{21}x_2 - k_{12}x_1)/(x_1 + x_2 + x_3 + x_4); \tag{5}$$

Without further explanation it is nearly impossible to see that this is an analytic solution to the diffusion equations between four states of the sodium potassium pump, of which only the state transitions between state 1 and 2 are electrogenic. Inada *et al.* cite Kurata *et al.* as direct source for I_{NaCa}, but to obtain an explanation of the rationale behind the equations, one has to go one step further to an article by Matsuoka *et al.* [57]. This information was important for us since it meant that we could not further modularize I_{NaCa}, because it would not have been possible to automatically extract the analytic solution from individual diffusion models.

The last and most important example of lost semantics was the $[Ca^{2+}]$ handling. Here, the equations describe the transport of Ca^{2+} cations between four compartments. This is not

$$\frac{\mathrm{d}\left[Ca^{2+}\right]_{\mathrm{sub}}}{\mathrm{d}t} = \frac{-I_{\mathrm{Ca,L}}}{2FV_{\mathrm{sub}}} \quad (a)$$

$$+ \frac{2I_{\mathrm{NaCa}}}{2FV_{\mathrm{sub}}} \quad (b)$$

$$+ j_{\mathrm{rel}}\frac{V_{\mathrm{rel}}}{V_{\mathrm{sub}}} \quad (c)$$

$$- j_{\mathrm{Ca,dif}} \quad (d)$$

$$- [CM]_{\mathrm{tot}}\frac{\mathrm{d}f_{\mathrm{CMs}}}{\mathrm{d}t} \quad (e)$$

$$- SL_{\mathrm{tot}}\frac{\mathrm{d}f_{\mathrm{CSL}}}{\mathrm{d}t} \quad (f)$$

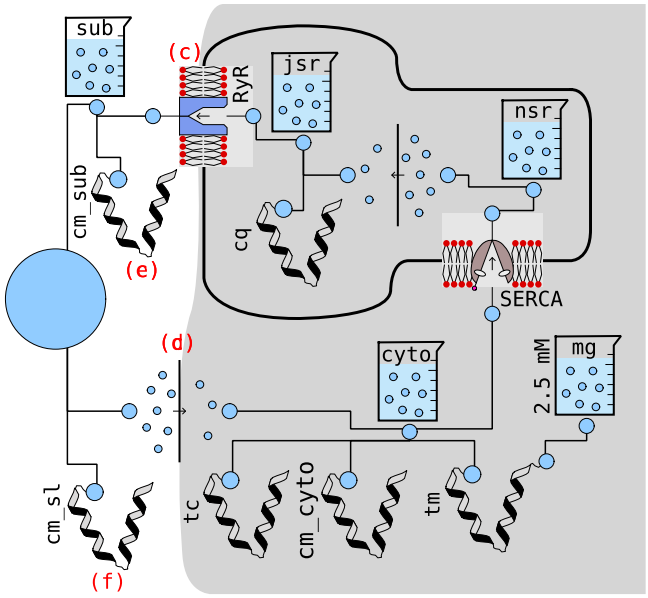


Fig 3. [Ca^{2+}] handling in the Inada model. Left: One of the 15 differential equations that govern the intracellular calcium concentrations as presented in the original article. This single equation mixes the following six physiological effects: the transport of calcium cations through the L-type calcium channels (a) and the sodium-calcium exchanger (b), the release of calcium from the JSR into the subspace via ryanodine receptors (c), the diffusion from the subspace into the cytosol (d), and the calcium buffer calmodulin in the subspace (e) and in the sarcolemma (f). Right: Graphical representation of the [Ca^{2+}] handling in InaMo version 1.4.1. Each component represents a single physiological effect or quantity with intuitive icons for concentrations (beaker), calcium buffers (stylized protein), diffusion reactions (arrow from high to low concentration of circles), the ryanodine receptor (pore in lipid bilayer), and the SERCA pump (scissor-like structure in lipid bilayer). Effects (c)–(f) of the left-hand side equation are represented by the four components connected to the beaker on the upper left (marked in red), while effects (a) and (b) are handled by the external ion channel components when they are connected to the large calcium connector (blue circle) on the center left. These external connections can be seen in Fig 6 (left).

https://doi.org/10.1371/journal.pone.0254749.g003

apparent in [7], but only in [54], which contains a graphical representation of the model. However, Kurata *et al.* still do not couple this understandable graphical representation with the actual equations. Instead of separating them into diffusion reactions, the ryanodine receptor and the SERCA pump, they are only grouped by compartments. Additionally, the effects of all ionic currents on the Ca^{2+} concentration are lumped together in the same equations, which further complicates understanding. An illustration of this problem can be seen in Fig 3. Even after disentangling the equations into small components, we were still confused by the volume terms that were applied to the "flux" variables j_{rel}, j_{up}, j_{tr}, and $j_{\mathrm{Ca,diff}}$ in seemingly arbitrary fashion. An example can be seen in Fig 4. The reason behind this confusing use of volume terms is that the original equations only use *concentrations*, but the transport has to conserve the

General rule:

$$\frac{\mathrm{d}\left[Ca^{2+}\right]_{\mathrm{src}}}{\mathrm{d}t} = \ldots - \frac{\min(V_{\mathrm{src}}, V_{\mathrm{dst}})}{V_{\mathrm{src}}} j_{\mathrm{src,dst}} + \ldots$$

$$\frac{\mathrm{d}\left[Ca^{2+}\right]_{\mathrm{dst}}}{\mathrm{d}t} = \ldots + \frac{\min(V_{\mathrm{src}}, V_{\mathrm{dst}})}{V_{\mathrm{dst}}} j_{\mathrm{src,dst}} + \ldots$$

Equations in article:

$$\frac{\mathrm{d}\left[Ca^{2+}\right]_{\mathrm{up}}}{\mathrm{d}t} = \ldots - j_{\mathrm{tr}}\frac{V_{\mathrm{rel}}}{V_{\mathrm{up}}} + \ldots$$

$$\frac{\mathrm{d}\left[Ca^{2+}\right]_{\mathrm{rel}}}{\mathrm{d}t} = \ldots + j_{\mathrm{tr}} + \ldots$$

Fig 4. Comparison between general rule for inactive transport equations (left) and actual equations occurring in the article by Inada *et al.* (right). The right-hand side is the result of substituting src = up and dst = rel in the left-hand side and then simplifying due to $\min(V_{\mathrm{up}}, V_{\mathrm{rel}}) = V_{\mathrm{rel}}$. If only the right-hand side is given, it is not trivial to trace back these steps to arrive at the general rule, which is required to understand the meaning of the equation. The name "tr" does not immediately make it apparent what are the source (src) and destination (dst) concentrations affected by j_{tr} and since $V_{\mathrm{rel}}/V_{\mathrm{rel}}$ cancels out in the second equation, the structure is also lost.

https://doi.org/10.1371/journal.pone.0254749.g004

$$h_{1\infty} = \dots$$

$$\tau_{h_1} = \frac{0.03}{1 + \exp\left((V + 40)/6\right)} + 0.00035$$

$$\frac{\mathrm{d}h_1}{\mathrm{d}t} = \frac{h_{1\infty} - h_1}{\tau_{h_1}}$$

```
GateTS inact_fast(
  redeclare function fsteady = ...,
  redeclare function ftau =
    genLogistic(
      y_min=0.00035, y_max=0.03+0.00035,
      x0=-0.040, sx=-1000/6.0
    )
) "inactivation gate (type1/h1)";
```

Fig 5. Equations for the fast inactivation gate of I_{Na} in the original article (left) and in InaMo (right). The equations on the left-hand side constitute a typical description of an HH-type ion channel using a steady state $h_{1\infty}$ and a time constant τ_{h_1} to define the time course of the gating variable h_1 via a differential equation. The equation for τ_{h_1} was found by fitting a generalized logistic function to experimental data. It has a declining sigmoid shape with an inflection point at 40 mV, a minimum of 0.35 ms, and a maximum of 30.35 ms. This may be apparent for an expert modeler, who is familiar with similar models, but not to novices or biologists without a deep mathematical background. The InaMo code on the right-hand side therefore aims to make this expert view of the equations available to non-experts by capturing common equation structures in named and documented components. `GateTS` defines a HH-type gating variable based on the two replaceable functions `fsteady` for the steady state and `ftau` for the time constant. `genLogistic` is a fitting function, whose parameters are explained in its documentation: `y_min` is the minimum, `y_max` is the maximum, `x0` is the inflection point, and `sx` determines the steepness and direction (`sx` < 0 yields a declining sigmoid shape).

https://doi.org/10.1371/journal.pone.0254749.g005

amount of substance between both sides. This introduces the need to convert from concentrations to amounts of substance (by multiplying with a volume term) and then back to concentrations (by dividing by another volume term). Even with this background knowledge, which is assumed by the articles about the $[Ca^{2+}]$ handling, it is not trivial to infer the general rule from the equations in the article. This is due to simplifications, which obscure the common equation structure, and the naming of the flux variables, which does not clearly indicate source and destination of the corresponding transport effect.

3.6.2 Solution 1: Model design utilizing MoDROGH criteria. The software engineering equivalent of these unclear, entangled and undocumented model semantics is termed "spaghetti code", which is code that is hard to maintain, because the program flow is hard to follow. The solution to this problem is a combination of modularization, documentation and clear design patterns for the code. As mentioned in Section 2.4, InaMo follows the guidelines associated for building models with a language that is modular, descriptive, human-readable, open, graphical, and hybrid (MoDROGH), which can increase the understandability as well as the methods and results reproducibility of a model [22].

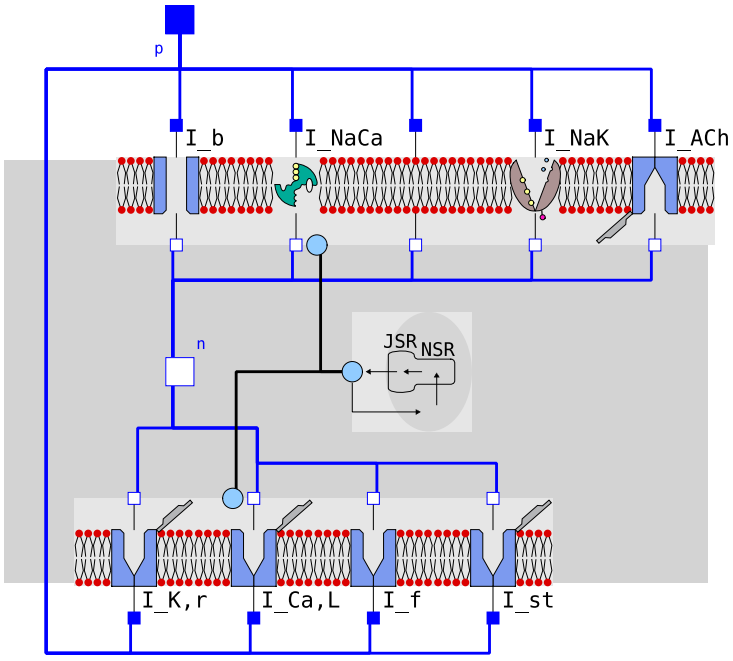
The issue with non-transparent fitting functions is solved by defining a set of fitting functions with understandable names and a common structure for gate components. As Fig 5 shows, this allows to understand the gate equations without having to untangle the structure of the fitting functions in memory. For example, the most common fitting function `genLogistic` can be quickly identified as a sigmoid function, whose parameter `x0` defines the point of maximum steepness, while `y_min` and `y_max` define the minimum and maximum value that the function can assume. It also becomes apparent that almost all gates use HH-type equations governed by a time constant and a steady state function.

Similarly, the GHK flux equation is implemented in a separate component that features both a detailed documentation in HTML format and explicit unit definitions, including the custom type `PermeabilityFM` that is used to document the unusual unit used for P_{Na}. This also fixes a minor issue mentioned in Section 3.1, as the documentation also explains the handling of the avoidable discontinuity in the function.

As mentioned above, the equations for the sodium calcium exchanger unfortunately could not be modularized to make more explicit that they are intended to model diffusion reactions between four states. However, we added a documentation string to each variable explaining its physiological interpretation. We also added the variables E_1–E_4 from Matsuoka *et al.*, which

represent the actual ratio of molecules in each of the four states and therefore facilitate the interpretation of the behavior of the component.

Finally, we already showed the effect that modularization has on the $[Ca^{2+}]$ handling in Fig 3. By separating the model into modules that each only represent a single physiological effect, these individual effects become more understandable. Readers can focus on understanding one module at a time, grouping the equations and parameters in memory to form a concept that can easily be recalled. With these concepts in mind, understanding the whole component becomes possible by inspecting its graphical representation, which shows how the individual effects are connected. This is facilitated by the fact that the graphical representation is neither a separate biological drawing, nor an automatically generated graph, but rather an accurate representation of the model defined with Modelica constructs similar to a circuit diagram. An example showing the definition of the graphical representation in the Modelica code can be seen in Fig 6. On the code and equation level, InaMo uses amounts of substance instead of concentrations as interface. This leads to a more natural representation of active and inactive transport components, which explicitly ensure conservation of mass. The diffusion reactions and the ryanodine receptor use a common base class `InactiveChemicalTransport`, which clearly explains the use of volume terms and presents the gradient-based transport equations in their general, more understandable form. Additionally, we change the naming of the individual concentrations from $[Ca^{2+}]_i$, $[Ca^{2+}]_{up}$ and $[Ca^{2+}]_{rel}$ to $[Ca^{2+}]_{cyto}$, $[Ca^{2+}]_{nsr}$ and $[Ca^{2+}]_{jsr}$ respectively, which allows us to also assign intuitive names to the transport




```
model NCell
  BackgroundChannel bg
    annotation(Placement(
      transformation(
        origin = {-51, 53},
        extent = {{-17, -17}, {17, 17}}
      )
    ));
  ...
equation
  connect(bg.p, p) annotation(Line(
    points = {{-50, 70}, {-50, 100}},
    color = {0, 0, 255}
  ));
  ...
annotation(Diagram(graphics={Rectangle(
  fillColor = {211, 211, 211},
  pattern = LinePattern.None,
  fillPattern = FillPattern.Solid,
  extent = {{-100, 60}, {100, -60}}
)}));
end NCell;
```

Fig 6. Graphical representation of the N cell model. Left: Diagram resulting from drag and drop composition of model components (InaMo version 1.4.1). Right: Automatically generated embedding of graphical annotations in the model code showing the placement of the background channel (`annotation(Placement(...))`), the connection line to the positive pin (`annotation(Line(...))`) and the definition of the gray rectangle in the background (`annotation(Diagram(...))`).

https://doi.org/10.1371/journal.pone.0254749.g006

components. For example, the module for the diffusion from the subspace to the cytosol is called `sub_cyto`.

Apart from these individual examples, InaMo uses the general MoDROGH guidelines to ensure that the model code reflects the physiological semantics as much as possible, making them transparent for the user. For example, the whole model only uses two kinds of interfaces: an electrical interface for ion currents and a chemical interface governing the changes in the amount of ions in a compartment. Following the convention that outward currents are positive, each ionic current has a positive pin on the extracellular side and a negative pin on the intracellular side. The electrical interface is also compatible to electrical components of the Modelica standard library `Modelica.Electrical`, allowing standard electrical components such as a ground or current source to be used in InaMo. Both kinds of interfaces can be seen in Fig 6, where electrical connections are represented by blue squares, which are filled for positive pins, and chemical connections are represented by blue circles. It can also be seen that no component has more than three of these connections, which keeps the cognitive effort required to understand them at a low level.

Listing 9. Interface for electrical connections between model components in the Modelica standard library. The keyword `flow` establishes an acausal connection with the conservation law that the sum of the `i` variables of all connected components must be zero.

```
connector PositivePin "Positive pin of an electrical component"
  SI.ElectricPotential v "Potential at the pin" annotation(...);
  flow SI.Current i "Current flowing into the pin" annotation (...);
  annotation (...);
end PositivePin;
```

One important aspect of these interfaces is that they are *acausal*, which means that no prior assumption is made which variables are defined by input signals and which will be observed as the output of an experiment. For example, this means the same model code can be used for voltage- and current-clamp experiments. Modelica achieves this by using connector variables with the `flow` keyword and automatically generating conservation law equations representing Kirchhoff's current law for the electrical interfaces and the conservation of mass for the chemical interface. An example of such a connector definition can be seen in Listing 9. The most important effect of this feature for the design of the model is that adding or removing ion currents is as easy as adding and removing the component and its connections in the graphical representation, which automatically adds or removes the required term to the conservation law equations.

These two physical connectors ensure that the model structure in the code follows the biological structure of the modeled system. The full cell is composed of models of the lipid bilayer, ion channels, ionic pumps, and the $[Ca^{2+}]$ handling, which only exposes $[Ca^{2+}]_{sub}$ as the concentration that is relevant for the ion currents. The ion channels, in turn, contain gate models which are composed of basic HH-type gates with predefined or custom fitting functions. This structure closely ties the equations to their semantic meaning and therefore facilitates interpretation.

At the lowest hierarchical level, each variable and parameter in the model is annotated with proper units following the SI and has both a human-readable name and a documentation string explaining its physiological role. Models that are more involved additionally contain a documentation text in HTML format with detailed information about the model structure. This ensures that the model is understandable without further literature research.

3.6.3 Solution 2: Annotation of sources and rationale for parameter values. To spare researchers that want to reproduce our model skimming through a large body of literature as in Fig 1 and to make our parameter choices transparent, we annotated the experiments with a

literature source or rationale for each parameter value. Currently, this is done within the HTML documentation string of the models defining the experiments. However, Modelica also allows using so-called vendor-specific annotations to add structured annotations with custom content. This feature could be used to make these annotations not only human- but also machine-readable and, for example, allow to automatically add this information to the table of parameters generated by MoST.jl.

4 Discussion

4.1 Answer to RQ1

In research question RQ1, we asked which factors hindered the reproduction of the methods and results of the Inada model. Despite the efforts of the authors to provide detailed reference plots and publish their code, a considerable reverse engineering effort was required to build our Modelica implementation InaMo. Fig 1 in S1 Text shows an estimation of the distribution of the working time that went into InaMo. In total, the development took us an estimated 86 work days (i.e. four months). Small errors in published equations and parameter values required the debugging (19 working days) of individual parts of the model. This debugging was hindered by missing information about some model components and missing and incomplete reference code and experiment protocols. This in turn required further literature research (33 working days) to recapture the model components and the semantics of their equations, which revealed several issues with the understandability of the equations, which were structured for ease of implementation and not for ease of understanding. It is quite conceivable that other researchers before us have encountered these issues and decided that the benefit of including the Inada model in their research did not warrant the effort that would be required to do so. This is especially unfortunate since it is a ground-breaking model with several interesting properties, which deserves more attention.

4.2 Answer to RQ2

In our second research question RQ2, we wanted to know what can be done to overcome or to avoid these reproducibility issues. With InaMo we have adopted a model engineering strategy, that uses a suitable MoDROGH language to apply proved techniques from software engineering to the problem. These approaches broadly fall into three categories: First, we established an automated testing pipeline using CI and CD techniques to guarantee completeness and methods reproducibility of the published version of the model. This also includes automated unit consistency checks, performing the actual simulations used for plots in the article in the CI pipeline, and publishing the simulation data both for the reproduction of results and to use them in regression tests. Second, we paid special attention to the model design to increase the understandability and reusability of the model code, using a MoDROGH language and building a component hierarchy with two simple interfaces and small independent components, which only represent a single physiological effect or compartment, and which can be combined via drag and drop in an easily interpretable graphical representation. Third, we provided extensive documentation both within the model in the form of unit definitions, human-readable variable names, and embedded HTML documents and through external services for version control and archival.

Both testing (11 working days) and refactoring and documentation (16 working days) took considerable effort (see Fig 1 in S1 Text). However, we found that this additional effort was well justified by the benefits gained during development, even if we do not consider the benefits for other researchers who want to reproduce our methods or results. For debugging, it was invaluable to have a CI pipeline performing regression tests for individual components,

because we would immediately notice when a change accidentally introduced an error in other models than the one that was currently under development. Utilizing the version control system, we could quickly identify and roll back the changes that introduced bugs. As concrete example, we added unit and regression tests for the individual components of the $[Ca^{2+}]$ handling precisely because they helped us to ensure that our transformation of the model structure from a concentration interface to an interface using amounts of substance did not change the simulation output. Transforming the model into a modular structure that follows the biological structure of the modeled system also helped us to notice some of the errors in the original model, which we would otherwise have overlooked. Much like explaining a concept to somebody else can reveal own misconceptions, making a model more understandable can reveal potential error sources. Additionally, documenting the meaning of variables and parameters as well as the source and rationale of parameter values meant that we only had to look up such information once. This reduced the time required to get an overview over a part of the code that we had to revisit a few weeks or months after it was written. We are positive that without these measures, finding the last errors that prevented us from obtaining reasonable simulation results with the full cell models would have taken considerably more time, if we had achieved a working reproduction at all.

It also has to be noted that the effort required for the solutions presented in this article can and already has been reduced for other researchers. During the development of InaMo, we created the Julia library MoST.jl, which allows setting up tests with only three relevant lines of code (see Listing 5) and provides more and better readable error messages than the OpenModelica compiler when used with default settings. Setting up these tests on a CI service does not require much more effort. In fact, if the same folder structure is used as in our project, it would be possible to simply copy the GitHub Actions configuration script shown in Listing 1.

Furthermore, the systems biology community could choose to provide own CI/CD pipelines using open source tools like Jenkins [68, 69] or services like NF-CORE [25] or FAIRDOMHub [70], which are already established in bioinformatics and systems biology. This way, specific virtual machine images and/or pipelines for common modeling languages could be provided, which already include all necessary tools for simulation and plotting. This would further reduce both the size of the setup script and the execution time required.

4.3 Generalization and alternatives

While this work is only a case study of the Inada model, we believe that the issues that we found here and the solutions that we presented can be highly relevant for mathematical modeling in systems biology in general. For example, the aforementioned reproducibility study of models in the BioModels database found very similar errors and reproducibility hurdles in half of the 455 examined models [27]. In summary, this study lists the following reproducibility issues: sign errors, missing terms in equations, typing errors in parameter values, unit errors, missing or incorrect parameter values, missing or incorrect initial concentrations, errors in equations, ambiguous or inconsistent variable names, and poor readability and lacking documentation in the code. Our case study of the Inada model showcased concrete examples for each of these categories, which indicates that it at least can be representative for these 455 other models. If the issues are similar, it is reasonable to assume that the same or similar solutions like the ones that we used here will also work for other models.

This is further backed by the fact that the software engineering techniques that we applied, such as version control, CI and delivery, automated testing, modularization, and documentation, are not limited to any specific property of the Inada model. They can be, and in fact are, applied to all kinds of software solutions. There are some adjustments required for

mathematical modeling, such as the development of specialized testing libraries like MoST.jl. However, there is no reason to believe that there is any area of mathematical modeling that cannot benefit from these general techniques in some way.

We also think that the Inada model is a fitting example to represent reproducibility challenges in the development of multi-scale models, including a large number of equations, the combination of different preexisting models, and the need to incorporate the model into a larger multi-scale context. As mentioned in Section 1, the three groups that did reproduce the results of Inada *et al*., did so in a multi-scale context, and this was also our original purpose. The model is in itself a combination of multiple existing models for ionic currents and the $[Ca^{2+}]$ handling by the sarcoplasmic reticulum. With its 116 equations, 79 parameters, and 27 initial values, which are only partly shared between the three different cell types, it is large enough that it can no longer be handled well in a classic monolithic structure that only lists all equations in a loosely grouped fashion. The Inada model therefore shows that even a well-crafted and relevant model can be subject to reproducibility issues, simply because of its inherent complexity.

Furthermore, our findings are not specific to the language Modelica. Integration for scripting languages like Python or Julia also exist, for example, for SBML [71, 72] and CellML [73]. This is sufficient to utilize the unit testing features of these languages and to build a simulation script that can be run in a CI pipeline. One remaining caveat is the need to download and install all software necessary to run the script on the CI server, which rules out proprietary solutions with expensive licenses such as MATLAB/Simscape. Regarding the model design utilizing MoDROGH criteria, our previous work shows that multiple languages exhibit MoDROGH criteria [22] and illustrates trade-offs between different choices. We did use some Modelica features for our design that do not exist in other languages like SBML and CellML. This includes the graphical composition of models, object-oriented programming with multiple inheritance, acausal connections between electrical and chemical components, the grouping of interface variables to connectors, and the annotation of the experiment setup within the model file itself. It is also interesting to note that unlike CellML and SBML, which are mainly designed as exchange formats, Modelica code focuses on human-readability over machine-readability and is designed to be directly written by humans. This removes tool-specific barriers between model designer and model user and avoids clutter in version control systems [22]. However, Modelica also has downsides: A CI/CD pipeline is only possible with the open source compiler from OpenModelica, and not with the proprietary compiler for the IDE Dymola, which is more widespread in industry and not fully compatible with OpenModelica, although both implement the same language standard. Additionally, Modelica is a general purpose modeling language, which lacks biology-specific features and language constructs like annotation of components with ontology terms, or the `<kineticLaw>` tag in SBML.

As an important implication, Modelica and SBML or CellML tools are not interoperable. This is important, because interoperability allows model users to reproduce results using the tools that they are familiar with and thus to easily combine models. Modelica's mechanism for interoperability is the FMI that allows to create executable artifacts, so called FMU, from models, which can be reused even across different languages. On the downside, these FMUs are mostly opaque boxes. They can contain source files in C and a list of variables and equations in JavaScript Object Notation (JSON), but they are not suitable for results replication with modifications that go beyond changing parameter settings. In contrast, SBML and CellML are directly designed as exchange formats, which is why they are based on XML. Both FMI and SBML report support by over 100 tools [63, 74], but crossing between the two ecosystems is more difficult. SBML2Modelica allows to directly translate SBML models to Modelica [75],

but we are not aware of any tool that operates in the opposite direction. The only tools used in systems biology that also support FMI currently are MATLAB with the SimBiology and the FMI toolbox, and custom Python code using appropriate libraries for both standards. As a first remedy, FMI support could be added in SBML and CellML tools. Alternatively, the systems biology community could, like Modelica, adopt the software engineering practice to distinguish between "source" and "distribution" formats for models. In this analogy, SBML and CellML would be used as distribution formats, which are used to make models easily accessible for simulations by other researchers, but models would additionally be published in a more human-readable and version control-friendly source language like Antimony [40] or CellML Text [53], which can directly be used for model development. In the case of this article, it would be ideal to have a Modelica2SBML tool, that compiles from the source language Modelica to the distribution language SBML. This is not possible in general, because Modelica supports more formalisms than SBML, but it might be possible for a subset of the language. As a first compilation step, Modelica models are transformed into a "flat" model that collects all variables, functions, events, and equations in a single file without any hierarchy or modular structure. If the translation process is restricted to a subset of the Modelica language, a translation of a "flat" model to SBML code might be achievable. However, this would also mean that the benefit of the modularity and understandability of Modelica models is largely lost in translation. It becomes clear that further research is needed to bridge this gap.

Regarding alternatives, GitHub Actions is not the only choice to implement a CI/CD pipeline. The open source project Jenkins [68, 69] can be used to set up a server that is controlled by a scientific institution, a journal, or a specific lab, alleviating privacy concerns when working with patient data. Additionally, other major open source repository hosting providers like BitBucket [76] and GitLab [77] also offer CI pipelines with varying amounts of free computing time for open source projects. Finally, modeling-specific solutions could be implemented in existing workflow environments like NF-CORE [25] or FAIRDOMHub [70].

Our findings can also be seen in a more general light with respect to the FAIR Guiding Principles for scientific data management and stewardship [78], since they contribute to making the model code findable, accessible, interoperable, and reproducible (FAIR). InaMo is findable in the Zenodo database [38], on GitHub (https://github.com/CSchoel/inamo), and in the BioModels database (https://www.ebi.ac.uk/biomodels/MODEL2102090002). Zenodo allows us to cite specific versions of the code with a unique DOI and GitHub provides a platform for discussing issues and open questions regarding the implementation. BioModels, Zenodo, and the GitHub Archive Program also contribute to making the code accessible for future researchers. Interoperability is provided by the CI/CD pipeline, which ensures that the code runs on other machines. Additionally, the main model is exported as executable artifact using the FMI, which allows to incorporate it in other projects even across different modeling languages. The modular design utilizing MoDROGH criteria and the additional documentation effort for InaMo do not only facilitate reproduction but also reuse, because the model becomes more understandable and extensible. Additionally, each published version of the model uses an open license (MIT license for Zenodo and GitHub, and Creative Commons CC0 1.0 for BioModels). However, as mentioned before, Modelica does not directly support the annotation of model parts with ontology terms. For full compliance with the FAIR Guiding Principles, this has to be addressed either by using vendor-specific annotations and developing tools that can read and write ontology data in Modelica models, or by implementing common ontologies like the systems biology ontology (SBO) [79] as a type hierarchy in Modelica as outlined in [22].

4.4 Limitations

There are some limitations to our approach regarding unit testing. First, unit tests are only meaningful, if the "unit" in question can be used in a simulation that does not involve other complex components. For example, for the $[Ca^{2+}]$ handling we had to create dummy components to mimic the time course of $I_{Ca,L}$ and I_{NaCa} during an action potential in order to obtain a meaningful curve for $[Ca^{2+}]_i$. It is possible that other models may include components that require so many connections to other parts of the model that creating a unit test requires a lot of effort. However, it can be argued that such a component should then be investigated for possible design flaws, since the goal in a modular design is to minimize the interface of a component.

Another problem can be the lack of reference data. Our current unit tests already can be criticized, because they do not follow the usual pattern of a test that has a defined input and an expected output. We only test that the simulations runs error-free and the output is only compared to the output of a previous iteration with the help of regression tests. Tying this output to the actual goal of approximating measured data from biological experiments is currently still performed by a human who has to inspect and compare the resulting plots. For InaMo, this was the only approach we could take, since most of the experimental data was only available as plots, and we would have to reconstruct the original data points by hand. We did this for the simulation output of the models, but not for the experimental data, because the latter is even harder to read from the plots. If the data were published and included in the repository, it would also be possible to define a new kind of test in MoST.jl, which tests the agreement between experimental data and simulation output by some metric such as the root-mean-square error. However, this is of course only possible if such data is available and this may not be the case at every level of detail, limiting the usefulness of unit testing.

Additionally, it has to be considered that test suites like ours can become computationally demanding. We currently run the full simulations that we use to produce our result plots from within the test suite, because it is convenient to only need one script and because this ensures that the CI server reproduces our methods in every iteration. However, if we had included the full one-dimensional model by Inada *et al.* with hundreds of cells, this would mean that our test suite might not run in a few minutes but instead require hours. This prolongs the response time unduly, which limits the usability for quickly testing and debugging small changes to the models. One solution to this problem is to limit the length or size of the simulations in the test suite and to add a second script that is used to produce the actual simulation result. However, this then introduces a source for errors since the content of this new script cannot be tested using CI. For example, in another model, we encountered an error related to the synchronization between two event sources that only occurred after 170 seconds of simulated time.

Apart from the computational effort, the human effort in designing a model with MoDROGH criteria can also be significant. Most modelers probably did not receive training in software engineering and therefore first have to learn to apply design patterns. This is especially difficult, because while guidelines can help, good software design ultimately arises from experience and experimentation. We argue that the benefits are worth the additional effort, but the initial barrier may be high for many systems biologists.

Even if an understandable modular model structure is achieved during development, it is still likely that the model has to be translated to a grouped list of equations for presentation in a scientific article or even just to communicate some details to a researcher who is more familiar with this representation. Even though OpenModelica does allow to inspect a flattened version of arbitrarily complex models, this representation includes a lot of visual clutter due to alias variables that are introduced by the hierarchical structure. It is therefore not trivial to

translate this code into a human-readable list of equations. This task can be facilitated by libraries producing automated documentation like MoST.jl, but this feature of MoST.jl is still in an experimental stage. At the same time, we are not aware of any other approaches that provide similar facilities for flattening highly modular model structures in an equation-based format while retaining the grouping information from the modular design.

A similar argument can be made about documentation. In software engineering, there is little doubt that documentation is valuable and even essential for understandable and maintainable code, yet it is often lacking, even in large, successful projects. This is because writing good documentation requires a lot of time and does not generate its benefit at the time of writing, but only at a later stage in the project. For systems biology in particular, we see the concern that there is not much incentive to document code beyond ones own understanding. This would be different, if academic journals did not only require the code to be available but also had some requirements for understandability and documentation standards.

5 Conclusion

From our case study we can derive several suggestions for tackling reproducibility issues in mathematical modeling in systems biology. Using a CI service, like GitHub Actions, in conjunction with unit and regression tests that are as fine-grained as possible can guarantee methods reproducibility and the completeness of the published code and data. The more automated tests can be performed within such a system, the better the chances for the model to be reproducible and reusable in different ways. It might be worthwhile for the systems biology community to consider implementing or using a CI service with predefined virtual machine images for typical modeling workflows. These images could then be archived allowing not only the long-term storage of the model code but also of the software that was used to simulate it. Journals like *PLOS Computational Biology* and the *Physiome Journal*, which already employ rigorous testing of reproducibility standards by reviewers, might be able to host such a service to provide authors with a standardized mechanism to facilitate reproducibility and to reduce the burden placed on reviewers. Beyond methods reproducibility, results reproducibility cannot be guaranteed by automated tests. They do increase the likelihood that a reproduction attempt is successful, but it might still be complicated by missing documentation or poor understandability of the code. Here, the MoDROGH guidelines or similar "style guides" for model code, can help to make models approachable and reusable for other researchers. However, the only thing that truly guarantees results reproducibility is and remains an actual validation study. We therefore suggest that more of these studies should be performed and published and that there should be some way to indicate that the results of a model have been successfully reproduced in model repositories. In general, we find the philosophy of model engineering, i.e. the application of software engineering techniques to mathematical modeling, very promising. We think that building models with more care to design and engineering aspects will both benefit the scientific impact of a model and scientific progress in systems biology as a whole. In particular, we hope that InaMo, our understandable implementation of the Inada model with reproducible methods and results, can kick-start some new projects on the electrophysiological properties of the AV node.

Supporting information

S1 Text. Data supplement.
(PDF)

Author Contributions

Conceptualization: Christopher Schölzel.

Data curation: Christopher Schölzel.

Formal analysis: Christopher Schölzel.

Investigation: Christopher Schölzel.

Methodology: Christopher Schölzel.

Software: Christopher Schölzel.

Supervision: Alexander Goesmann, Andreas Dominik.

Validation: Christopher Schölzel, Valeria Blesius, Gernot Ernst.

Visualization: Christopher Schölzel.

Writing – original draft: Christopher Schölzel.

Writing – review & editing: Valeria Blesius, Gernot Ernst, Alexander Goesmann, Andreas Dominik.

References

1. Waltemath D, Wolkenhauer O. How Modeling Standards, Software, and Initiatives Support Reproducibility in Systems Biology and Systems Medicine. IEEE Transactions on Biomedical Engineering. 2016; 63(10):1999–2006. https://doi.org/10.1109/TBME.2016.2555481
2. Medley JK, Goldberg AP, Karr JR. Guidelines for Reproducibly Building and Simulating Systems Biology Models. IEEE transactions on bio-medical engineering. 2016; 63(10):2015–2020. https://doi.org/10.1109/TBME.2016.2591960
3. Stodden V, Seiler J, Ma Z. An Empirical Analysis of Journal Policy Effectiveness for Computational Reproducibility. Proceedings of the National Academy of Sciences. 2018; 115(11):2584–2589. https://doi.org/10.1073/pnas.1708290115
4. Topalidou M, Leblois A, Boraud T, Rougier NP. A Long Journey into Reproducible Computational Neuroscience. Frontiers in Computational Neuroscience. 2015; 9(30):1–2. https://doi.org/10.3389/fncom.2015.00030
5. Plesser HE. Reproducibility vs. Replicability: A Brief History of a Confused Terminology. Frontiers in Neuroinformatics. 2018; 11:76. https://doi.org/10.3389/fninf.2017.00076
6. Goodman SN, Fanelli D, Ioannidis JPA. What Does Research Reproducibility Mean? Science Translational Medicine. 2016; 8(341):341ps12. https://doi.org/10.1126/scitranslmed.aaf5027
7. Inada S, Hancox JC, Zhang H, Boyett MR. One-Dimensional Mathematical Model of the Atrioventricular Node Including Atrio-Nodal, Nodal, and Nodal-His Cells. Biophysical Journal. 2009; 97(8):2117–2127. https://doi.org/10.1016/j.bpj.2009.06.056
8. Noble D, Garny A, Noble PJ. How the Hodgkin-Huxley Equations Inspired the Cardiac Physiome Project. The Journal of Physiology. 2012; 590(11):2613–2628. https://doi.org/10.1113/jphysiol.2011.224238
9. Yu T, Lloyd CM, Nickerson DP, Cooling MT, Miller AK, Garny A, et al. The Physiome Model Repository 2. Bioinformatics. 2011; 27(5):743–744. https://doi.org/10.1093/bioinformatics/btq723 PMID: 21216774
10. Sandve GK, Nekrutenko A, Taylor J, Hovig E. Ten Simple Rules for Reproducible Computational Research. PLoS Computational Biology. 2013; 9(10):e1003285. https://doi.org/10.1371/journal.pcbi.1003285
11. Lewis J, Breeze CE, Charlesworth J, Maclaren OJ, Cooper J. Where next for the Reproducibility Agenda in Computational Biology? BMC Systems Biology. 2016; 10(1):52. https://doi.org/10.1186/s12918-016-0288-x
12. Mulugeta L, Drach A, Erdemir A, Hunt CA, Horner M, Ku JP, et al. Credibility, Replicability, and Reproducibility in Simulation for Biomedicine and Clinical Applications in Neuroscience. Frontiers in Neuroinformatics. 2018; 12. https://doi.org/10.3389/fninf.2018.00018 PMID: 29713272
13. Hellerstein JL, Gu S, Choi K, Sauro HM. Recent Advances in Biomedical Simulations: A Manifesto for Model Engineering. F1000Research. 2019; 8:261. https://doi.org/10.12688/f1000research.15997.1

14. Kirouac DC, Cicali B, Schmidt S. Reproducibility of Quantitative Systems Pharmacology Models: Current Challenges and Future Opportunities. CPT: Pharmacometrics & Systems Pharmacology. 2019; 8(4):205–210. https://doi.org/10.1002/psp4.12390

15. Papin JA, Mac Gabhann F, Sauro HM, Nickerson D, Rampadarath A. Improving Reproducibility in Computational Biology Research. PLOS Computational Biology. 2020; 16(5):e1007881. https://doi.org/10.1371/journal.pcbi.1007881

16. Medley JK, Choi K, König M, Smith L, Gu S, Hellerstein J, et al. Tellurium Notebooks—An Environment for Reproducible Dynamical Modeling in Systems Biology. PLOS Computational Biology. 2018; 14(6): e1006220. https://doi.org/10.1371/journal.pcbi.1006220 PMID: 29906293

17. Afgan E, Baker D, Batut B, van den Beek M, Bouvier D, Čech M, et al. The Galaxy Platform for Accessible, Reproducible and Collaborative Biomedical Analyses: 2018 Update. Nucleic Acids Research. 2018; 46(W1):W537–W544. https://doi.org/10.1093/nar/gky379 PMID: 29790989

18. Berthold MR, Cebron N, Dill F, Gabriel TR, Kötter T, Meinl T, et al. KNIME: The Konstanz Information Miner. In: Preisach C, Burkhardt H, Schmidt-Thieme L, Decker R, editors. Data Analysis, Machine Learning and Applications: Proceedings of the 31st Annual Conference of the Gesellschaft Für Klassifikation. Freiburg, Germany; 2008. p. 319–326.

19. Sauro H, Gennari J, Karr J, Moraru I. Center for Reproducible Biomedical Modeling; 2021. https://reproduciblebiomodels.org/.

20. Karr JR, Sanghvi JC, Macklin DN, Gutschow MV, Jacobs JM, Bolival B, et al. A Whole-Cell Computational Model Predicts Phenotype from Genotype. Cell. 2012; 150(2):389–401. https://doi.org/10.1016/j.cell.2012.05.044 PMID: 22817898

21. Millard P, Smallbone K, Mendes P. Metabolic Regulation Is Sufficient for Global and Robust Coordination of Glucose Uptake, Catabolism, Energy Production and Growth in Escherichia Coli. PLOS Computational Biology. 2017; 13(2):e1005396. https://doi.org/10.1371/journal.pcbi.1005396

22. Schölzel C, Blesius V, Ernst G, Dominik A. Characteristics of Mathematical Modeling Languages That Facilitate Model Reuse in Systems Biology: A Software Engineering Perspective. npj Systems Biology and Applications. 2021; 7(1):27. https://doi.org/10.1038/s41540-021-00182-w

23. Schölzel C, Blesius V, Ernst G, Dominik A. An Understandable, Extensible, and Reusable Implementation of the Hodgkin-Huxley Equations Using Modelica. Frontiers in Physiology. 2020; 11:583203. https://doi.org/10.3389/fphys.2020.583203

24. Shahin M, Ali Babar M, Zhu L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access. 2017; 5:3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629

25. Ewels PA, Peltzer A, Fillinger S, Patel H, Alneberg J, Wilm A, et al. The Nf-Core Framework for Community-Curated Bioinformatics Pipelines. Nature Biotechnology. 2020; 38(3):276–278. https://doi.org/10.1038/s41587-020-0439-x PMID: 32055031

26. Sarma GP, Jacobs TW, Watts MD, Ghayoomie SV, Larson SD, Gerkin RC. Unit Testing, Model Validation, and Biological Simulation. F1000Research. 2016; 5:1946. https://doi.org/10.12688/f1000research.9315.1

27. Tiwari K, Kananathan S, Roberts MG, Meyer JP, Sharif Shohan MU, Xavier A, et al. Reproducibility in Systems Biology Modelling. Molecular Systems Biology. 2021; 17(2). https://doi.org/10.15252/msb.20209982 PMID: 33620773

28. Szilágyi SM, Szilágyi L, Benyó Z. A Patient Specific Electro-Mechanical Model of the Heart. Computer Methods and Programs in Biomedicine. 2011; 101(2):183–200. https://doi.org/10.1016/j.cmpb.2010.06.006

29. Szilágyi SM, Szilágyi L, Hirsbrunner B. Modeling the Influence of High Fibroblast Level on Arrhythmia Development and Obstructed Depolarization Spread. In: Computing in Cardiology 2013. vol. 40. Zaragoza, Spain; 2013. p. 45–48.

30. Szilágyi SM, Szilágyi L, Hirsbrunner B. Simulation of Arrhythmia Using Adaptive Spatio-Temporal Resolution. In: Computing in Cardiology 2013. vol. 40. Zaragoza, Spain; 2013. p. 365–368.

31. Boyle PM, Veenhuyzen GD, Vigmond EJ. Fusion during Entrainment of Orthodromic Reciprocating Tachycardia Is Enhanced for Basal Pacing Sites but Diminished When Pacing near Purkinje System End Points. Heart Rhythm. 2013; 10(3):444–451. https://doi.org/10.1016/j.hrthm.2012.11.021

32. Hulsmans M, Clauss S, Xiao L, Aguirre AD, King KR, Hanley A, et al. Macrophages Facilitate Electrical Conduction in the Heart. Cell. 2017; 169(3):510–522.e20. https://doi.org/10.1016/j.cell.2017.03.050 PMID: 28431249

33. Winkelmann B, M Zgierski-Johnston C, Moritz Wülfers E, Timmer J, Seemann G. Computational Mechanistic Investigation of Chronotropic Effects on Murine Sinus Node Cells. In: Computing in Cardiology 2018. vol. 45. Maastricht, Netherlands; 2018.

34. Hucka M, Finney A, Sauro HM, Bolouri H, Doyle JC, Kitano H, et al. The Systems Biology Markup Language (SBML): A Medium for Representation and Exchange of Biochemical Network Models. Bioinformatics. 2003; 19(4):524–531. https://doi.org/10.1093/bioinformatics/btg015 PMID: 12611808

35. Cuellar AA, Lloyd CM, Nielsen PF, Bullivant DP, Nickerson DP, Hunter PJ. An Overview of CellML 1.1, a Biological Model Description Language. SIMULATION. 2003; 79(12):740–747. https://doi.org/10.1177/0037549703040939

36. Lederer W, Niggli E, Hadley R. Sodium-Calcium Exchange in Excitable Cells: Fuzzy Space. Science. 1990; 248(4953):283–283. https://doi.org/10.1126/science.2326638

37. Lloyd CM. Inada 2009—Physiome Model Repository; 2009. https://models.physiomeproject.org/workspace/inada_2009.

38. Schölzel C. CSchoel/Inamo: Release v1.4.3; 2021. Zenodo.

39. Mattsson SE, Elmqvist H. Modelica—An International Effort to Design the next Generation Modeling Language. In: 7th IFAC Symposium on Computer Aided Control Systems Design, CACSD'97. vol. 30. Gent, Belgium; 1997. p. 151–155.

40. Smith LP, Bergmann FT, Chandran D, Sauro HM. Antimony: A Modular Model Definition Language. Bioinformatics. 2009; 25(18):2452–2454. https://doi.org/10.1093/bioinformatics/btp401

41. Fritzson P, Aronsson P, Lundvall H, Nyström K, Pop A, Saldamli L, et al. The OpenModelica Modeling, Simulation, and Development Environment. In: Proceedings of the 46th Conference on Simulation and Modelling of the Scandinavian Simulation Society. Trondheim, Norway: Tapir Academic Press; 2005.

42. Justus N, Schölzel C, Dominik A, Letschert T. Mo—E—A Communication Service between Modelica Compilers and Text Editors. In: Proceedings of the 12th International Modelica Conference. Prague, Czech Republic: Linköping University Electronic Press, Linköpings universitet; 2017. p. 815–822.

43. Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: A Fresh Approach to Numerical Computing. SIAM Review. 2017; 59(1):65–98. https://doi.org/10.1137/141000671

44. Schölzel C. THM-MoTE/ModelicaScriptingTools.Jl: Release v1.1.0; 2021. Zenodo.

45. Chacon S, Long J, Git community. Git; 2021. https://git-scm.com/.

46. GitHub, Inc. GitHub; 2021. https://github.com/.

47. GitHub, Inc. Features—GitHub Actions; 2021. https://github.com/features/actions.

48. Python Software Foundation. Welcome to Python.Org; 2021. https://www.python.org/.

49. Hunter JD. Matplotlib: A 2D Graphics Environment. Computing in Science & Engineering. 2007; 9(3):90–95. https://doi.org/10.1109/MCSE.2007.55

50. Caswell TA, Droettboom M, Lee A, Hunter J, Firing E, Stansby D, et al. Matplotlib/Matplotlib v3.1.2; 2019. Zenodo.

51. Inkscape developers. Inkscape—Draw Freely; 2019. https://inkscape.org/.

52. Schölzel C. MoNK—A MOdelica iNKscape Extension; 2020. Zenodo.

53. Garny A, Hunter PJ. OpenCOR: A Modular and Interoperable Approach to Computational Biology. Frontiers in Physiology. 2015; 6:26. https://doi.org/10.3389/fphys.2015.00026

54. Kurata Y, Hisatome I, Imanishi S, Shibamoto T. Dynamical Description of Sinoatrial Node Pacemaking: Improved Mathematical Model for Primary Pacemaker Cell. American Journal of Physiology-Heart and Circulatory Physiology. 2002; 283(5):H2074–H2101. https://doi.org/10.1152/ajpheart.00900.2001

55. Zhang H, Holden AV, Kodama I, Honjo H, Lei M, Varghese T, et al. Mathematical Models of Action Potentials in the Periphery and Center of the Rabbit Sinoatrial Node. American Journal of Physiology-Heart and Circulatory Physiology. 2000; 279(1):H397–H421. https://doi.org/10.1152/ajpheart.2000.279.1.H397 PMID: 10899081

56. Dokos S, Celler B, Lovell N. Ion Currents Underlying Sinoatrial Node Pacemaker Activity: A New Single Cell Mathematical Model. Journal of Theoretical Biology. 1996; 181(3):245–272. https://doi.org/10.1006/jtbi.1996.0129

57. Matsuoka S, Hilgemann DW. Steady-State and Dynamic Properties of Cardiac Sodium-Calcium Exchange: Ion and Voltage Dependencies of the Transport Cycle. Journal of General Physiology. 1992; 100(6):963–1001. https://doi.org/10.1085/jgp.100.6.963

58. The Daring Fireball Company LLC. Daring Fireball: Markdown Syntax Documentation; 2021. https://daringfireball.net/projects/markdown/syntax.

59. Kumar A, Sjölund M, Tinnerholm J, Heuermann A, Pop A. OpenModelica/OMJulia.Jl: Julia Scripting OpenModelica Interface; 2021. https://github.com/OpenModelica/OMJulia.jl.

60. Brand S. Pace Layering: How Complex Systems Learn and Keep Learning. Journal of Design and Science. 2018; 3. https://doi.org/10.21428/7f2e5f08

61. GitHub, Inc. GitHub Archive Program; 2021. https://archiveprogram.github.com/.

62. Cern Data Centre. Zenodo; 2021. https://zenodo.org/.

63. Modelica Association. Tools—Functional Mock-Up Interface; 2021. https://fmi-standard.org/tools/.

64. Demir SS, Clark JW, Murphey CR, Giles WR. A Mathematical Model of a Rabbit Sinoatrial Node Cell. American Journal of Physiology-Cell Physiology. 1994; 266(3):C832–C852. https://doi.org/10.1152/ajpcell.1994.266.3.C832

65. Lindblad DS, Murphey CR, Clark JW, Giles WR. A Model of the Action Potential and Underlying Membrane Currents in a Rabbit Atrial Cell. American Journal of Physiology—Heart and Circulatory Physiology. 1996; 271(4):H1666–H1696. https://doi.org/10.1152/ajpheart.1996.271.4.H1666

66. Seidel H. Nonlinear Dynamics of Physiological Rhythms [PhD Thesis]. Technische Universität Berlin. Berlin, Germany; 1997.

67. Schölzel C, Goesmann A, Ernst G, Dominik A. Modeling Biology in Modelica: The Human Baroreflex. In: Proceedings of the 11th International Modelica Conference. Versailles, France; 2015. p. 367–376.

68. Jenkins Governance Board. Jenkins; 2021. https://www.jenkins.io/.

69. Moutsatsos IK, Hossain I, Agarinis C, Harbinski F, Abraham Y, Dobler L, et al. Jenkins-CI, an Open-Source Continuous Integration System, as a Scientific Data and Image-Processing Platform. SLAS DISCOVERY: Advancing the Science of Drug Discovery. 2017; 22(3):238–249. https://doi.org/10.1177/1087057116679993 PMID: 27899692

70. Wolstencroft K, Krebs O, Snoep JL, Stanford NJ, Bacall F, Golebiewski M, et al. FAIRDOMHub: A Repository and Collaboration Environment for Sharing Systems Biology Research. Nucleic Acids Research. 2017; 45(D1):D404–D407. https://doi.org/10.1093/nar/gkw1032 PMID: 27899646

71. Bornstein BJ, Keating SM, Jouraku A, Hucka M. LibSBML: An API Library for SBML. Bioinformatics. 2008; 24(6):880–881. https://doi.org/10.1093/bioinformatics/btn051

72. Bergmann F, Hucka M, Beber ME, Redestig H. Sbmlteam/Python-Libsbml: Python LibSBML 5.18.3; 2020. Zenodo.

73. Iravanian S, Rackauckas C. CellMLToolkit.Jl; 2020. https://github.com/SciML/CellMLToolkit.jl.

74. Bergmann F, Shapiro BE, Hucka M. SBML Software Matrix; 2021. http://sbml.org/SBML_Software_Guide/SBML_Software_Matrix.

75. Maggioli F, Mancini T, Tronci E. SBML2Modelica: Integrating Biochemical Models within Open-Standard Simulation Ecosystems. Bioinformatics. 2020; 36(7):2165–2172. https://doi.org/10.1093/bioinformatics/btz860

76. Atlassian. Bitbucket—The Git Solution for Professional Teams; 2021. https://bitbucket.org/product/.

77. GitLab. The First Single Application for the Entire DevOps Lifecycle—GitLab; 2021. https://about.gitlab.com/.

78. Wilkinson MD, Dumontier M, Aalbersberg IJ, Appleton G, Axton M, Baak A, et al. The FAIR Guiding Principles for Scientific Data Management and Stewardship. Scientific Data. 2016; 3:160018. https://doi.org/10.1038/sdata.2016.18 PMID: 26978244

79. Courtot M, Juty N, Knüpfer C, Waltemath D, Zhukova A, Dräger A, et al. Controlled Vocabularies and Semantics in Systems Biology. Molecular Systems Biology. 2011; 7:543. https://doi.org/10.1038/msb.2011.77 PMID: 22027554

8.4. Modeling biology in Modelica: The human baroreflex

Title	Modeling biology in Modelica: The human baroreflex
Authors	C. Schölzel, A. Goesmann, G. Ernst, and A. Dominik
Publication type	Conference proceedings (talk)
Conference	11th International Modelica Conference
Location	Versailles, France
Year	2015
Pages	367–376
DOI	10.3384/ecp15118367

Contributions by dissertation author (using the Contributor Roles Taxonomy):

Conceptualization	Formulated research question
Data curation	Maintained the code of the original SHM-C model
Investigation	Performed all experiments and analyses
Methodology	Defined metrics for comparison
Software	Implemented the Modelica model SHM-M
Visualization	Created all figures and tables
Writing — original draft	Wrote initial draft
Writing — review & editing	Incorporated reviews by other authors and reviewers

Modeling Biology in Modelica: The Human Baroreflex

Christopher Schölzel[1] Alexander Goesmann[2] Gernot Ernst[3] Andreas Dominik[1]

[1]KITE, Technische Hochschule Mittelhessen, Giessen, Germany,
{christopher.schoelzel,andreas.dominik}@mni.thm.de
[2]Justus Liebig University Giessen, Giessen, Germany
[3]Vestre Viken Hospital Trust, Kongsberg, Norway

Abstract

Systems biology is a field that requires complex multi-scale models of systems that are evolved rather than engineered. No unifying theory exists for biology as it does for engineering domains. Thus, models appear in very diverse forms. Components can be genes, cells, organs or even whole ecosystems. These components can intuitively be represented as classes in an object-oriented language, making systems biology a perfect application for Modelica. However, we still only see very few models from this domain. In an attempt to change this, we show that Modelica can exactly reproduce the simulation results of a reference implementation of an established biological model of the human baroreflex. Our implementation highlights the strengths of Modelica like the event finding mechanism, which makes the model more precise. We also show that biological systems pose interesting challenges like signals with non-uniform delays and the interaction of complex rhythms.
Keywords: systems biology, baroreflex, cardiovascular system, heart rate variability

1 Introduction

Biological systems are complex, dynamic and packed with feedback loops. Even small academic examples of systems that exhibit these properties are very hard to understand and analyze for humans without proper tools (Voit, 2013, pp. 8–10). Recent support comes in form of mathematical models, forming the field of systems biology. A strong mathematical foundation can help where intuition fails and indeed there are now projects such as the Virtual Liver Network (Holzhütter et al., 2012), the Blue Brain Project (Markram, 2006) or the Physiome Project (Hunter et al., 2002) that are on the way of building comprehensive multi-scale models for complete human organs.

In all of these projects, communication between models at different scales of time and space is a key challenge. Low-level models of biochemical reactions have to be integrated into models on the cellular level which then again need to be composed together to reach the desired level of abstraction. This is made more difficult by the fact that there exists no unifying theory in biology as it does in other domains such as electrical engineering (Voit, 2013, pp. 413–415). It is often not possible to build biological models as a bottom-up approach from the biochemical or genetic level, because too much is still unknown. For example, even when we narrow down the problem to these two lowest levels we still only begin to understand the role of long noncoding RNAs (Ponting et al., 2009) or glycoproteins (Ranzinger and York, 2012) in the regulation of cellular processes. Instead, researchers such as Denis Noble suggest a "middle-out" approach, that starts at the layer of abstraction where most experimental data is available (Noble, 2002).

There exist projects that aim to provide a common language for biological modeling, the most prominent being the Systems Biology Markup Language (SBML). SBML is an XML-based description format for models featuring the biochemical concept of different substances (called *species*) and *reactions* that modify the quantity of these substances. Support for the SBML is widespread in the systems biology community, but two main factors limit its usefulness for multi-scale modeling and the middle-out approach. Firstly, SBML is a data format and not a programming language. Mathematical formulas have to be specified in MathML which is not a human-readable format. Therefore, often multiple tools are needed for the generation and simulation of SBML models. Secondly, SBML is specifically designed for modeling biochemical processes which enforces a bottom-up approach and makes it impossible to start the modeling process at a higher level of abstraction. Other languages that are currently used in systems biology include numerical computing environments like Matlab and Mathematica, and general purpose programming languages like C. These languages all have sufficient expressive power and flexibility to start the modeling process at any layer of abstraction, but they also come with a lot of cognitive overhead. Scientists interested in modeling biological systems, such as physicians or biologists, have to build their equation systems and solve them by hand, or fit them

into a specific structure for existing implementations of the desired solver. As a result, the structure of the model is fitted to the programming platform instead of the other way around. It also makes the model harder to understand and discuss both for language experts that are not familiar with the modeled system and biologists that are not familiar with the specific language constructs. Especially with large equation systems, one has to put a lot of effort into structuring his code to preserve a clear distinction between different components of the modeled systems, let alone different abstraction layers of one and the same component.

Modelica offers an elegant solution for these problems. As an object-oriented language it makes the encapsulation of subsystems into larger components intuitive and thus highly facilitates multi-scale modeling. At the highest level, a Modelica model may present cells or entire organs as components with a clear interface that hides the details of the implementation at lower levels. If desired, however, a user of the model always has the possibility to inspect a component and dig one level deeper to look at the subsystems constituting the organ or cell. In theory, this makes it possible to build arbitrarily deep nested structures without overwhelming the model user with too much detail at a single level. Additionally, the declarative acausal nature of Modelica allows to write most formulas verbatim in the same way as they may appear in the mathematical definition of the model. This greatly reduces the cognitive overhead necessary to understand, maintain and extend an existing model.

Some projects already use Modelica for tasks related to systems biology. The BioChem library allows to translate SBML models to Modelica and vice versa offering a starting point for system biologists interested in Modelica (Nilsson and Fritzson, 2005). Additionally the Physiolibrary by Mateják et al. (2014) and the HumanLib by Brunberg and Abel (2010) are both targeting the modeling of the human physiology. Yet, when we looked at three recent systems biology textbooks, we did not find any reference to Modelica (Voit, 2013; Klipp et al., 2011; Kremling, 2012), although one of these books featured a list of over 80 tools for modeling in systems biology, including Matlab, Mathematica, SBML and a variety of application-specific SBML tools (Klipp et al., 2011).

In our opinion not only system biologists can benefit from Modelica, but also the field of systems biology provides interesting challenges for Modelica modelers. In contrast to most other application areas of Modelica, biological systems are evolved rather than engineered. Specifically, this means that there is usually a high level of complicated communication between multiple parts of the system and that these interactions are not always straightforward. One major example is the the "central dogma" of molecular biology. It states that every aspect of a living system can be explained starting from the DNA which is translated to proteins. These proteins then carry out some function in the system, but do not change the genetic code. This is a natural assumption that would seem intuitive to an engineer or computer scientist: A source code defines programs regulating the behavior of a system. However, the study of *epigenetics* shows that there are many factors that can influence gene expression and thus change the way in which the DNA-code is read (Holliday, 2006). To make things more difficult, there is no moment in the life of an organism where a cell is constructed from nothing but DNA. Even a single fertilised egg cell still has not only inherited the DNA from its parents but also all of the other biochemical substances in this cell.

Therefore, when we build models of biological systems, we might encounter unusual connection patterns between components. These new types of problems may indicate areas, where Modelica still can be improved. As an additional argument for the study of biological models, the field of systems biology spans a large area of interesting and relevant topics, from the modeling of brain activity to finding diagnosis criteria and treatments for cardiac diseases, diabetes or cancer to the modeling of whole ecosystems like oceans (Voit, 2013, pp. 399–415). The potential of applying mathematical modeling to biological systems is vast, and with Modelica we can facilitate the generation of new insights.

With this paper we want to take a further step towards bringing together Modelica modelers and systems biologists. We show that it is not only possible to convert SBML models to Modelica with the BioChem library or build physiological models from predefined components with the Physiolibrary, but also to implement a biological model in Modelica directly from the mathematical description. As a proof of concept, we therefore implemented an established model of the human baroreflex by Seidel (1997) in Modelica and compared it with the original implementation of Seidel written in C. An introduction to the Seidel-Herzel model (SHM) will be given in section 2. The two implementations of the SHM – hereinafter called SHM-M for our Modelica version and SHM-C for Seidels C implementation – will both be described in section 3. To demonstrate that Modelica is indeed flexible enough to precisely reflect the mathematical description of the model, we directly compare the output of SHM-C and SHM-M in section 4 followed by a discussion of the results in section 5. There we also demonstrate that biological models like the SHM fit nicely into the modeling paradigms of Modelica and highlight some interesting challenges of the implementation process. Finally, a short conclusion can be found in section 6.

2 The Seidel-Herzel model

The Seidel-Herzel model (SHM) is a model of the human baroreflex that was created by Henrik Seidel and Hanspeter Herzel and first published in 1995 with the

DOI
10.3384/ecp15118367

main purpose of analyzing heart rate variability (HRV) (Ernst, 2014). There are two main reasons why we chose this model as example for a typical biological system that can be implemented in Modelica: Firstly, although the cardiovascular system is well researched, heart diseases are still the most common cause of death worldwide (Naghavi et al., 2014). This means that models like the SHM are still relevant and may even help towards finding diagnostic criteria for heart-related diseases. The second reason to choose the SHM over other models of the human heart was that it is rather compact but still covers the effects of multiple organs. The doctorate thesis of Seidel contains the most recent version of the model, which has 24 equations and 52 parameters (Seidel, 1997).

Although there is another publication by Seidel *et al.* from 1998 (Seidel and Herzel, 1998), we used this version because it features several improvements of the model that were not present in the journal publication. In the following, we will give a short introduction into the components of the SHM and also explain its relevance in literature. For a more detailed explanation of the formulas, the reader is referred to Seidel's doctorate thesis (Seidel, 1997). This version of the SHM considers the physiological effects of the baroreceptors in the blood vessels, the autonomic nervous system, the lung, the sinus- and av-nodes, the heart itself and the Windkessel arteries. It does not introduce different compartments in the blood system but instead models the arterial blood pressure as a single physical quantity.

2.1 Baroreceptors

Baroreceptors are the sensory neurons measuring the pressure in a blood vessel. The basic neural firing frequency of the baroreceptors ν_b in the SHM is calculated with the following formula.

$$\nu_b'(t) = p - p_b^{(0)} + k_b^{dp}\frac{dp}{dt} \tag{1}$$

This includes the effects that baroreceptors respond to the static blood pressure level p as well as to an increase or decrease in blood pressure and that they only respond to blood pressure levels above a threshold $p_b^{(0)}$. The parameter k_b^{dp} is a scaling factor to adjust the relative influence of the blood pressure slope.

To account for the saturation effect of baroreceptors, this value is passed through the saturation function

$$\nu_b(t) = p_b^{c0}\left(1+\tanh\left(\frac{\nu_b'(t)-p_b^{c0}}{p_b^{c0}}\right)\right) \tag{2}$$

$$p_b^{c0} = p_b^c - p_b^{(0)} \tag{3}$$

where p_b^c is a scaling parameter to adjust the maximum of the saturation function, which lies at $2p_b^{c0}$.

In a living organism, however, the signal of baroreceptors at different parts of the body reach the autonomic nervous system (ANS) at different time instants. This effect is modeled in the SHM with a broadening function that is additionally applied to the saturated baroreceptor response.

$$\tilde{\nu}_b(t) = \int_{-\infty}^{\infty} g(t-\tau)\,\nu_b(\tau)\,d\tau \tag{4}$$

$$g(t) = \begin{cases} 0 & \text{for } t \le 0 \\ \frac{1}{\sigma}\chi^2_{2+\frac{\eta}{\sigma}}\left(\frac{t}{\sigma}\right) & \text{for } t > 0 \end{cases} \tag{5}$$

In this equation χ_n^2 is the probability distribution function of the chi squared distribution and σ and η are scaling parameters to adjust the broadening range.

2.2 Lung

The Lung influences the heart rate both through neural signals and the mechanical pressure in the thorax. The SHM assumes a constant breathing rate that is only modified by a noise term. The activity of respiratory neurons $\nu_r(t)$ is given by

$$\nu_r(t) = \frac{1}{2}(1-\sin(2\pi\phi_r(t))) \tag{6}$$

$$\phi_r(t) = \frac{t-(t_{r,i}-\theta_r)}{T_{r,i}} \tag{7}$$

where $t_{r,i}$ is the beginning of the last inspiration phase and θ_r is a phase shift parameter that determines the time between the firing of respiratory neurons and the actual mechanical movement of the lungs.

The mechanical respiratory influence $f_m(t)$ is defined similarly by the following equation.

$$f_m(t) = -\sin(2\pi\phi_r(t-\theta_r)) \tag{8}$$

Even during voluntarily controlled breathing, the breathing rate of a human is always subject to fluctuations. Seidel models these fluctuations by introducing a noise term which is applied to the mean breathing rate $\bar{T}_r$ at each breathing cycle with an autoregressive function.

$$T_{r,i} = k_{T_r}\bar{T}_r + k_{T_r}^{\text{last},1}T_{r,i-1} + k_{T_r}^{\text{last},2}T_{r,i-2} + \sigma_{T_r}\xi \tag{9}$$

$$k_{T_r} = (1-k_{T_r}^{\text{last},1}-k_{T_r}^{\text{last},2}) \tag{10}$$

In this formula, $T_{r,i}$ is the breathing period at the ith breathing cycle; $k_{T_r}^{\text{last},1}$ and $k_{T_r}^{\text{last},2}$ are parameters that determine the influence of the last and second last breathing period on the current period; and σ_{T_r} is the amplitude of the white noise ξ.

2.3 Autonomic Nervous System

The autonomic nervous system (ANS) consists of the sympathetic and the parasympathetic system. The sympathetic system increases the heartbeat frequency through the release of norepinephrine as neurotransmitter (via synapses) and as Hormone (via the blood vessels). The parasympathetic system has inhibitory influence on the heartbeat frequency through the release of the neurotransmitter acetylcholine.

The formulas for the neural activity of the sympathetic system ν_s and the parasympathetic system ν_p therefore only differ by the sign with which the baroreceptor activity enters the equation.

$$\nu_s'(t) = \nu_s^{(0)} - \left(1 + k_s^{br}\nu_r(t)\right)\tilde{\nu}_b(t) + k_s^r\nu_r(t) \quad (11)$$

$$\nu_p'(t) = \nu_p^{(0)} + \left(1 + k_p^{br}\nu_r(t)\right)\tilde{\nu}_b(t) + k_p^r\nu_r(t) \quad (12)$$

$$\nu_s(t) = \max(0, \nu_s'(t)) \quad (13)$$

$$\nu_p(t) = \max(0, \nu_p'(t)) \quad (14)$$

Both equations have a base firing rate $\nu_{s/p}^{(0)}$ and scaling parameters $k_{s/p}^r$ for the respiratory influence and $k_{s/p}^{br}$ for the correlation between the activity of the baroreceptors and the respiratory neurons.

2.4 Substance Concentrations

The SHM models several concentrations of neurotransmitters and hormones. The concentration of norepinephrine at the sinus node (c_{sNe}) directly influences the pacemaker phase together with the concentration of acetylcholine (c_{sAc}) at the sinus node. Additionally, norepinephrine can also act as a hormone. The ventricular concentration c_{vNe} in the heart itself increases the contractility (force of the contraction). The concentration in the Windkessel arteries c_{wNe} increases the stiffness of the vessel walls, resulting in a higher blood pressure during the diastole.

The release of this concentration is triggered by one neural signal and can be inhibited by another neural signal. For norepinephrine the excitatory signal comes from the sympathetic system while the parasympathetic system inhibits the release. For acetylcholine the parasympathetic system is the excitatory part and there is no inhibition modeled. Both inhibitory and excitatory signals only take effect after a delay θ_c^x and are subject to saturation. This leads us to the following equations

$$\mathrm{ex}_c^x(t) = \mathrm{in}_c^x(t) = \tanh(k_c^x\,\nu_x(t - \theta_c^x)) \quad (15)$$

$$\tau_{sNe}\frac{dc_{sNe}}{dt} = -c_{sNe}(t) + \mathrm{ex}_{sNe}^s(t)\,(1 - \mathrm{in}_{sNe}^p(t)) \quad (16)$$

$$\tau_{vNe}\frac{dc_{vNe}}{dt} = -c_{vNe}(t) + \mathrm{ex}_{vNe}^s(t)\,(1 - \mathrm{in}_{vNe}^p(t)) \quad (17)$$

$$\tau_{wNe}\frac{dc_{wNe}}{dt} = -c_{wNe}(t) + \mathrm{ex}_{wNe}^s(t)\,(1 - \mathrm{in}_{wNe}^p(t)) \quad (18)$$

$$\tau_{sAc}\frac{dc_{sAc}}{dt} = -c_{sAc}(t) + \mathrm{ex}_{sAc}^p(t) \quad (19)$$

where the τ_c and the k_c^x are scaling parameters for the overall slope of the concentrations $c_c(t)$ and the influence of the inhibitory or excitatory signal.

2.5 Sinus Node

The sinus node is the main pacemaker of the heart. In the SHM it is modeled with the pacemaker phase $\phi(t)$ which generates a sinus signal when its value becomes one and is then directly reset to zero. The rate of the pacemaker phase increases with an increased concentration of norepinephrine and decreases with an increase in acetylcholine. The latter is additionally modified by a "phase-effectiveness curve" $F(\phi)$, because the effect of the parasympathetic signal on the pacemaker changes with the phase of the heart cycle. The resulting behavior is given by the following formula

$$\frac{d\phi}{dt} = \frac{1}{T^{(0)}}\left(1 + k_\phi^{sNe}c_{sNe}(t) - k_\phi^{sAc}c_{sAc}(t)\frac{F(\phi(t))}{\bar{F}_\phi}\right) \quad (20)$$

$$\bar{F}_\phi = \int_0^1 F(\phi)d\phi \quad (21)$$

$$F(\phi) = \phi^{1.3}(\phi - 0.45)\frac{(1-\phi)^3}{(1-0.8)^3 + (1-\phi)^3} \quad (22)$$

where $T^{(0)}$ is the duration of the heart cycle without any input from the ANS and k_ϕ^{sNe} and k_ϕ^{sAc} are scaling parameters for the influence of the concentrations of norepinephrine and acetylcholine.

The heart period of a human is always subject to additional fluctuations that do not originate from breathing or the signals of the ANS. These influences can be implemented by replacing the parameter $T^{(0)}$ with a noisy base period $T_n^{(0)}$, which varies with the heartbeat number n similarly to the respiratory base period in Equation 9.

$$T_n^{(0)} = \bar{T}^{(0)} + k_{T^{(0)}}^{\mathrm{last}}(T_{n-1}^{(0)} - \bar{T}^{(0)}) + \sigma_{T^{(0)}}\xi \quad (23)$$

2.6 Contraction Model

Not every sinus signal generates a heartbeat and not every heartbeat is triggered by a sinus signal. On the one hand, if there is no sinus signal for a prolonged time period, the atrioventricular node (AV node) will trigger a

DOI 10.3384/ecp15118367

contraction by itself. This is represented by the parameter T_{av} so that a contraction is triggered if more than T_{av} seconds have passed since the last contraction at time $t_{c,n}$. On the other hand, a sinus signal does not immediately correspond to the beginning of a contraction. There is an atrioventricular concudtion delay $T_{avc,n}$ that passes from the firing of the sinus node (which is located at the atrium) at time $t_{s,n}$ to the contraction of the ventricles. It depends on the time that has passed since the last contraction at $t_{c,n}$. Additionally, the sinus node has a refractory period T_{refrac} during which no new signal may be generated. Combining these two effects, we receive the following equation for atrioventricular conduction time.

$$T_{avc,n} = \begin{cases} T_{avc}^{(0)} + k_{avc}^{t} e^{-\frac{(t_{s,n}-t_{c,n})}{\tau_{avc}}} & \text{if } t_{s,n} - t_{c,n} > T_{\text{refrac}} \\ \infty & \text{else} \end{cases} \tag{24}$$

In this equation $T_{avc}^{(0)}$ is the base value for the atrioventricular conduction time, k_{avc}^{t} is a scaling parameter for the influence of the time that has passed since the last contraction and τ_{avc} is a reference value for the atrioventricular conduction time.

2.7 Heart

The final components of the SHM are the contraction of the heart and the Windkessel arteries that are responsible for the blood pressure increasing during the systole and decreasing during the diastole. The switch between systole and diastole is modeled explicitly by a fixed systole duration of τ_{sys}. During the systole from $t_{c,n}$ to $t_{c,n} + \tau_{\text{sys}}$ the blood pressure follows the equations

$$\frac{dp}{dt} = \frac{1}{\tau_{\text{sys}}} \frac{S_n}{C} \left(1 - \phi_{\text{sys}}(t)\right) e^{1-\phi_{\text{sys}}(t)} \tag{25}$$

$$\phi_{\text{sys}}(t) = \frac{t - t_{c,n}}{\tau_{\text{sys}}} \tag{26}$$

where C is a scaling constant for the contractility S_n. The value of S_n is determined at the beginning of the systole at $t_{c,n}$ as follows.

$$S_n = S^{(0)} + \left(k_S^{vNe} c_{vNe}(t_{c,n}) + k_S^{m} f_m(t_{c,n})\right) S^{(T_{n-1})} \tag{27}$$

$$S^{(T_{n-1})} = \left(1 - \left(1 - \min\left(1, \frac{t_{c,n} - t_{c,n-1}}{\hat{T}}\right)\right)^2\right) \tag{28}$$

During the diastole, the equation for the blood pressure switches to the following formula that accounts for the effect of the Windkessel arteries. These arteries directly connected to the heart are elastic and act as a dampening system. During the systole they "store" blood by expanding the blood vessels. During the diastole they contract back to their original state slowly releasing the stored blood.

$$\frac{dp}{dt} = \frac{-\left(p - p_w^{(0)}\right)}{\tau_w(t)} \tag{29}$$

$$\tau_w(t) = \tau_w^{(0)} + k_w^{wNe} c_{wNe}(t) \tag{30}$$

In this formula $p_w^{(0)}$ is the minimum blood pressure that is still present even if the Windkessel arteries are fully relaxed and the heart does not pump, $\tau_w^{(0)}$ is a base value for the time needed for the Windkessel arteries to fully relax, and k_w^{wNe} is a scaling factor for the influence of the norepinephrine concentration in the arteries on this relaxation time.

2.8 Physiological Relevance

The SHM is able to reproduce several characteristics of complex heart rate dynamics. The first and most obvious effect are fluctuations of the heart rate with the frequency of breathing cycles called respiratory sinus arrhythmia (RSA). This behavior is not surprising as it is directly built into the model with the definition of ν_r. A more interesting observation is that the model also exhibits fluctuations with a period of approximately 10 seconds, which also corresponds to a physiological phenomenon called Mayer waves (Seidel and Herzel, 1995). When investigating the reaction of the model to changes in parameter values, Seidel and Herzel (1998) also found bifurcations related to the sympathetic and parasympathetic delays, the baroreceptor sensitivity and repetitive vagal stimulation. The observed dynamical properties were in good agreement with patients with baroreceptor hypersensitivity and animal experiments. However, results by Duggento et al. (2012) show that these bifurcations can actually be triggered by most parameters of the model. They suggest that the model should be reparameterized to make all modeled variables physiologically plausible, and assume that this could lead to a "'unifying theory to account for slow oscillation' in cardiovascular variability".

Kotani et al. (2002) extended the model by noise and more detailed respiratory influences. They showed that the model can explain the synchronization between the heartbeat and the breathing frequency observed in humans (Kotani et al., 2002). In a later study they also found that the (modified) SHM could produce statistically valid simulations of congestive heart failure and primary autonomic failure – diseases that are known to affect the parasympathetic and sympathetic neural activity (Kotani et al., 2005).

To sum up, we can say that the SHM is able to produce physiologically plausible simulations of character-

istics of both healthy patients and several disease conditions. Its potential may not have been fully exploited yet, making our implementation a possible start for future investigations.

3 Implementations and Simulation Setup

Thanks to Henrik Seidel we were able to use his original implementation (SHM-C) as reference for our Modelica implementation (SHM-M). It is written in C and uses a self-implemented Runge-Kutta Method for solving the differential equations. The executable allows to set starting values and parameters with a parameter file and writes the simulation result to a CSV-file.

With our Modelica-version of the Seidel-Herzel-model we wanted to reproduce the output of SHM-C as closely as possible, while still retaining the object-oriented implementation style of Modelica. We divided the formulas of the SHM according to the physiological parts they represent to obtain small Modelica components.

On the highest level SHM-M consists of the models `Baroreceptors`, `SinusNode`, `Heart`, `Lung`, `SympatheticSystem`, `ParasympatheticSystem`, `HormoneRelease` and `NeurotransmitterRelease` as well as the compartment models `BloodSystem`, `HormoneAmount` and `NeurotransmitterAmount`. These models are the entry points for users of SHM-M. Therefore, we kept them as simple as possible by encapsulating the broadening and saturation of the baroreceptors, the contraction model of the heart and the phase effectiveness function into the separate classes `Broaden`, `TanhSaturation`, `Contraction`, and `PhaseEffectiveness`. Additionally, due to the similarities in equation 11 and 12 and equations 16–19, the models `SympatheticSystem` and `ParasympatheticSystem` share a common base class `ANSPart` and the models `HormoneRelease` and `NeurotransmitterRelease` are even functionally equivalent to their base class `SubstanceRelease` only providing different icons. A diagram of the full model can be seen in Figure 1.

Most of the connections between models in the SHM-M are implemented as a set of causal input-/output-connectors. The neural signals and the mechanical respiratory influence as well as the boolean trigger output of the sinus node all have a clear physiological direction. The substance concentrations and the blood flow, however, are implemented using acausal connectors with flow variables. Actually this is not a mathematical requirement, because both the substance concentrations and the blood pressure are defined by a single equation in only one component that could also be implemented with causal connectors. However, physiologically the release and uptake of substances are separate processes and the blood pressure is influenced by both the Windkessel arteries and the heart itself. For a more realistic representation of these physiological properties, future versions of the model should therefore also separate these effects mathematically, which will be easier to implement using flow variables in the connectors.

With this structure, the implementation could mostly be achieved by just transferring the mathematical formulas directly to Modelica notation. Where approximations of the mathematical definition were necessary – namely the broadening of the baroreceptor response with a Green's function – the same numerical algorithm used in SHM-C was implemented as a function in Modelica.

The only components where a straightforward implementation was not possible are the submodels `Contraction` and `Broaden`. The model `Contraction` captures the interplay of the sinus signal, refractory period and AV node and thus features rather complicated expressions in when-conditions involving discrete variables. In the current stable version of the OpenModelica compiler (Version 1.9.1) these discrete equation systems are not supported. We therefore had to implement the contraction signal using continuous variables in the when-condition as the following code snippet shows.

```
when sinus_phase < 1e-10 and signal
     and refrac_countdown <= 0 then
  T_avc = T_avc0 +
          k_av_t * exp(-T_passed/tau_av);
  contraction = sinus_phase > 1 or
                av_phase > 1;
end when;
```

This introduced several additional phase variables for the atrioventricular conduction time, the refractory period and the period of the AV node.

The major challenge regarding the model `Broad` is the implementation of the convolution in Equation 4. We found no better way to calculate this convolution than to build an array with delay expressions of the baroreceptor signal with the following loop:

```
for i in 1:size(hist,1) loop
  hist[i] = delay(x,(i-1)*step,(i-1)*step);
end for;
```

This implementation works as desired for small broadening lengths, but becomes extremely slow for larger values.

As a final difference between SHM-M and SHM-C we did not implement the noise model for the breathing frequency and the heartbeat duration, because this noise would only complicate the comparison of the two models. Instead, to obtain comparable data, noise was also disabled in SHM-C through parameter settings.

DOI 10.3384/ecp15118367

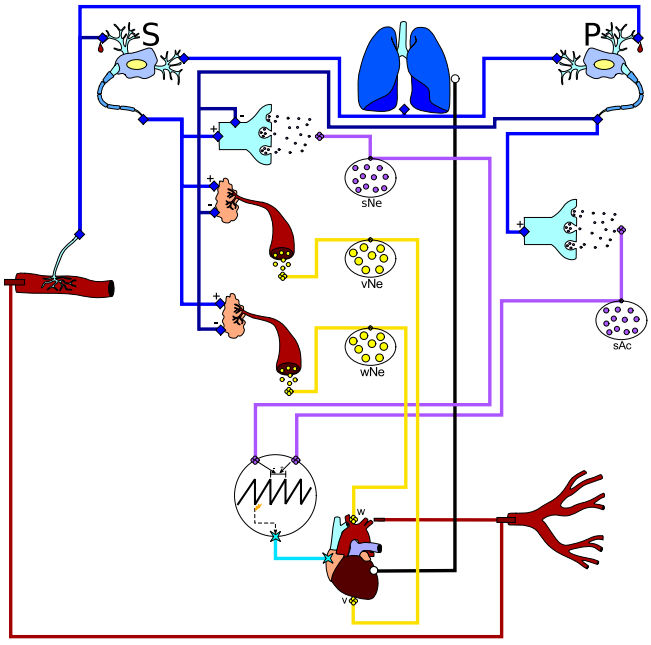


Figure 1. Diagram view of the SHM-M implementation showing the components of the model.

The simulation of our model was performed using OpenModelica. The Runge-Kutta method implemented in OpenModelica uses the same classical RK4-Parameters that are also used in SHM-C. Therefore we only had to use the same step size and the same set of starting values and parameters for the simulation to obtain directly comparable results.

We used the standard parameter set by Seidel and only adjusted the manually defined starting values in SHM-C to match the starting values of SHM-M that were calculated by OpenModelica. The resulting parameter configuration can be seen in Table 1. We then did a simulation with both models for 1000 seconds with a step size of one millisecond and recoded the important values in steps of ten milliseconds as well as the duration and end time for each heartbeat.

4 Results

The SHM was designed for the analysis of heart rate variability. Therefore, to compare different implementations it is most important to look at the duration of heartbeats and the blood pressure. A direct comparison of the time series for these physical quantities can be seen in Figure 2. For the blood pressure both curves have no visual differences until 5 seconds after the start of the simulation, when SHM-M starts to run ahead slightly. At the end of the simulation, the situation is similar: The only difference between the curves seems to be a time shift. For the heartbeat duration the differences are already noticeable at the first heart beat, which is 3 milliseconds longer in SHM-M than in SHM-C.

To better quantify these differences, we plotted the

Table 1. Parameter and initial values used for the comparison of SHM-C and SHM-M.

Parameter	*Value*	*Parameter*	*Value*
Baroreceptors		*Sinus node*	
$p_b^{(0)}$	60	$T^{(0)}$	0.9
k_b^{dp}	0.06	k_ϕ^{sNe}	0.6
$p_b^{(c)}$	120	k_ϕ^{sAc}	0.2
σ	0.001	*Contraction*	
η	0.01	$T_{avc}^{(0)}$	0.09
Lung		k_{avc}^{t}	0.78
$\theta_r^{(0)}$	0.16	τ_{avc}	0.11
$\bar{T}_r$	4	T_{refrac}	0.22
$k_{T_r}^{\text{last},1}$	0	T_{av}	1.7
$k_{T_r}^{\text{last},2}$	0	*Heart*	
σ_{T_r}	0	τ_{sys}	0.125
ANS		C	2
$v_s^{(0)}$	50	$S^{(0)}$	110
k_s^{br}	0.38	k_S^{vNe}	110
v_s^r	30	k_S^m	0
$v_p^{(0)}$	10	$p_w^{(0)}$	0
k_p^{br}	0.38	$\tau_w^{(0)}$	1.3
v_p^r	30	k_w^{rNe}	0.8
Concentrations		$\hat{T}$	1
k_{sNe}^s	0.014	*Initialization*	
θ_{sNe}^s	2	$p(0)$	100
k_{sNe}^p	0.006	$c_{sNe}(0)$	0.12
θ_{sNe}^p	0.4	$c_{sAc}(0)$	0.5
τ_{sNe}	2	$c_{vNe}(0)$	0.12
k_{vNe}^s	0.014	$c_{rNe}(0)$	0.12
θ_{vNe}^s	2	T_0	1
k_{vNe}^p	0.006	$t_{c,0}$	-1
θ_{vNe}^p	0.4	$T_{avc}(0)$	0.15
τ_{vNe}	4	S_0	110
k_{rNe}^s	0.014		
θ_{rNe}^s	3		
k_{rNe}^p	0		
θ_{rNe}^p	0.4		
τ_{rNe}	4		
k_{sAc}^p	0.005		
θ_{sAc}^p	0.4		
τ_{sAc}	0.05		

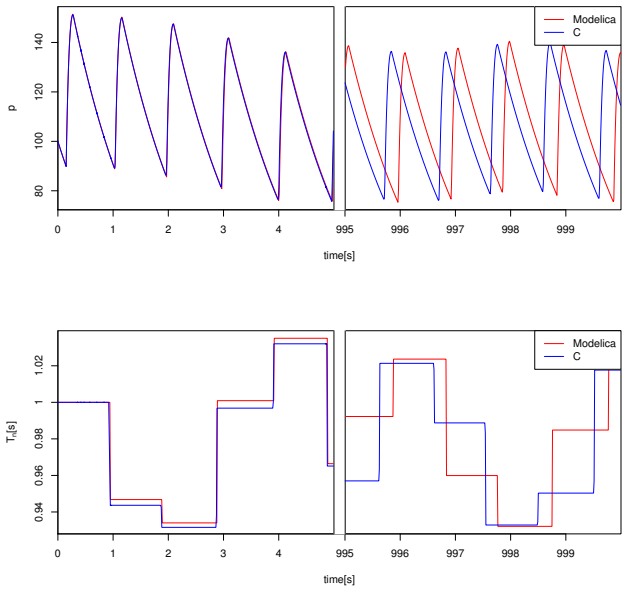


Figure 2. Comparison of time series of blood pressure p and heartbeat duration T_n between SHM-C (blue) and SHM-M (red) for a simulation time of 1000 seconds, showing seconds 0 to 5 and seconds 995 to 1000.

difference between heartbeat durations in SHM-M and SHM-C against the standard deviation between heartbeat durations in SHM-C. The result can be seen in Figure 3. It turns out that the duration of the first 40 heartbeats only differs by less than 10 milliseconds with a standard deviation of 34 milliseconds. The plot also shows that on average SHM-M produces heartbeat periods that are 3 milliseconds longer.

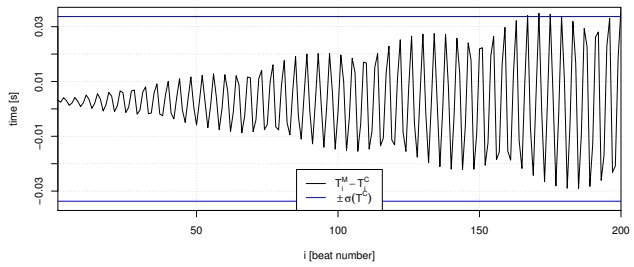


Figure 3. Difference between RR-Intervals of SHM-M and SHM-C relative to the standard deviation of RR-Intervals in SHM-C. Values above zero represent RR-Intervals that are longer in SHM-M compared to SHM-C.

While absolute differences can give an impression of the size of possible calculation errors, for the SHM it is much more interesting to look at the long-time behavior of the model. Seidel used a plot of the spectral density of RR-Intervals (i. e. heartbeat durations) as one of his main arguments for the physiological plausibility of his model. We therefore also compared SHM-C and SHM-M on the frequency domain. The result can be seen in Figure 4. The plot shows a clear peak identical in magnitude and position at approximately 0.25 Hz, which corresponds to the breathing frequency and can thus be thought to represent respiratory sinus arrhythmia. We can also see another less pronounced peak for both implementations at approximately 0.1 Hz which Seidel attributes to Mayer waves. However, in SHM-M the peak at 0.25 Hz is much sharper than in SHM-C.

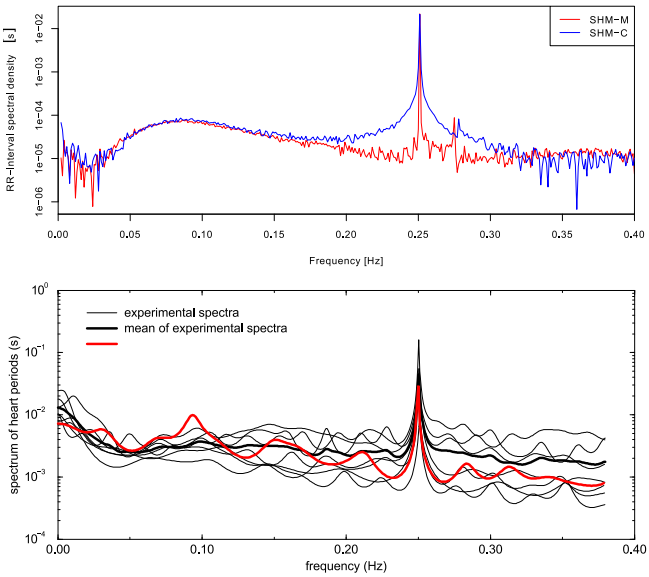


Figure 4. Top: Comparison of spectral density of RR-Intervals in SHM-C (blue) and SHM-M (red) after a simulation for 1000 seconds with standard parameters. Bottom: Figure from Seidel's doctorate thesis comparing spectral density of RR-Intervals of SHM with noise (red) and experimental data (black) (Seidel, 1997).

5 Discussion

The comparison between both implementations SHM-M and SHM-C shows that our proof of concept was successful. At the beginning of the simulation, the blood pressure stays almost the same. There are noticeable differences in the heartbeat duration, but these are not unexpected as the event finding mechanism of Modelica can determine the exact time at which an event occurs more precisely compared to the simple check after each Runge-Kutta step implemented in SHM-C. This is also consistent with the sharper peak in the frequency domain observed in Figure 4. Experimental data with voluntarily controlled breathing actually shows a rather smooth RSA-related peak in the frequency domain similar to SHM-C. However, the reason for this is that the subjects naturally cannot time their breathing to the exact millisecond, introducing a noise to the breathing frequency. This type of noise has been incorporated into the model by Seidel and it can also be incorporated in SHM-M. Our model therefore may allow a more precise

DOI 10.3384/ecp15118367

analysis of the theoretical effects of RSA without sacrificing realism.

Now that we have seen that SHM-M is able to reproduce the simulation results of SHM-C we can ask what makes the new implementation (or biological models in general) interesting from a Modelica perspective. First of all, the notion of a system composed of multiple organs fits nicely into the object-oriented paradigm and leads to a natural and intuitive class structure. In fact, each subsection in the explanation of the SHM in section 2 corresponds exactly to a Modelica model in SHM-M. The model structure directly reflects the structure of the real world system and thus makes the model very explainable and accessible for domain experts. Furthermore, encapsulation, inheritance and the reuse of objects instantiated with different parameters could be applied to yield a hierarchical structure that hides implementation complexities and avoids code repetition. The idea of hierarchically structured taxonomies is deeply rooted in biology. It therefore seems reasonable to expect that most biological models can be implemented in such a clean and intuitive manner using Modelicas object-oriented approach.

Additionally, due to the tendency of biological systems to feature multiple complex rhythms, these models can showcase the strength of Modelica's event finding mechanism in comparison to a naive implementation of the Runge-Kutta- or Euler-method or other tools that mainly focus on continuous modeling.

These complex rhythms also turned out to be one of the two major challenges that arose during the implementation of SHM-M. As already mentioned a straightforward implementation of the contraction model was not possible in OpenModelica, because the nontrivial conditions in the when-equations formed a discrete equation system. We did not test the model with other compilers, so this may be only an issue with OpenModelica, but nonetheless our biological model requires a feature that seems not as crucial for most other application areas of Modelica.

The second major implementation challenge was the broadening function used for the baroreceptors. To assess the performance issues with the implementation using direct delay equations, we recorded the time taken for a simulation over 1000 seconds for both SHM-M and SHM-C with broadening lengths ranging from 0.1 seconds to 3 seconds. We found that simulation times of SHM-M rise linearly from 75 seconds with a broadening length of 0.1 seconds to as much as one hour for a broadening length of 3 seconds. In contrast, SHM-C only shows an increase from 16.6 to 21.3 seconds respectively. This results suggest that OpenModelica uses a separate history buffer for each delay equation in the loop. If this is the case, an increase of the broadening length by only one simulation step would require the allocation and management of an additional buffer of the same size as the single history buffer used in SHM-C, explaining the additional overhead. We are not aware of a language construct that allows to indicate that the delay equations in the loop may share the same buffer. Building the buffer manually in Modelica is also not possible, because the language itself has no notion of discrete simulation steps. This performance issue is therefore hard to fix as a Modelica programmer and would possibly be an argument to include convolutions as a language element.

We can therefore conclude that the SHM, as a model that exhibits the typical properties of biological models in general, does not only fit nicely into the modeling style of Modelica but also has some challenging aspects that point to possible areas of improvement for OpenModelica or Modelica in general. This suggests that biological models could indeed become a new and interesting application area for Modelica.

6 Conclusion

With our implementation of the SHM we demonstrated as a proof of concept that Modelica is perfectly suited for the implementation of biological systems in a natural representation. The language can directly reflect the biological composition of the system instead of having to fit the system into the language constructs. This is shown by the fact that our Modelica version of the SHM – which uses a lot of the features of Modelica such as acausal declarations, encapsulation and component reuse through instantiation and inheritance – can reproduce the same behavior as the original reference implementation in C. Moreover, it is in some parts even more precise thanks to the event finding mechanisms of Modelica.

We also demonstrated that new challenges that are not present in other domains may arise when we model evolved rather than engineered systems. The interaction of complex rhythms together with time-varying signal conduction delays lead to a contraction model that could not be intuitively implemented with the current version of the OpenModelica compiler. Additionally, implementing the behavior that the baroreceptor signal reaches the ANS through many different nervous connections with varying delays required a convolution that seems to be a performance bottleneck.

These are strong arguments both for biologists to choose Modelica over a general purpose programming language and for Modelica modelers to look for interesting applications and models in the systems biology domain. This paper laid the ground for the implementation of more biological models from the side of the Modelica community, but to encourage interdisciplinary research we also have to take the opposite perspective. We need to investigate the benefits of Modelica more closely in regard to the needs of systems biologists. A first step could be to reparameterize the SHM as suggested by Duggento et al. (2012) and to incorporate additional components that can simulate vagal stimulation and different disease conditions (which is possible but

cumbersome with the C implementation). We also plan to extend the SHM to a multi-scale model, for example by exchanging the heart model with a more detailed representation modeling individual heart cells. Finally, it would be interesting to embed the model into the Physiolibrary to provide a single point-of-entry for biologists and physicians interested in physiological modeling with Modelica. We believe that there is a lot of potential for interesting projects involving biological models in Modelica and we are looking forward to seeing more of them in the future.

References

Anja Brunberg and Dirk Abel. Simulation verkoppelter physiologischer Regelkreise mit Hilfe der objektorientierten Modellbibliothek „HumanLib“. In *Automatisierungstechnische Verfahren für die Medizin, 9. Workshop, Tagungsband*, pages 29–30, Zürich, Switzerland, 2010. doi:10.1524/auto.2011.0951.

Andrea Duggento, Nicola Toschi, and Maria Guerrisi. Modeling of human baroreflex: Considerations on the Seidel–Herzel model. *Fluctuation and Noise Letters*, 11(1): 1240017, 2012. doi:10.1142/S0219477512400172.

Gernot Ernst. *Heart rate variability*. Springer-Verlag, London, England, 2014. ISBN 978-1-4471-4308-6. doi:10.1007/978-1-4471-4309-3.

Robin Holliday. Epigenetics: A historical overview. *Epigenetics*, 1(2):76–80, 2006. doi:10.4161/epi.1.2.2762.

Hermann-Georg Holzhütter, Dirk Drasdo, Tobias Preusser, Jörg Lippert, and Adriano M. Henney. The virtual liver: A multidisciplinary, multilevel challenge for systems biology. *Wiley Interdisciplinary Reviews: Systems Biology and Medicine*, 4(3):221–235, 2012. doi:10.1002/wsbm.1158.

Peter Hunter, Peter Robbins, and Denis Noble. The IUPS human physiome project. *Pflügers Archiv - European Journal of Physiology*, 445(1):1–9, 2002. doi:10.1007/s00424-002-0890-1.

Edda Klipp, Wolfram Liebermeister, Christoph Wierling, Axel Kowald, Hans Lehrach, and Ralf Herwig. *Systems biology: A textbook*. Wiley-Blackwell, Hoboken, New Jersey, 2011.

Kiyoshi Kotani, Kiyoshi Takamasu, Yosef Ashkenazy, H. Stanley, and Yoshiharu Yamamoto. Model for cardiorespiratory synchronization in humans. *Physical Review E*, 65(5): 051923, 2002. doi:10.1103/PhysRevE.65.051923.

Kiyoshi Kotani, Zbigniew Struzik, Kiyoshi Takamasu, H. Stanley, and Yoshiharu Yamamoto. Model for complex heart rate dynamics in health and diseases. *Physical Review E*, 72(4):041904, 2005. doi:10.1103/PhysRevE.72.041904.

Andreas Kremling. *Kompendium Systembiologie: Mathematische Modellierung und Modellanalyse*. Vieweg+Teubner, Wiesbaden, Germany, 2012. ISBN 978-3-8348-1907-9.

Henry Markram. The blue brain project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006. doi:10.1038/nrn1848.

Marek Mateják, Tomáš Kulhánek, Jan Šilar, Pavol Privitzer, Filip Ježek, and Jiří Kofránek. Physiolibrary - Modelica library for physiology. In *Proceedings of the 10th International Modelica Conference*, pages 499–505, Lund, Sweden, 2014. doi:10.3384/ecp14096499.

Mohsen Naghavi, Haidong Wang, Rafael Lozano, Adrian Davis, Xiaofeng Liang, Maigeng Zhou, and Stein Emil Vollset. Global, regional, and national age-sex specific all-cause and cause-specific mortality for 240 causes of death, 1990–2013: a systematic analysis for the Global Burden of Disease Study 2013. *The Lancet*, 385(9963):117–171, 2014. doi:10.1016/S0140-6736(14)61682-2.

Emma Larsdotter Nilsson and Peter Fritzson. A metabolic specialization of a general purpose modelica library for biological and biochemical systems. In *Proceedings of the 4th International Modelica Conference*, pages 85–93, Hamburg, Germany, 2005.

Denis Noble. The rise of computational biology. *Nature Reviews Molecular Cell Biology*, 3(6):459–463, 2002. doi:10.1038/nrm810.

Chris P. Ponting, Peter L. Oliver, and Wolf Reik. Evolution and functions of long noncoding RNAs. *Cell*, 136(4):629–641, 2009. doi:10.1016/j.cell.2009.02.006.

R. Ranzinger and William S. York. Glyco-bioinformatics today (august 2011) – solutions and problems. In *Proceedings of the 2nd Beilstein Symposium on Glyco-Bioinformatics*, Potsdam, Germany, 2012.

Henrik Seidel. *Nonlinear dynamics of physiological rhythms*. PhD thesis, Technische Universität Berlin, Berlin, Germany, 1997.

Henrik Seidel and Hanspeter Herzel. Modelling heart rate variability due to respiration and baroreflex. In Erik Mosekilde and Ole G. Mouritsen, editors, *Modelling the Dynamics of Biological Systems*, number 65 in Springer Series in Synergetics, pages 205–229. Springer, Berlin Heidelberg, Germany, 1995. ISBN 978-3-642-79292-2.

Henrik Seidel and Hanspeter Herzel. Bifurcations in a nonlinear model of the baroreceptor-cardiac reflex. *Physica D: Nonlinear Phenomena*, 115(1-2):145–160, 1998. doi:10.1016/S0167-2789(97)00229-7.

Eberhard O. Voit. *A first course in systems biology*. Garland Science, New York City, New York, 2013. ISBN 978-0-8153-4467-4.

8.5. Mo|E — A communication service between Modelica compilers and text editors

Title	Mo|E — A communication service between Modelica compilers and text editors
Authors	N. Justus, C. Schölzel, A. Dominik, and T. Letschert
Publication type	Conference proceedings (talk)
Conference	12th International Modelica Conference
Location	Prague, Czech Republic
Year	2017
Pages	815–822
DOI	10.3384/ecp17132815

Contributions by dissertation author (using the Contributor Roles Taxonomy):

Conceptualization	Conceived project, formulated research goals
Supervision	Supervised the bachelor's thesis of first author
Validation	Tested the Mo|E server and Atom client on large projects
Writing — review & editing	Revised and reviewed first author's draft

Mo|E – A Communication Service Between Modelica Compilers and Text Editors

Nicola Justus[1] Christopher Schölzel[1] Andreas Dominik[1] Thomas Letschert[1]

[1]KITE, Technische Hochschule Mittelhessen, Giessen, Germany, {nicola.justus, christopher.schoelzel, andreas.dominik, thomas.letschert}@mni.thm.de

Abstract

The Modelica language is becoming increasingly popular among scientists and engineers as platform for modelling physical or biological systems. Although Modelica is maintained as non-proprietary language by the Modelica Association, a considerable number of commercial implementations and development environments is complemented by a surprisingly small number of open source tools.

In this paper, we present the communication service Mo|E that connects any text editor as front-end with a Modelica compiler as back-end. Based on the simple HTTP communication protocol, editor plugins for a software developer's favourite text editor can be developed easily, hence turning any editor into a lightweight Modelica development tool.

We also present a first implementation of a plugin for the text editor Atom that exhibits features necessary for efficient software development, such as display of compile errors, code completion, go to declaration or view of context-sensitive documentation. In addition, Modelica-specific checking of the number of equations in a model is supported.

Keywords: Modelica, open source, integrated development environment, distributed systems, structured editor, ENSIME, OpenModelica, JModelica, MoTE

1 Introduction

Modelica is a powerful object-oriented programming language that facilitates acausal description of physical systems. Although many commercial and open source tools for developing or working with Modelica are available, the OpenModelica suite (Fritzson et al., 2005) is the only comprehensive set of tools for Modelica. OpenModelica provides a standalone Modelica compiler, an Eclipse plugin for developing Modelica inside of Eclipse (MDT), a graphical model editor for connecting components (OMEdit), and a Modelica debugger. The primary tools for developing Modelica are MDT and OMEdit. Both are full-fledged integrated development environments (IDEs).

IDEs are well suited for working with big projects but may have some disadvantages. They often are slow, difficult to use and and may be even scary for novice users. For Modelica additional challenges arises from the differences between Modelica compilers, such as JModelica or OpenModelica which silghtly differ in their understanding of Modelica. In order to develop code compatible with different compilers, the IDE should be able to compile models using different compilers.

Today, when writing source code or any other type of structured text, it is common to use a structured editor which is aware of the document's structure. Structured editors are an essential part of most IDEs. Experienced developers usually prefer them to other – graphical – means of input. A structure aware editor must be able to analyze the text given to it. Thus structure awareness means awareness of the syntax and to some extend also of the semantics of the texts it deals with. The structured editor is deeply integrated with the IDE, rather than being just a mere component.

In this paper we present Modelica | Editor (Mo|E), a development environment for Modelica, centered on editing and checking complex models, refraining form all issues of model execution. A structured editor is its main component and user interface.

A key concept of Mo|E is that the user may use a text editor of her own choice, attach it to a service process that provides syntactic and semantic analysis and transforms the plain text editor to a structured editor.

Thus users may edit texts using the editor they are used to and still benefit from automatic recompilation, code completion, semantic highlighting, go to declaration, refactoring, and so on.

A central part of our solution is a server process that mediates between the text editor and Modelica aware analytical services. These services are provided by existing Modelica compilers, and/or further existing or future tools that may be plugged into this infrastructure (Figure 1). We have enhanced one text editor to a Modelica editor, but other text editors may be integrated with little effort. These editors only have to provide a plugin that implements the service API. This API provides a unique interface to different Modelica compilers and eases the communication with compilers and related tools, protecting users from complex and differing command-line interfaces.

The design of Mo|E was inspired by the ENSIME project (ENSIME Contrib., 2016) with its server process that mediates between text editors and Scala compilers.

Mo|E is an environment for developing Modelica models

using editors that are enhanced to be Modelica aware. It was realized as part of the first author's bachelor's thesis (Justus, 2016).

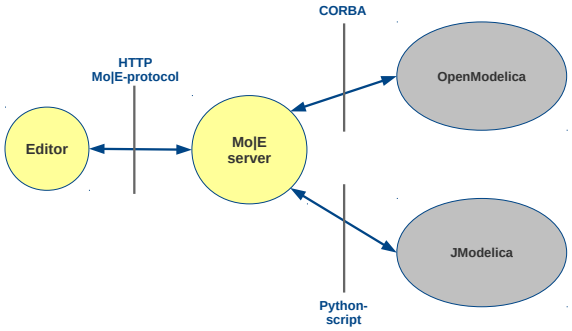


Figure 1. Survey of the communication between a text editor with a Mo|E-plugin, the Mo|E server and OpenModelica or JModelica.

1.1 Structure of the Paper

Section 2 describes which technologies and standards where used to implement Mo|E. Section 3 describes the protocol between client and service process, the communication with OpenModelica as well as with JModelica. Section 4 presents the key features of Modelica | Editor (Mo|E) and their use in the text editor Atom. Finally, section 5 gives a summary and a short outlook on future extensions.

1.2 Naming

The name Modelica | Editor (Mo|E) alludes to the use of the pipe character (|) in UNIX-like operating systems, which establishes a pipeline between two programs. Mo|E can be seen as such a pipeline between the Modelica compiler and a structured editor. In contexts where special characters like the pipe may cause problems, we chose the alternative spelling Modelica–Pipe–Editor (MoPE).

1.3 Goals

Our goals for Mo|E are:

- Provide an extendable client server application which makes it possible to develop Modelica inside existing text editors.
- Provide a client implementation for the text editor Atom as reference for other clients.
- Highlight syntax and type errors, perhaps while typing, inside the text editor.
- Provide code completion for models, data types, and variables.
- Provide jump to the source of a model. This is better known as "go to declaration".
- Provide a view of the documentation of a model.

1.4 Background and Related Work

OneModelica (Samlaus, 2015) is a an Eclipse-based IDE for Modelica models tailored to the domain of fluid dynamics. It was realized using tools and techniques of Model Driven Software Development. It may be compared to our approach in that it restricts itself to syntax and static semantics of the language and refrains from simulation issues. It differs considerably in its technological base, which in the years since its development has lost a lot of its attraction and support, not without reason as we think.

Mo|E is the first tool in a more ambitious project called Modelica Tool Ensemble (MoTE). MoTE aims at the provision of a collection of small user-friendly standalone applications for developing and executing Modelica models, i.e. a lightweight development environment for Modelica.

Modelica does not differ in principle from other languages when it comes to development environments. However, due to its complex static and dynamic semantics, it poses special challenges, mainly for the support of incremental development (see e.g. (Höger, Lorenzen, and Pepper, 2010) or (Broman, Fritzson, and Furic, 2006)).

We are well aware of these problems. Thus, at least for the time being, MoTE and Mo|E do not include a Modelica compiler or tools incorporating compiler features much beyond parsing. Instead we rely on mature compilers like OpenModelica and JModelica.

2 Technologies

2.1 Scala and Akka

Scala (EPFL, 2016) is a hybrid programming language that combines object orientation with functional programming. Because the Scala compiler generates bytecode for the Java Virtual Machine (JVM), it integrates with many available Java libraries. In addition, resulting compiled programs are platform independent. The service process of Mo|E is implemented in Scala.

Akka is a library for concurrent and distributed systems, based on the actor model that facilitates concurrency by providing a high level of abstraction (Allen, 2013). We use Akka as a provider of communication services, such as an implementation of the HTTP-protocol and for structuring the system according the actor model.

2.2 OMC and CORBA

OpenModelica provides the *Advanced Interactive OpenModelica Compiler* (OMC), a server that provides an API to query loaded Modelica code (Asghar et al., 2011).

The Common Object Request Broker Architecture (CORBA) is used by the OpenModelica compiler server OMC as interface to other applications and other programming languages.

DOI 10.3384/ecp17132815

CORBA developed by the Object Management Group (OMG) defines a standard for inter-process communication modeled as interaction of distributed objects. Because the public API of remote objects is defined in an Interface Definition Language (IDL), processes may be implemented in different programming languages (OMG, 2012).

2.3 Atom and the Electron Engine

Atom (GitHub, 2016a) is a text editor created by GitHub in the style of Sublime Text (Sublime HQ Pty Ltd, 2016). Basic design concepts of Atom include customization and extensibility through plugins (called packages in the context of Atom). Extending Atom is possible with JavaScript, HTML and CSS by using the Electron Engine (GitHub, 2016c). This allows to *rapidly develop* extensions and to implement communication protocols using *AJAX requests*. Furthermore, Atom already includes a package for syntax highlighting for Modelica (Chenouard et al., 2016), a simple API for completion suggestions (GitHub, 2016b) and a plugin for clicking on text (Facebook, 2016), which is used to implement *go to declaration* functionality.

We have created an Atom plugin as first reference implementation of a Mo|E client.

3 Design

3.1 Mo|E – Editor Protocol

Clients are connected to the service process, by means of Hypertext Transfer Protocol (HTTP)-based communication and JavaScript Object Notation (JSON) data representation. HTTP provides status codes, Uniform Resource Identifiers (URIs) and content negotiation (Fielding and Reschke, 2014). JSON is a compact text format, based on the JavaScript Object Notation (Bray, 2014).

The communication flow follows several steps: Firstly, the client connects to the service process using a *connect request* that communicates the current project. In this context a project is a directory containing Modelica source files.

Secondly, after initialization the service process answers with the respective *project id*. The unique project id identifies the project in the client server communication.

Henceforth, the client uses this project id to *request further IDE functionality* for this project from the service process.

To finally *finish a session,* the client sends a disconnect request that triggers the service process to delete all project-related information and cached data.

The following sections describe each supported IDE functionality in more detail.

3.1.1 Connecting to the server

As introduced in the preceding section, each client needs to connect initially with the server. A `connect` request is initiated through a POST request containing the respective JSON object with the project description. The JSON object contains the full path into the project directory and the relative path to a directory that is used to store compiled files:

```
POST /mope/connect

{
  "path": <String>,
  "outputDirectory": <String>
}
```

This project information is stored in the `mope-project.json` file that is placed in the project directory.

If the request was successful, the server answers with a project id. If not, the server answers with `400 BadRequest` and a detailed error message.

3.1.2 Compiling Modelica source files & Modelica script files

Compiling a Modelica source file is initiated through a `compile` request. The request body contains the path to the currently opened file. As a result of the request a model is instantiated and type errors are retrieved:

```
POST /mope/project/0/compile

{ "path": <String> }
```

If the request was successful, the server answers with a JSON array containing compiler errors:

```
{
  "type": "Error" | "Warning", //type of
      message
  "file": <String>, //path to the file
      which contains the error
  "start": { //start of error
    "line": <Number>,
    "column": <Number>
  },
  "end": { //end of error
    "line": <Number>,
    "column": <Number>
  },
  "message": <String> //compiler error
}
```

Compiling a Modelica script file is initiated by sending an analogous `compileScript` request:

```
POST /mope/project/0/compileScript

{ "path": <String> }
```

Although the request is called “compiling a Script file”, the service process actually executes the script. This action is intended for debugging purposes of smaller scripts and not for scripts that simulate a model, since simulating a model is time-consuming and may freeze or possibly even kill the service process.

3.1.3 Checking a model

To check a model for its number of equations the client sends a `checkModel` request with the model path. The server calls the OpenModelica compiler to run `checkModel` and answers with a string containing the results:

```
POST /mope/project/0/checkModel

{ "path": <String> }
```

This functionality is only available, if the OpenModelica compiler is used.

3.1.4 Go to declaration

To retrieve the declaration of a model, the client sends a `declaration` request. This request contains the model/-class name as query string[1]:

```
GET /mope/project/0/declaration?class=[
    Modelname]
```

The server answers with a JSON object containing the file path and line number of the declaration:

```
{
  "path": <String>, //absolute path to the
      file
  "line": <Number> //line number
}
```

If the project id is unknown or the query string is missing, the server will answer with a `404 NotFound` error.

3.1.5 Go to documentation

A model documentation can be retrieved using a `doc` request with the model name encoded as query string:

```
GET /mope/project/0/doc?class=[Modelname]
```

The server embeds the documentation in a template and returns a HTML document that can be viewed in a web browser.

If the project id is unknown or the query string is missing, the server answers with a `404 NotFound` error.

3.1.6 Code completion

For code completion the client sends a `completion` request with a JSON object that describes the position of the cursor as *file* (name of current file), *line* and *column number* (position of the cursor) and *word* (part of the expression to be completed):

```
POST /mope/project/0/completion

{
  "file": <String>, //absolute path to the
      file
  "position": { //position inside the file
    "line": <Number>,
    "column": <Number>,
  },
  "word": <String>
}
```

The server responds by sending an JSON array of possible completions for the expression:

```
{
  //type of completion; 1 of the listed
      strings
  "kind": "Type" | "Variable" | "Function"
      | "Keyword" | "Package" | "Model" | "
      Class" | "Property",
  "name": <String>, //the completion
  //OPTIONAL: list containing names of
      parameters if kind=function
  "parameters": [
    <String>,
    <String>,
    ...
  ],
  //OPTIONAL: the class comment describing
      the name attribute
  "classComment": <String>,
  //OPTIONAL: the type of name
  "type": <String>
}
```

kind defines the type of the completion (such as package, class, function, variable, etc.). *name* is the suggestion for the subexpression.

The optional return values for *parameters*, *classComment* and *type* report the list of argument names if the suggestion is a function, the documentation string if the the suggestion is a class and the data type of the expression (usually the data type of a variable), respectively.

If the given project id is unknown, the server answers with `404 NotFound`.

3.1.7 Display data type of a variable

To retrieve data type and documentation string of a variable, the client sends a `typeOf` request with a body identical to the body of the `completion` request. If the request was successful, the server answers with a JSON object containing the name, type, and documentation string of the variable. Otherwise the server answers with `404 NotFound`:

```
POST /mope/project/0/typeOf

{
  "name": <String>, //name of property
  "type": <String>, //type of property
  //OPTIONAL: property comment
  "comment": <String>
}
```

3.1.8 Disconnecting from the server

A session is terminated by a disconnect request, which initiates the shutdown sequence for this project on the server:

```
POST /mope/project/0/disconnect
```

The server returns `204 NoContent` if the project id is known or `404 NotFound` elsewise.

3.1.9 Stopping the server

The client can stop the whole service process by sending a `stopServer` request. The server answer is `202 Accepted`.

```
POST /mope/stop-server
```

[1] A query string is a component of a URI, that starts with a **?** (Berners-Lee, Fielding, and Masinter, 2005).

DOI 10.3384/ecp17132815

3.2 Communication with OpenModelica

The modeling and development environment OpenModelica (OSMC, 2016) consists of a Modelica compiler (omc), a graphical connection editor (OMEdit), an Eclipse plugin (MDT) and a Modelica debugger (Fritzson et al., 2005). As described in Chapter 2.2 the compiler enables querying for model information via its CORBA interface that provides several types of information:

- list of all models/classes by sending `getClassNames`,
- source file of a model by sending `getSourceFile`,
- documentation annotation of a model by sending `getDocumentationAnnotation`,
- result of a model check for equations by sending `checkModel`,
- documentation string of a model by sending `getClassComment`,
- arguments of a function by sending `getParameterNames`,
- specialization of a class by sending `getClassRestriction`.

An additional difficulty arises from the fact that OpenModelica uses Modelica expressions as arguments for its CORBA interface. As a result, the functions listed above are not implemented explicitely in the CORBA interface. Instead, OpenModelica only provides a single method in its CORBA interface, namely `sendExpression` and sends Modelica source code strings and API function calls as arguments. Therefore, we create the function calls as strings and interpolate them into the function argument, as shown in Listing 1.

Listing 1. API function call through OpenModelica's CORBA interface.

```
val omc:OmcCommunication = ...
val fileName = "/tmp/model.mo"
val errors:String =omc.sendExpression(s"""
    parseFile("$fileName")""")
```

3.3 Communication with JModelica

JModelica (Modelon AB, 2016) is a Modelica compiler developed by Modelon AB (Åkesson et al., 2010). To allow dynamic adjustments during execution, JModelica offers a Python interface which enables code modification at run time. In addition it enables compilation of Modelica code. We are using this Python interface for compilation of the models by delivering the Modelica source files to a custom Python script, which calls the JModelica compiler, parses JModelica's output and encodes the output into JSON. The resulting JSON is printed to `stdout` which is afterwards parsed by the service process and finally decoded as Scala objects. The communication scheme is depicted in Figure 2.

Command	Description
Mope: Disconnect	Disconnect Atom from the service process
Mope: Compile Project	Compile the project
Mope: Run Script	Execute the Modelica script
Mope: Check Model	Check the model for its number of equations
Mope: Show Type	Display the data type of the variable below the cursor
Mope: Open Documentation	Open the documentation of the type below the cursor
Mope: Open Server Log	Open the log file of the service process
Mope: Open Server Config	Open the configuration file of the service process
Mope: Stop Server	Stop the server

Table 1. List of commands implemented in the Atom plugin.

Unfortunately JModelica does not offer access to the parsed model or its abstract syntax tree. That is the reason why code completion is restricted to local variables and go to documentation is not yet supported in the presented MoIE Atom plugin.

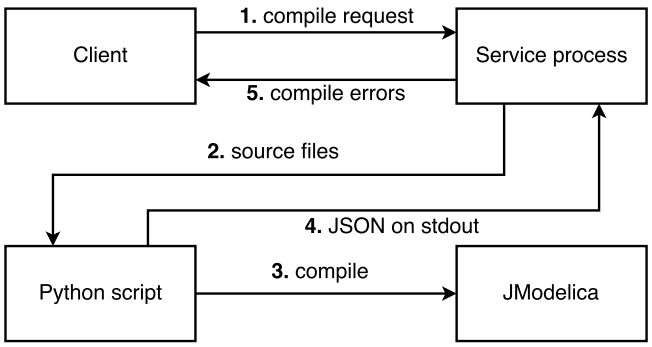


Figure 2. Diagram of the communication between a text editor (client) and JModelica.

4 Features

4.1 Client commands

Table 1 gives a full list of the commands available in the Atom plugin.

4.2 Compiler Feedback

Modelica | Editor (MoIE) provides *instant compiler feedback* for syntax errors and *type errors*. Background compi-

lation is automatically triggered when a file is saved and the errors are highlighted in the editor with a red indicator at the left side of the editor tab. Error messages are displayed at the bottom of the tab (Figure 3). Alternatively automatic compilation can be disabled and triggered manually.

As Mo|E supports JModelica and OpenModelica it is possible to use either JModelica or OpenModelica or both compilers for one project.

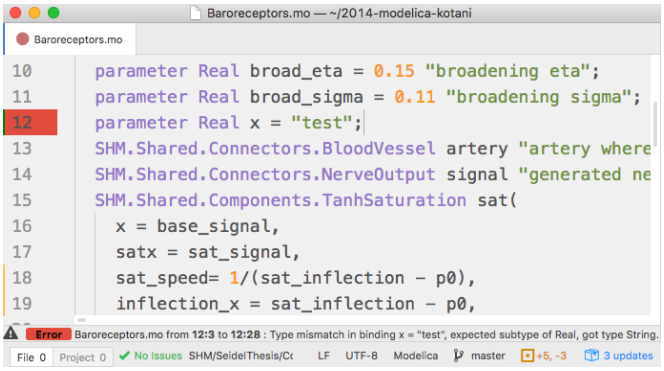


Figure 3. Compile errors are retrieved form the back-end (OpenModelica or JModelica) by the Mo|E server and highlighted in the source code by the editor plugin.

4.3 Code Completion

Modelica | Editor (Mo|E) features *enhanced code completion* on keystrokes or by pressing `Ctrl + Space`. Suggestions include classes, models, functions, model parameters and variables, keywords, built-in types as well as local variables. The suggestions contain a type indicator, documentation string and a link to the model's documentation (Figure 4). The type indicator displays the type of the suggestion (package, model, function or variable).

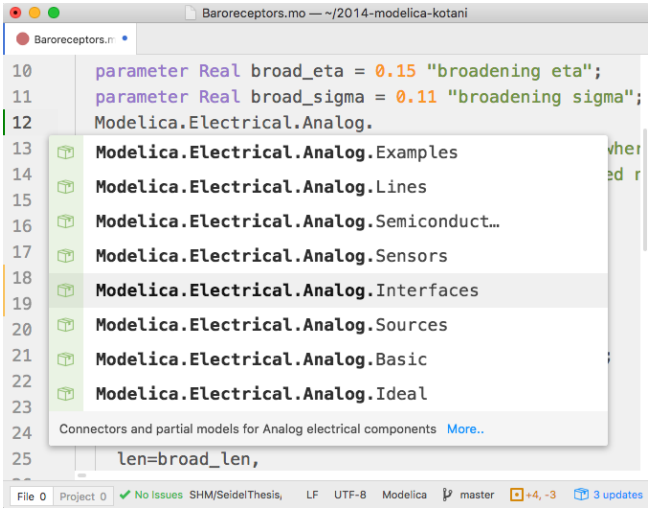


Figure 4. Code completion allows for selecting classes, models, functions, model parameters, variables, keywords or built-in types from a list of suggestions retrieved by the Mo|E server.

4.4 Go to Declaration

Mo|E provides *go to declaration* by clicking on the model/class name while holding down `Ctrl`. The source file of the model/class is opened in a separate tab. Go to declaration is mostly used for discovering source code or when editing multiple models that are linked to each other.

4.5 Documentation View

Mo|E embeds the queried documentation of a model in a predefined template and provides the documentation as HTML document. The implementation in the Atom plugin opens the requested documentation in the default browser. Furthermore it is possible to browse the model's child components using the links in the subcomponents section of the documentation (Figure 5).

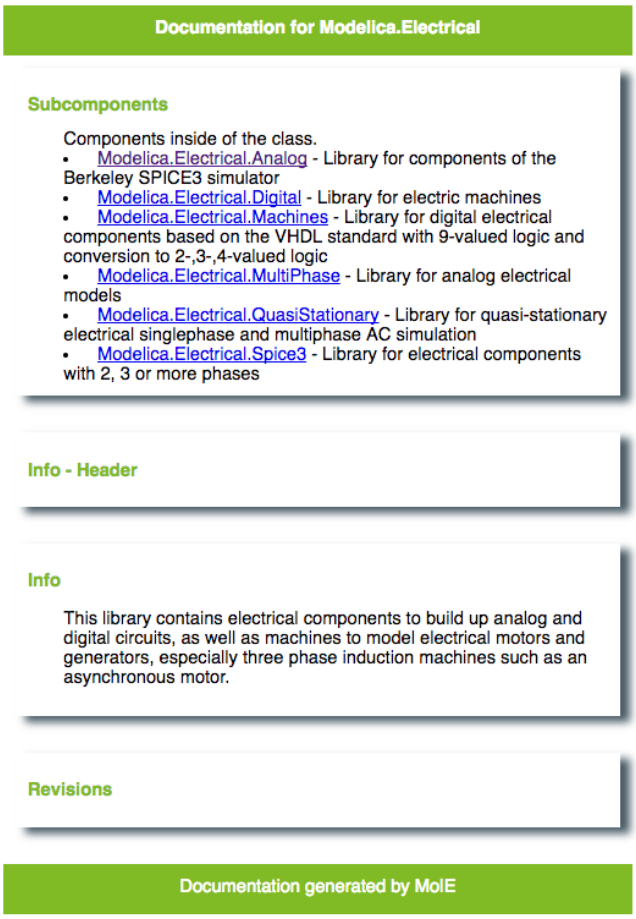


Figure 5. Example of a documentation display generated as HTML page by Mo|E by embedding the retrieved documentation string with a template page.

4.6 Type & Documentation String Display

Mo|E provides a command for displaying the type and documentation string of the variable at the cursor position. Type and documentation are displayed at the bottom of the editor tab (Figure 6).

DOI 10.3384/ecp17132815

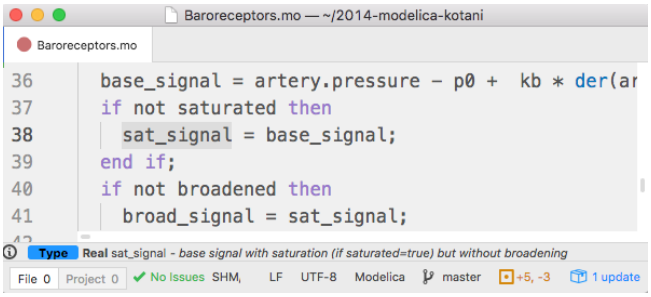


Figure 6. Data type and documentation string are displayed in the editor window by the Atom plugin.

4.7 Execution of Modelica Scripts

If the OpenModelica compiler is used, MoIE allows manually triggered execution of Modelica scripts and displays error messages in the editor.

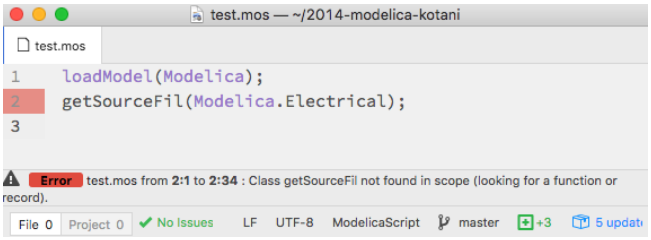


Figure 7. Compile errors of Modelica scripts are displayed in the editor window by the Atom plugin.

4.8 Model Check

MoIE supports checking of a model for the number of equations (Figure 8), if the OpenModelica compiler is used.

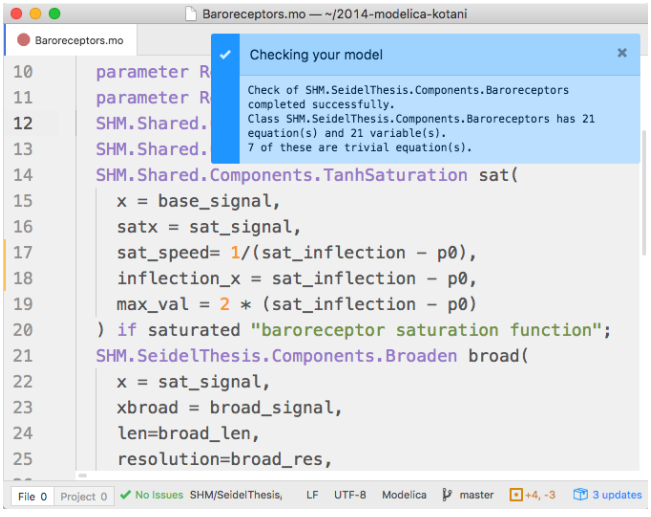


Figure 8. Result of a model check, performed by the OpenModelica back-end, is displayed as pop-up in the editor window by the Atom plugin.

5 Conclusions

This paper presented a extendable client/server application for developing Modelica in enhanced text editors like Atom. It shows how a service process is used to simplify communication with multiple Modelica Compilers and provide IDE features to various text editors through a simple interface. Text editors have to implement a small number of basic HTTP calls, which should be a minimal effort. A minimal setup with compilation and code completion would only require four HTTP calls. Installation instructions for MoIE can be found at `https://github.com/THM-MoTE/mope-server`.

MoIE is a base for further extensions. E.g. we intend to implement plugins for different editors, such as Sublime Text (Sublime HQ Pty Ltd, 2016), Visual Studio Code (Microsoft Corporation, 2016) or vim (Moolenaar, 2016). Including Visual Studio Code should not be a problem because it uses TypeScript for its plugins, which is a superset of Atom's JavaScript.

MoIE is part of a larger ensemble of tools called MoTE (Schölzel et al., 2016). MoTE will also include a vector graphic editor called Modelica Vector Graphics Editor (MoVE) (Justus et al., 2017) and a diagram editor called Modelica Diagram Editor (MoDE) (Hoppe et al., n.d.). Together with MoIE these tools provide alternative user interfaces for the interaction with existing Modelica compilers, which allow a simpler interaction than full-fledged IDEs like OpenModelica.

The projects are open source and hosted on GitHub:
`https://github.com/thm-mote/`

References

Åkesson, J. et al. (2010). "Modeling and Optimization with Optimica and JModelica.org — Languages and Tools for Solving Large-Scale Dynamic Optimization Problems". In: *Computers & Chemical Engineering* 34 (11), pp. 1737–1749.

Allen, Jamie (2013). *Effective Akka*. Sebastopol, USA: O'Reilly Media.

Asghar, Syed Adeel et al. (2011). "An Open Source Modelica Graphic Editor Integrated with Electronic Notebooks and Interactive Simulation". In: *Proceedings of the 8th International Modelica Conference*. Dresden, Germany, pp. 739–747.

Berners-Lee, T., R. Fielding, and L. Masinter (2005). *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. IETF.

Bray, T. (2014). *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. IETF.

Broman, D., Peter Fritzson, and S. Furic (2006). "Types in the Modelica Language". In: *In Proceedings of the 5th International Modelica Conference*. Ed. by Ch.and Haumer A. Kral. Vienna, Austria: The Modelica Association, pp. 303–317.

Chenouard, Raphael et al. (2016). *Modelica language support in Atom*. GitHub Repository. URL: https://github.com/modelica-tools/atom-language-modelica (visited on 11/03/2016).

École Polytechnique Fédérale de Lausanne (2016). *The Scala Programming Language*. URL: http://www.scala-lang.org/ (visited on 11/01/2016).

ENSIME Contributors (2016). *ENSIME*. URL: http://ensime.github.io/ (visited on 11/01/2016).

Facebook (2016). *Hyperclick*. GitHub Repository. URL: https://github.com/facebooknuclide/hyperclick (visited on 11/03/2016).

Fielding, R. and J. Reschke (2014). *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. IETF.

Fritzson, Peter et al. (2005). "The OpenModelica Modeling, Simulation, and Development Environment". In: *Proceedings of the 46th Scandinavian Conference on Simulation and Modeling (SIMS)*. Trondheim, Norway.

GitHub (2016a). *Atom*. URL: https://atom.io (visited on 11/01/2016).

– (2016b). *Autocomplete+ Package*. GitHub Repository. URL: https://github.com/atom/autocomplete-plus (visited on 11/03/2016).

– (2016c). *Electron — Build cross platform desktop apps with JavaScript, HTML, and CSS*. URL: http://electron.atom.io/ (visited on 09/14/2016).

Höger, Christoph, Florian Lorenzen, and Peter Pepper (2010). "Notes on the Separate Compilation of Modelica". In: *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Ed. by P. Fritzon et al. Oslo, Norway: Linköping Electronic Conference Proceedings, pp. 43–53.

Hoppe, Marcel, Christopher Schölzel, and Andreas Dominik. *MoDE – A Standalone Modelica Diagram Editor*. unpublished.

Justus, Nicola (2016). "Design and Implementation of a Client/Server Application for Editing Modelica Inside Various Text Editors". BA thesis. Giessen, Germany: Technische Hochschule Mittelhessen.

Justus, Nicola, Christopher Schölzel, and Andreas Dominik (2017). "MoVE – A Standalone Modelica Vector Graphics Editor". In: *12th International Modelica Conference*. Prague, Czech Republic. to be published.

Microsoft Corporation (2016). *Visual Studio Code - Code Editing. Redefined*. URL: https://code.visualstudio.com/ (visited on 22/12/2016).

Modelon AB (2016). *JModelica.org*. URL: www.jmodelica.org (visited on 11/01/2016).

Moolenaar, Bram (2016). *welcome home : vim online*. URL: http://www.vim.org/ (visited on 12/21/2016).

Object Management Group (2012). *Common Object Request Broker Architecture (CORBA). Part 1: CORBA Interfaces*. OMG document formal/2012-11-12. Version 3.3.

Open Source Modelica Consortium (2016). *OpenModelica*. URL: https://openmodelica.org (visited on 11/01/2016).

Samlaus, Roland (2015). "An Integrated Development Environment with Enhanced Domain-Specific Interactive Model Validation". PhD thesis. Linköping University, The Institute of Technology.

Schölzel, Christopher et al. (2016). *Modelica Tool Ensemble*. GitHub Repository. URL: https://github.com/orgs/THM-MoTE/ (visited on 12/22/2016).

Sublime HQ Pty Ltd (2016). *Sublime Text: The text editor you'll fall in love with*. URL: https://www.sublimetext.com/ (visited on 11/01/2016).

 DOI 10.3384/ecp17132815

8.6. MoVE — A standalone Modelica vector graphics editor

Title	MoVE — A standalone Modelica vector graphics editor
Authors	N. Justus, C. Schölzel, and A. Dominik
Publication type	Conference proceedings (talk)
Conference	12th International Modelica Conference
Location	Prague, Czech Republic
Year	2017
Pages	809–814
DOI	10.3384/ecp17132809

Contributions by dissertation author (using the Contributor Roles Taxonomy):

Conceptualization	Conceived project, formulated research goals
Investigation	Provided mathematical details of geometric calculations
Supervision	Supervised the bachelor's thesis of first author
Writing — review & editing	Revised and reviewed first author's draft

MoVE – A Standalone Modelica Vector Graphics Editor

Nicola Justus[1] Christopher Schölzel[1] Andreas Dominik[1]

[1]KITE, Technische Hochschule Mittelhessen, Giessen, Germany, {nicola.justus, christopher.schoelzel, andreas.dominik}@mni.thm.de

Abstract

Modelica models can have a graphical icon defined as a bitmap or vector graphics. Vector graphics have several benefits, the most obvious being free scaling of images from icon to poster size. With OpenModelica there already exists one open source tool that can be used for editing these vector graphics icon annotations, but it does not reach the usability comfort of professional vector graphics editing tools.

In this paper we present the Modelica Vector graphics Editor (MoVE), a standalone open source editor for Modelica's vector graphics syntax that provides many convenience features inspired by the vector graphics editor Inkscape. These features include grouping, snap to grid, move to foreground/background, rotation handles, and drawing perfect circles and squares as well as horizontal and vertical lines when holding *Shift*.

We hope that MoVE, as a part of the Modelica Tool Ensemble (MoTE), can enrich the open source ecosystem of Modelica by simplifying the creation of more elaborate vector graphics icons for Modelica models.

Keywords: JavaFX, vector graphics, open source, SVG, Inkscape, MVC, MoTE, OpenModelica

1 Introduction

Modelica is a language for modeling complex physical systems that also incorporates a graphical representation of model components into the language itself. These graphical representations come in the form of annotation statements that can either contain a link to a bitmap image or define an image using a vector graphics syntax (Modelica Assoc., 2012). Vector graphics have the advantage that they are freely scalable. This is interesting in any context where a model might not only be displayed as a small icon on a screen but also has to be presented to a larger audience on a slide or a poster.

Unfortunately, creating vector graphic icons for Modelica models is not as easy as it could be. Standard vector graphics tools such as Inkscape (Inkscape, 2016) provide a rich set of features for precise and fast interaction, such as grouping, rotation handles, sending elements to the front or back, snap to grid, or drawing straight horizontal lines and perfect circles when holding a modifier key. It would be ideal, if we could use such a tool and save the resulting image in Modelica notation. However, the Modelica annoations are not compatible with vector graphics formats such as Scalable Vector Graphics (SVG) (Dahlström et al., 2011), since there are both features in SVG that have no equivalent in the Modelica syntax and vice versa.

There are many commercial tools for Modelica but OpenModelica is the only open source choice for graphical editing of Modelica models (Fritzson et al., 2005). This graphical editor of OpenModelica is called OpenModelica Connection Editor (OMEdit) (Asghar et al., 2011). It has all features that are required to create vector graphic annotations, but does not provide the same level of user-friendliness as Inkscape or related tools. For example, non-standard rotation angles, fill patterns and line patterns can only be changed via a properties dialog that has to be opened separately for each component; the order of Elements cannot be changed (although respective entries exist in the context menu); drawing of straight horizontal lines and perfect circles is not supported; and when we began this project, OpenModelica did not even support undo-redo operations for graphical manipulations. Furthermore OMEdit generates the icon annotation as one big line of code. This is especially uncomfortable when the source code is managed through a version control system.

In this Paper we therefore present the Modelica Vector graphics Editor (MoVE), a new standalone open source Modelica vector graphic editor with a streamlined interface similar to Inkscape. In the following, we will first give a bit more detail of the context in which MoVE was created. In section two, we will then present an overview of the technologies used to create the editor followed by a discussion of the major design aspects in section three. Section three presents the major features of MoVE and section four explains current limitations leading to the conclusion in section six.

1.1 Background and Related Work

Modelica Tool Ensemble

MoVE is part of the Modelica Tool Ensemble (MoTE) (Justus, Hoppe, and Schölzel, 2017). MoTE aims to provide small user-friendly standalone appliations for editing and simulating Modelica models. With this toolset we follow the Unix philosophy to "make each program do one thing well" (McIlroy, Pinson, and Tague, 1978). Separating the different tasks needed for editing and simulating Modelica models leads to smaller applications that are easier to maintain than a full-featured development environment like OpenModelica. Additionally, users may choose to use the tools that they like and substitute the tools they do not like

with other alternatives, leading to a more flexible ecosystem that can accommodate different user needs. MoVE only touches the main annotation statement of a model. To edit other parts of the model, one can, for example, use the text editor Atom (GitHub, 2016) which can provide type checking, auto-completion and error highlighting when coupled with Modelica | Editor (MolE), another tool in the MoTE family. Instead one could also chose to use OMEdit or the Eclipse plugin Modelica Development Tooling (MDT) (Pop et al., 2006).

Inkscape

The main source of inspiration for MoVE was the aforementioned vector graphics editor Inkscape (Inkscape, 2016). Inkscape is an open source application that can be used to create professional and complicated vector graphic images. It supports a rich set of features including alignment of elements or individual nodes, combination and cutting of multiple paths, drawing with Bezier curves, bold and italic text, importing shapes from a PDF-file, and many many more. Features that are not already included can be made available with a language-independent scripting API. These features are presented to the user mainly through toolbars and hotkeys that make the interaction fast and seamless. The native format of Inkscape is SVG (Dahlström et al., 2011), which is an XML-based format that can easily be interpreted by other tools.

MoVE does not nearly provide as many features as Inkscape, but it tries to follow the same principles for usability and precision.

2 Technologies

This chapter is a short overview over the technologies that are used for implementing MoVE. Basically MoVE is written in the programming language Scala using the graphical user interface toolkit JavaFX.

2.1 Scala

Scala is a programming language for the Java Virtual Machine (JVM). This means that it is platform-independent and that it is possible to use any Java library. Especially the library JavaFX is useful for creating a modern Graphical User Interfaces (GUIs). Scala merges object orientation with functional programming, which allows to write code faster and more flexible than in plain Java. It also brings its own set of libraries such as a parser combinator library (EPFL and Typesafe, Inc., 2016) that proved very useful for this project.

2.2 JavaFX

JavaFX is a GUI toolkit for the Java programming language. Because it is written for Java it runs on the JVM and is also usable from Scala. JavaFX is the latest toolkit for GUIs running on the JVM and incorporates many ideas of modern GUI design. JavaFX provides a special format for describing the structure of a UI. This format is called FXML and based on XML. To develop GUIs using the FXML format, Oracle provides a graphical editor for building the user interface by dragging and dropping interface components, namely the *SceneBuilder*. This makes GUI design much faster and leads to a clean separation of the layout and the actual code.

3 Design

3.1 Parser

To load existing Modelica models we have created a simple parser for Modelica source code. This parser is built using the *scala-parser-combinators* library (EPFL and Typesafe, Inc., 2016). This library allows combination of simple parsers to create more complex ones. An external parser generator is not necessary. MoVE ignores everything in the source file, except the icon annotations of all models defined in the file. This makes the parser (and MoVE) mostly independent of future language modifications, thus MoVE should work with future versions of Modelica. If the icon annotations are modified, the parser has to be modified. This should be a small effort. Additionally this assures that MoVE interacts nicely with version control systems. Since we only parse annotations, we can *guarantee* that we will not change any other part of the model.

During the parsing process, the parser generates a MoVE-specific abstract syntax tree. This tree is then transformed into shapes, that are derived from the standard JavaFX shapes. Finally this shapes are displayed in the user interface.

3.2 Model–View–Controller

JavaFX is built around the Model-View-Controller (MVC) software design pattern (Reenskaug, 1979). MVC splits the application in three parts. The first part is the model[1], which represents the business data. The model updates the view if someone changes the model. The second part is the view, which displays the data and listens on updates to the model. The third part is the controller, which connects a model with the respective view. The controller is also responsible for user interactions and transforms them into commands for the model or the view. The typical MVC workflow is depicted in Figure 1.

Because JavaFX already provides views, which contain the data representation for shapes, MoVE is designed with controllers and views. There are no explicit models. They are *hidden inside* of the JavaFX shapes.

3.3 JavaFX Elements

To display Modelica's graphical primitives (Modelica Assoc., 2012), we have created a small set of JavaFX elements. These elements are all derived from the standard JavaFX shape elements and add additional properties and behavior such as fill and stroke patterns. The JavaFX shapes

[1]Here, in section 3.2, the word "model" refers to source code of a software project that is structured with the MVC-Pattern and not to a Modelica model.

DOI 10.3384/ecp17132809

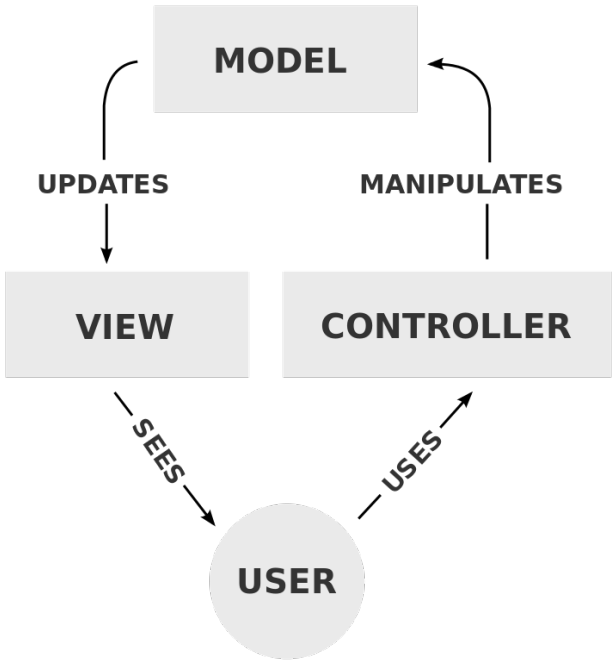


Figure 1. The Model-View-Controller (MVC) software design pattern splits an application into three parts in order to increase maintainability and extensibility (Frey, 2016).

provides the basic properties for rectangles, ellipses, lines, paths, polygons, and images.

Furthermore, for abstracting the common behavior of the shapes, they are all derived from a specific trait, which provides the common behavior. For example, all shapes that *behave like* a *rectangle* are derived from the trait *RectangleLike*. A similar trait is defined for shapes that *behave like* a *path*.

3.4 UI Overview

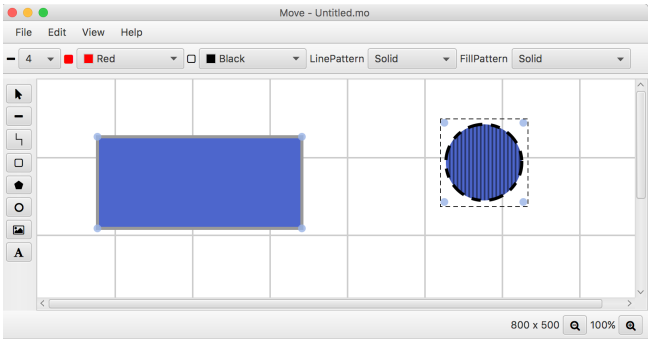


Figure 2. The user interface of MoVE is built with the JavaFX-framework and consists of three main toolbars: tool selection (left), tool properties (top) and zoom and size indicator (bottom).

The user interface contains 3 toolbars for interacting with MoVE (Figure 2).

The top toolbar contains controls for specifying the color of selected or newly drawn shapes. Going left to right this toolbar starts with a selector for the stroke size, followed by the color pickers for the fill color and stroke color. The color pickers are followed by a selector for the *LinePattern* and *FillPattern*. For these last two elements, all patterns defined in chapter 18.6 from (Modelica Assoc., 2012) can be selected.

The left toolbar contains the tools for selecting and moving as well as drawing the icon primitives. Going top to bottom it starts with the arrow, which is used for selecting and moving shapes. The arrow is followed by the tools for drawing lines, paths, rectangles, polygons and ellipses, and for inserting images, and text.

The bottom toolbar currently only contains two items: an indicator for the size of the draw pane and buttons for controlling the zoom. The magnifying glass with the minus zooms out and the magnifying glass with the plus zooms in.

4 Features

4.1 Code Generation

MoVE provides two code generators for the icon annotation. The first generates the annotation as one big line and writes it into the model. This is similar to OMEdit. The second code generator generates *pretty-printed* code with line breaks and indentations, which is more readable than a big line (Listing 1). This style is also better supported by version control systems as they can recognize which lines or properties have changed instead of reporting only a change of the whole annotation.

Listing 1. MoVE generates a well formatted icon annotation with line breaks and indentation.

```
model Test
  annotation(
  Icon (
    coordinateSystem(
      extent = {{0,0},{200,125}}
    ),
    graphics = {
      Rectangle(
        origin = {34,96},
        lineColor = {0,0,0},
        fillColor = {128,186,36},
        lineThickness = 4.0,
        pattern = LinePattern.Solid,
        fillPattern = FillPattern.Solid,
        extent = {{-14,8}, {14,-8}}
      ),
      Ellipse(
        origin = {75,91},
        lineColor = {0,0,0},
        fillColor = {128,186,36},
        lineThickness = 4.0,
        pattern = LinePattern.Solid,
        fillPattern = FillPattern.Solid,
        extent = {{-13,10}, {13,-10}},
        endAngle = 360
      )
    })
  );
end Test;
```

4.2 Grouping

MoVE supports grouping of multiple shapes through *Edit → Group* or by pressing *Ctrl+G*. Moving a shape which is part of a group moves all shapes that are part of this group (Figure 3). Ungrouping is also supported through *Edit → Ungroup* or by pressing *Ctrl+Shift+G*. Note that the groups are only used in MoVE and are discarded when the model is saved, because this is not supported by the icon annotation syntax of Modelica.

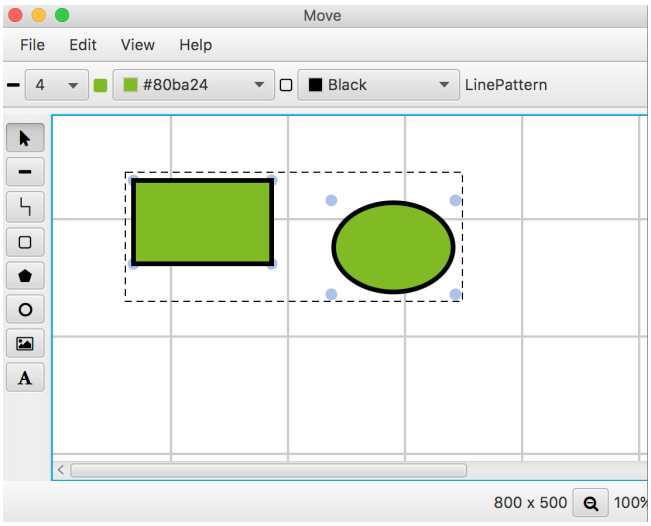


Figure 3. MoVE allows to group shapes together in the user interface, so that they can be easily moved together. These groups are lost when the annotation is saved.

4.3 Stacked Shapes

MoVE allows to move shapes into the background using *ContextMenu → In Background* and to move shapes into the foreground using *ContextMenu → In Foreground* (Figure 4). This allows easy modifying of the order of stacked shapes.

Furthermore, MoVE also handles shapes with the fill pattern *FillPattern.None* in an intuitive way. Shapes that are behind the transparent filling can still be selected. The transparent shape itself is only selected when the user clicks on the visible border.

4.4 Export as Images

MoVE enables exporting of Modelica icons either as *PNG* or as *SVG* (Figure 5). SVG is especially interesting because SVG images can be further modified in Inkscape. This is useful if the user likes to create a poster which contains a graphic from a Modelica model.

4.5 Rotation

After a double click on a shape, four red anchors appear at the corners of the shape (Figure 6). Moving the anchors rotates the shape around its center. This is more intuitive than rotating a shape by defining a specific degree value through a separate property dialog.

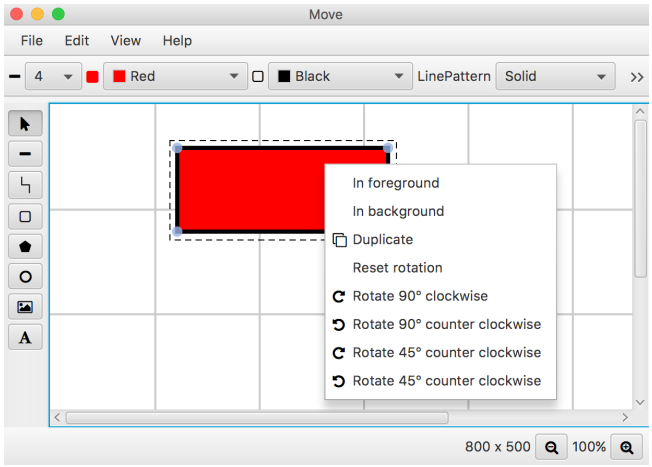


Figure 4. The context menu for a shape contains controls for rotation and stacking order.

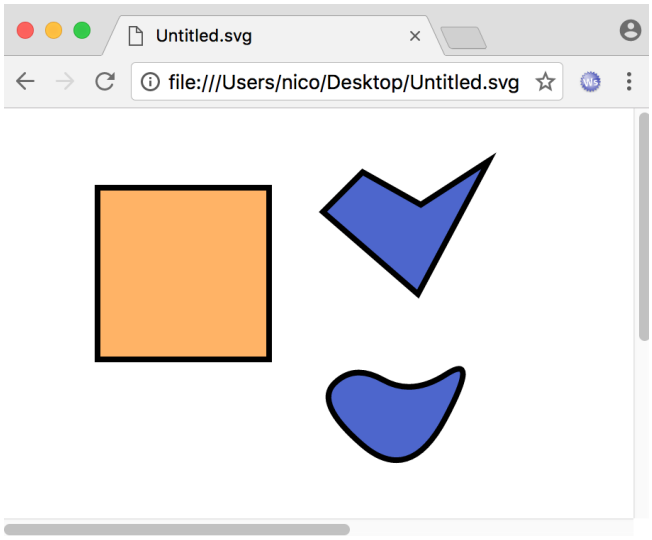


Figure 5. SVG image exported from MoVE displayed in Google Chrome.

Additionally to rotation by moving the anchors, it is possible to rotate an element using the context menu:

- *ContextMenu → Rotate 90° clockwise*
- *ContextMenu → Rotate 90° counter clockwise*
- *ContextMenu → Rotate 45° clockwise*
- *ContextMenu → Rotate 45° counter clockwise*

4.6 Snap to Grid

MoVE operates on a customizable grid. The size of the grid can be modified to fit the needs of the user. Via the menu entry *View → Enable snapping* or by pressing *Ctrl+A* the *snap to grid* function can be toggled. If activated, elements will snap to the precise location of the grid lines when they are moved close to such a line. This allows for a precise alignment of individual elements.

DOI 10.3384/ecp17132809

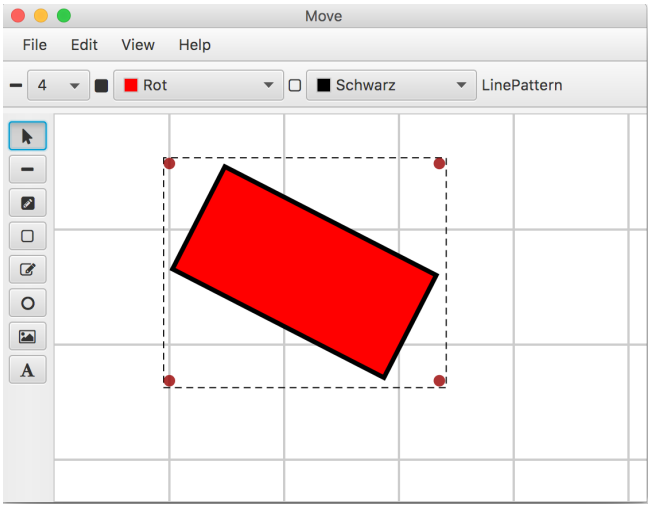


Figure 6. Arbitrary rotations can be realized in MoVE by rotation handles (red dots).

4.7 Config Files

MoVE uses simple text files as configuration files that are placed inside of `~/.move`. Application settings are placed in the file `~/.move/move.conf` and keyboard shortcuts are read from `~/.move/shortcuts.conf`. Both files can be customized with any text editor.

4.8 Additional features

When holding down *Shift* while drawing a shape, it is possible to create straight horizontal or vertical lines, perfect squares, and perfect circles (Figure 7).

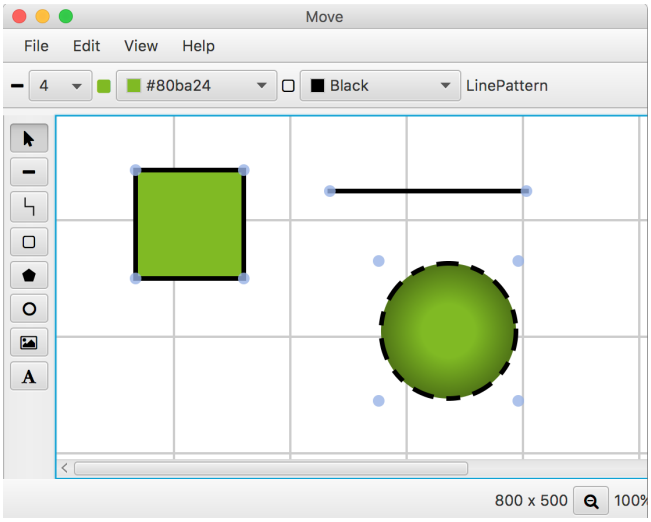


Figure 7. When holding *Shift*, MoVE will create perfect squares and circles and straight horizontal or vertical lines.

MoVE supports *undo and redo* using *Edit → Undo* / *Edit → Redo* or through the shortcuts *Ctrl+Z* and *Ctrl+Shift+Z*. It is also possible to *copy*, *paste* and *duplicate* selected shapes. Holding down *Shift* and selecting a shape selects multiple shapes.

5 Limitations

5.1 Annotations

MoVE supports every icon annotation except properties which are defined using a `if-clause` or a `DynamicSelect` statement, because the result of both statements is a dynamic value, which is only defined at runtime. These dynamic definitions do not fit into the scope of an editor for static vector graphic images. If MoVE finds properties, which are defined using this two statements, it warns the user that this properties will be overwritten by a static value after a save call (see Figure 8).

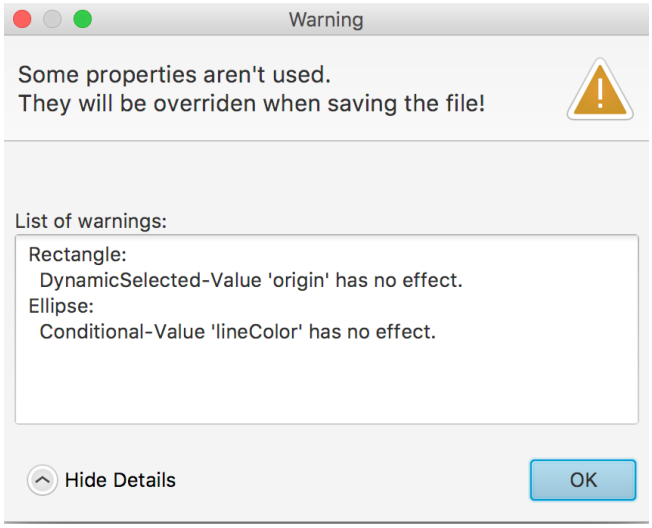


Figure 8. A Warning is displayed when opening a Model whose icon annotation contains `DynamicSelect` and `if-clause` elements in MoVE.

5.2 Line Scaling

The Modelica language specification does not define the meaning of the `thickness` property of a line (Modelica Assoc., 2012). The most intuitive definition would be to assume that the thickness of a line is given in units of the coordinate system of the icon. Both Dymola and OMEdit, however, define the line thickness in terms of the coordinate system of the users screen, so that lines scale automatically when zoomed. At the moment, MoVE does not follow this behavior, because it is unintuitive and cannot be reproduced when the image is exported to SVG or Portable Network Graphics (PNG).

5.3 Placing Connectors

MoVE currently does not support placing connectors in the icon, because this would require parsing and altering connector definitions in the model. Loading and saving models with MoVE does not affect existing connector placements. MoVE is only a graphical editor for the main annotation statement of Modelica models and leaves the rest of the code untouched. Connector placement would add another layer of complexity to the tool that goes beyond its intended scope.

We are currently working on another tool in the MoTE family named Modelica Diagram Editor (MoDE), which will be used for the graphical composition of Modelica models and could also be used handle the connector placement (Hoppe, 2016).

5.4 Inherited Annotations

Modelica allows inheritance of icon annotations. The inherited annotations are currently not displayed in MoVE. This feature was postponed to future versions, because it would require parsing of several files and inspection of inheritance hierarchies.

6 Conclusions

In this paper we presented a new graphical editor for Modelica icon annotations. In contrast to other open source alternatives, the user interface of MoVE is specifically designed to make editing and creating vector graphic icons for Modelica models as easy and fast as creating a vector graphic image with tools such as Inkscape. MoVE builds on the modern platform-independent framework JavaFX. It has many convenience features such as grouping, snap to grid, move to foreground/background, rotation handles, and drawing perfect circles and squares as well as horizontal and vertical lines when holding *Shift*. It is also designed to work well with version control systems so that changes to individual elements can be captured. Except for dynamic elements, it supports every part of the icon definition in the Modelica language specification.

There are many possibilities for future improvement which can be drawn from the feature set of Inkscape such as component and node alignment or the combination and cutting of multiple paths. Ideally, these features could be brought to MoVE by a (partial) import of SVG graphics. This would allow to create icons in Inkscape and convert them into Modelica code so that they are used directly in Modelica models. For this, one would need to define a subset of SVG that is translatable to Modelica and somehow restrict the user in Inkscape to only use this subset. Futhermore, if MoVE should be able to place and display connectors of the model, the parser needs to be extended and additional parts of the model have to be altered.

We hope that this tool can enrich the open source ecosystem of Modelica and will enable more elaborate vector graphic icons for Modelica libraries. MoVE is part of a larger ensemble of tools called MoTE, which also features an integration of Modelica compiler features into a structured text editor.

The projects are open source and hosted on GitHub:
`https://github.com/thm-mote/`

References

Asghar, Syed Adeel et al. (2011). "An Open Source Modelica Graphic Editor Integrated with Electronic Notebooks and Interactive Simulation". In: *Proceedings of the 8th International Modelica Conference*. Dresden, Germany, pp. 739–747.

Dahlström, Erik et al. (2011). *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. W3C Recommendation REC-SVG11-20110816. W3C. URL: `https://www.w3.org/TR/SVG/`.

EPFL and Typesafe, Inc. (2016). *scala-parser-combinators*. GitHub Repository. URL: `https://github.com/scala/scala-parser-combinators` (visited on 12/09/2016).

Frey, Regis (2016). *The model, view, and controller (MVC) pattern relative to the user*. URL: `https://en.wikipedia.org/wiki/File:MVC-Process.svg` (visited on 12/07/2016).

Fritzson, Peter et al. (2005). "The OpenModelica Modeling, Simulation, and Development Environment". In: *Proceedings of the 46th Scandinavian Conference on Simulation and Modeling (SIMS)*. Trondheim, Norway.

GitHub (2016). *Atom*. URL: `https://atom.io` (visited on 11/01/2016).

Hoppe, Marcel (2016). *Modelica Diagram Editor*. URL: `https://github.com/THM-MoTE/MoDE` (visited on 12/20/2016).

Inkscape (2016). *Inkscape — Draw Freely*. URL: `https://inkscape.org` (visited on 12/09/2016).

Justus, Nicola, Marcel Hoppe, and Christopher Schölzel (2017). *Modelica Tool Ensemble (MoTE)*. URL: `https://github.com/thm-mote` (visited on 03/28/2017).

McIlroy, M. D., E. N. Pinson, and B. A. Tague (1978). "Unix Time-Sharing System: Foreword". In: *The Bell System Technical Journal* 57.6, pp. 1899–1904.

Modelica Association (2012). *Modelica - A Unified Object-Oriented Language for Systems Modeling*. Language Specification. Version 3.3.

Pop, Adrian Dan Iosif et al. (2006). "OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging". In: *Proceedings of the 5th International Modelica Conference*. Vienna, Austria, pp. 459–465.

Reenskaug, Trygve (1979). *Thing-Model-View-Editor — An Example from a planningsystem*. technical note. Xerox PARC.

A. Supplements

A.1. Characteristics of mathematical modeling languages that facilitate model reuse in systems biology: A software engineering perspective

Data supplement

Characteristics of mathematical modeling languages that facilitate model reuse in systems biology: A software engineering perspective

Christopher Schölzel* Valeria Blesius Gernot Ernst Andreas Dominik

March 16, 2021

1 Supplementary Note: Accuracy of the modular model with respect to original

1.1 Result: The modular model behaves similar to the original version

Although the modular version of the conduction model presented in Section 2.3 covers the same physiological effects as the original version, the implementation differs not only in structure but also in the mathematical representation. The original model used the elapsed time since the last contraction as a reference for the refractory period and the pacemaker effect of the atrioventricular node (AV node) instead of the elapsed time since the last sinus signal was received. This means that the duration of the refractory period needs to be adjusted. In the old model, the performed check was whether $t - (t_{\text{sinus}} + T_{\text{delay}}) < T_{\text{refrac}}$ where t is the current time stamp, t_{sinus} is the time stamp of the last sinus signal, T_{delay} is the duration of the delay and T_{refrac} is the refractory period. Since the check is now whether $t - t_{\text{sinus}} < T'_{\text{refrac}}$ one can deduce that if the same behavior is desired, T'_{refrac} should equal $T_{\text{refrac}} + T_{\text{delay}}$, that is T_{refrac} must be increased by the average delay duration. The pacemaker component does not have to be changed at all, because, although the pacemaker signal is delayed, the pacemaker clock is also started earlier. Effectively the delay duration is added and then subtracted from the resulting time stamp of the next contraction.

Supplementary Figure 2 shows a comparison of the resulting interbeat intervals (i.e. the time passed between two contractions) for both models with input of varying frequency. For the most part both versions behave identically. Only when the sinus cycle length drops below the refractory period one can see a difference in the plots. In these areas the interbeat intervals of the modular model fluctuate with a higher amplitude and lower frequency compared to the original Seidel-Herzel model (SHM).

1.2 Discussion

The simulation of the modular model shows a similar but not identical behavior with respect to the original version. Increased amplitude and lower frequency of fluctuations during very fast sinus rhythms are explained by the use of the average delay duration to adjust the refractory period of the AV node in the modular model. In the original model the delay duration T_{delay} varies over time. As shown in Section 1.1, this variable also plays a role in the check for the refractory period, making the effective duration of the refractory period itself time-dependent. It is not clear if this behavior

is intentional in the SHM or if it is a side effect of other design choices. If this behavior is desired, it can be emulated in the modular version by making the variable `d_refrac` in the component `RefractoryGate` time dependent much like the duration in `ConductionDelay`. Physiologically the refractory period indeed changes when the sinus frequency is increased, but the effect is a decrease instead of an increase as in the monolithic model [1]. It can therefore be said that the modular design both helps to identify plausibility issues and can be more easily adapted to the biological reality.

2 Supplementary Note: Extension of the cardiac conduction system with a trigger for PVC

2.1 Result: The new model structure facilitates the extension with a trigger for premature ventricular contractions

To show that the modular structure and the use of a Modular, Descriptive, human-Readable, Open, Graphical, and Hybrid (MoDROGH) language facilitate reuse and extension we added a trigger for premature ventricular contractions (PVCs) to the model. When we tried to do add this extension to the monolithic model, we struggled to identify which equations would have to change and where exactly the effect of a PVC should be incorporated. In the modular model, it is now possible to separately address the effect of a PVC on each individual component.

PVCs arise if some part of the ventricular tissue generates a signal without stimulation from the AV node. This can happen in a healthy individual, but the heart rate response to such an ectopic beat can be used as a risk indicator during pathological conditions [2].

An ectopic beat in the ventricles leads to a stimulation of the AV node that travels back upwards to the sinoatrial node (SA node) either cancelling out an oncoming downward signal or (in rare cases) resetting the clock of the SA node. Therefore, the *correct* way to model a PVC would be to include components that are bidirectional.

To keep it simple, the unidirectional components are used and it is assumed that a PVC will always reset the pacemaker and refractory time of the AV node but never reach the SA node. We modeled this by extending the components `RefractoryGate` and `ConductionDelay` with a "reset" input similar to the one that already exists for the `Pacemaker` component.

With these changes, there could still be two beats arbitrarily close to each other when a PVC is triggered right after a normal beat. Therefore, we also modeled the refractory behavior of the ventricles themselves. The only change needed for this was the addition of a second instance of the already existing `RefractoryGate` component that receives input from the PVC trigger and the delay component (combined with a logical OR). The output of this additional component was used to ensure that the reset of the AV node would only happen if the PVC actually did trigger a contraction. This was achieved by an additional logical AND gate with input from the PVC signal and the contraction output. A graphical representation of the resulting model can be seen in Figure 2 in the main article.

2.2 Result: The PVC extension shows plausible results

The behavior of the resulting model is shown in Supplementary Figure 3. For a normal sinus rhythm of 75 bpm the model behaves as expected. When a PVC happens while a beat is delayed,

it replaces the normal beat, leading to one interbeat interval that is shorter than normal followed by one interval that is larger than normal by the same magnitude. The same behavior can be observed for a PVC happening directly before a sinus signal is issued. A PVC that follows a normal beat within the ventricular refractory period is completely ignored and a PVC right between two normal beats leads to two interbeat intervals that are shorter than normal since all three beats (two normal, one ectopic) lead to a contraction.

To test the behavior of the AV node in the presence of ectopic beats, the experiment was repeated without any sinus signal. Here, the behavior is the same for PVCs while an AV signal is delayed, during the ventricular refractory period and directly before an AV signal. A PVC right between two normal beats does not result in two reduced interbeat intervals but in one reduced and one increased interval. This is due to the reset of the pacemaker clock that will issue the next signal after the pacemaker period has passed. This signal has to travel through the delay component, which increases the interval duration.

2.3 Discussion

The PVC model also behaves as expected. In the simulation with a sinus frequency of 75 bpm, the stimulations very close to a sinus signal effectively replace that signal, leading to a short coupling interval and a long compensatory pause. Stimulations during the refractory period are correctly ignored and a stimulation between two sinus signals is just treated as an additional contraction leading to a series of two subsequent interbeat intervals that are shorter than the sinus cycle length. The third interbeat interval is also a little shorter than normal, because, although the cycle length has not changed, the sinus signal immediately after the PVC has an increased delay duration shifting it closer to its successor. In the simulation without a sinus signal, the only qualitative difference can be observed in the case of an additional signal in between two AV node signals. The first interbeat interval is reduced, but the second interval is actually increased. Physiologically this can be explained by the signal of the PVC traveling upwards stimulating the AV node in the same way a signal from the sinus node would do. The next contraction can therefore only happen after the normal AV cycle length *and* the delay duration has passed.

2.4 Methods: PVC model code

For the PVC model, we will now only discuss the changes required for the existing components. The full code can be found in Supplementary Listing 1–27.

In the `RefractoryGate`, the condition **`when`** `outp` simply has to be replaced with **`when`** `outp` **`or`** `reset`. For `ConductionDelay` the process is a little more complicated since oncoming signals have to be canceled. This is achieved by temporarily setting `t_next` to a very large value (larger than the total simulation time). The equations section changes as follows:

```
delay_passed = time > t_next or t_next > 1e99;
outp = edge(delay_passed);
when pre(reset) or (inp and pre(delay_passed)) then
  d_outp_inp = time - pre(t_last);
end when;
when pre(reset) then
  t_next = 1e100;
elsewhen inp and pre(delay_passed) then
  t_next = time + d_delay;
end when;
when outp or reset then
  t_last = time;
```

```
  end when;
```

The resulting extended model `ModularConductionX` looks as follows:

```
model ModularConductionX
  extends UnidirectionalConductionComponent(outp.fixed=true);
  extends SHMConduction.Icons.Heart;
  import SI = Modelica.SIunits;
  RefractoryGateX refrac_av(T_refrac=0.364)
    "refractory component for AV node" annotation(...);
  Pacemaker pace_av(T=1.7)
    "pacemaker effect of AV node" annotation(...);
  AVConductionDelayX delay_sa_v
    "delay between SA node and ventricles" annotation(...);
  RefractoryGate refrac_v(T_refrac=0.2)
    "refractory component for ventricles" annotation(...);
  InstantInput pvc(fixed=true)
    "trigger signal for a PVC" annotation(...);
  discrete SI.Period d_interbeat(start=1, fixed=true)
    "duration of last heart cycle";
  discrete SI.Time cont_last(start=0, fixed=true)
    "time of last contraction";f
  Modelica.Blocks.Logical.Or vcont
    "groups inputs for refrac_v" annotation(...);
  Modelica.Blocks.Logical.Or rpace
    "groups reset signals of pace_av" annotation(...);
  Modelica.Blocks.Logical.And pvc_upward
    "true if we have PVC that travels upward" annotation(...);
equation
  connect(inp, pace_av.inp) annotation(...);
  connect(pace_av.outp, refrac_av.inp) annotation(...);
  connect(refrac_av.outp, delay_sa_v.inp) annotation(...);
  connect(delay_sa_v.outp, vcont.u1) annotation(...);
  connect(refrac_av.outp, rpace.u2) annotation(...);
  connect(vcont.y, refrac_v.inp) annotation(...);
  connect(refrac_v.outp, outp) annotation(...);
  connect(outp, pvc_upward.u1) annotation(...);
  connect(pvc, pvc_upward.u2) annotation(...);
  connect(pvc_upward.y, refrac_av.reset) annotation(...);
  connect(pvc_upward.y, delay_sa_v.reset) annotation(...);
  connect(pvc_upward.y, rpace.u1) annotation(...);
  connect(rpace.y, pace_av.reset) annotation(...);
  connect(pvc, vcont.u2) annotation(...);
  when outp then
    d_interbeat = time - pre(cont_last);
    cont_last = time;
  end when;
end ModularConductionX;
```

2.5 Methods: PVC experiment setup

The simulation experiment for the PVC model is also defined as a Modelica class. Here the variable `d_interbeat` was plotted once with `with_sinus = true` and once with `with_sinus = false`:

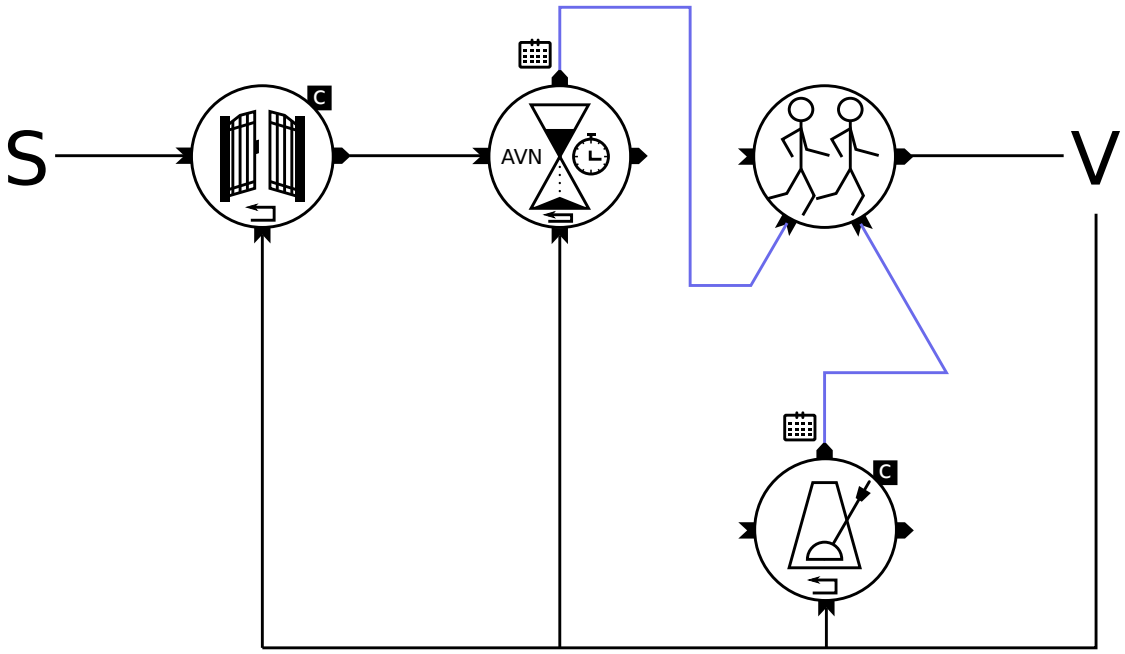
```
model PVCExample
    import SI = Modelica.SIunits;
  discrete SI.Time sig_last(start=0, fixed=true)
    "time where last SA/AV signal was received";
  Integer count_sig(start=0, fixed=true) "counts SA/AV signals";
  parameter Boolean with_sinus = true
    "if true, a sinus signal is applied, otherwise only the AV node is active";
  parameter SI.Period normal_interval = if with_sinus
    then 0.8 else con.pace_av.period "normal cycle duration without PVC";
  SI.Duration t_since_sig = time - pre(sig_last)
    "time since last signal from SA/AV node";
  Boolean pvc_a = pre(count_sig)==5 and t_since_sig > con.delay_sa_v.d_avc0/2
```

```
    "timer for PVC a): while 6th beat is delayed";
  Boolean pvc_b = pre(count_sig)==12 and t_since_sig > con.refrac_av.d_refrac/2
    "timer for PVC b): after 12th beat within refractory period";
  Boolean pvc_c = pre(count_sig)==19 and t_since_sig > normal_interval / 2
    "timer for PVC c): between 19th and 20th beat (after refractory period)";
  Boolean pvc_d = pre(count_sig)==26 and
    t_since_sig > normal_interval - con.delay_sa_v.d_avc0 / 2
    "timer for PVC d): just before the 27th beat was signalled";
  Boolean trigger(start=false, fixed=true) = pvc_a or pvc_b or pvc_c or pvc_d
    "pvc trigger signal";
equation
  con.pvc = edge(trigger);
  if with_sinus then
    con.inp = sample(0, normal_interval) "undisturbed normal sinus rhythm";
  else
    con.inp = false "no sinus, only AV node";
  end if;
  when con.refrac_av.outp then
    count_sig = pre(count_sig) + 1;
    sig_last = time;
  end when;
  annotation(
    experiment(
      StartTime = 0, StopTime = 55,
      Tolerance = 1e-6, Interval = 0.002
    ),
    __OpenModelica_simulationFlags(lv = "LOG_STATS", s = "dassl")
  );
end PVCExample;
```

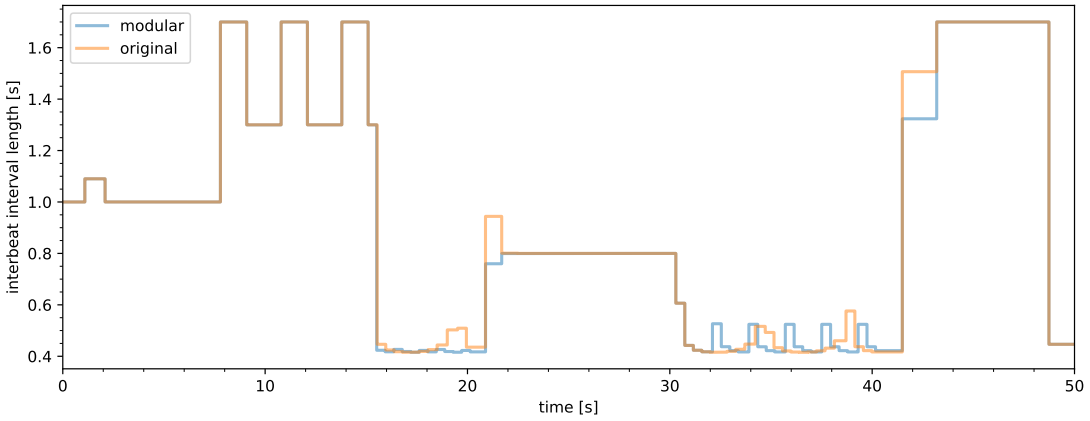
References

1. Mendez, C., Gruhzit, C. C. & Moe, G. K. Influence of Cycle Length upon Refractory Period of Auricles, Ventricles, and A-V Node in the Dog. *American Journal of Physiology-Legacy Content* **184,** 287–295 (1956).
2. Schmidt, G. *et al.* Heart-Rate Turbulence after Ventricular Premature Beats as a Predictor of Mortality after Acute Myocardial Infarction. *The Lancet* **353,** 1390–1396 (1999).
3. Schölzel, C. *CSchoel/Shm-Conduction: Release v1.1.1* version v1.1.1. Zenodo, 2021. `https://doi.org/10.5281/zenodo.4585654` (2021).

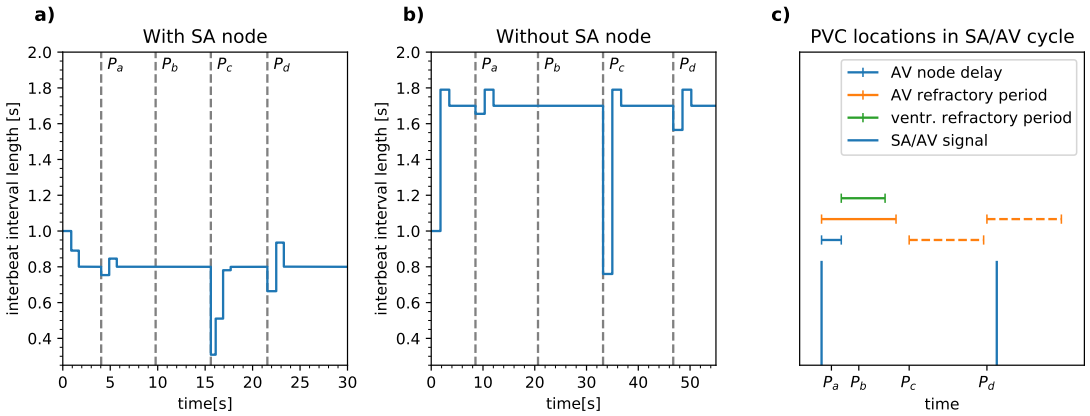
Supplementary Figures



Supplementary Figure 1: Diagram of the monolithic model of the cardiac conduction system in the SHM. Sinus signals (S) are only propagated if enough time has passed since the last ventricular contraction (V). Propagated signals are delayed by a duration that also depends on the amount of time that has passed since the last ventricular contraction. The delay does result in a signal but in a scheduled time stamp that indicates the next time when a ventricular contraction could happen. This time stamp is compared to the time stamp produced by the pacemaker effect (a constant offset added to the time of the last ventricular contraction). The smaller time stamp wins the race condition (running stick figures symbol) and will trigger the next contraction while the larger time stamp is ignored.
This diagram illustrates why we had to change the model structure for our modular version. If we translated the monolithic version one-to-one into modules, we would have ended up with this quite convoluted diagram that does not separate the individual physiological effects very well. This would have had negative effects on the explanatory power and physiological soundness of the result.



Supplementary Figure 2: Comparison of the interbeat intervals of the original conduction model (orange) and the modular version (blue). The plot was obtained with an artificial sinus signal that switches its cycle duration every ten seconds according to the following schedule: 1 s, 3 s, 0.05 s, 0.8 s, 0.2 s, 1.8 s. This schedule was chosen to cover a large range with different cycle durations that are a) below the refractory period of the AV node (0.2 s, 0.05 s), b) above the refractory period, but below the pacemaker period of the AV node (0.8 s, 1 s), or c) above the pacemaker period (3 s, 1.8 s). The schedule is not in any particular order so that both reactions to a sudden increase and a sudden decrease in sinus frequency can be observed. Only the cycle durations of 0.05 s and 0.2 s produce qualitative differences. This plot was generated with shm-conduction version v1.1.1 [3] using OpenModelica v1.17.0-dev.344+gc8233fa62a.



Supplementary Figure 3: Interbeat intervals of the modular model with PVCs with a) a normal sinus rhythm of 75 bpm and b) without sinus signal. Part c) shows the location of the ectopic beats in the normal cycle duration. PVCs are triggered with different prematurity: P_a while a signal is delayed, P_b within the ventricular refractory period, P_c just between two beats, P_d just before a beat would be triggered by the SA or AV node. This plot was generated with shm-conduction version v1.1.1 [3] using OpenModelica v1.17.0-dev.344+gc8233fa62a.

3 Supplementary Tables

				Python						Julia	
	MATLAB[1]	SBML	CellML	pySB	PySCeS	SimuPy	PyDSTool	Antimony	Modelica	Modia	DiffEq.jl
modular	✓	(✓)	✓	(✓)	×	(✓)	×	✓	✓	✓	×
instantiation / import	✓	✓	✓	(✓)	×	✓	×	✓	✓	✓	×
object orientation	✓	×	×	×	×	×	×	×	✓	✓	×
multiple inheritance	(✓)	×	×	×	×	×	×	×	✓	✓	×
interface definition	✓	(✓)	✓	(✓)	×	✓	×	✓	✓	✓	×
grouping of interface variables	✓	(✓)	×	×	×	×	×	×	✓	✓	×
overwrite variables on import	×	(✓)	×	×	×	×	×	✓	✓	✓	×
overwrite equations on import	×	(✓)	×	×	×	×	×	✓	(✓)	(✓)	×
delete variables on import	×	(✓)	×	×	×	×	×	✓	×	×	×
delete equations on import	×	(✓)	×	×	×	×	×	✓	×	×	×
declarative	✓	✓	✓	(✓)	(✓)	(✓)	(✓)	✓	✓	✓	(✓)
mathematical semantics	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
implicit ODE/DAE	✓	✓	(✓)	×	×	×	×	×	✓	✓	(✓)
unit definition	✓	✓	✓	×	✓	×	×	✓	✓	✓	✓
enforced unit definition	(✓)	×	✓	×	×	×	×	×	×	×	×
ontology support	×	✓	✓	(✓)	×	×	×	×	(✓)	(✓)	×
readable	×	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	✓	✓	(✓)
model files focus on human-readability	× [2]	(✓)	(✓)	✓	✓	(✓)	(✓)	✓	✓	✓	✓
format designed to be written by humans	× [2]	×	×	✓	✓	✓	✓	✓	✓	✓	✓
labels for variables	✓	✓	✓	×	×	×	×	×	✓	✓	×
labels for models	✓	✓	✓	×	×	×	×	×	✓	✓	×
labels for equations	×	✓	✓	×	×	×	×	×	✓	✓	×
rich text documentation	(✓)	✓	✓	×	×	×	×	×	✓	✓	×
open	×	✓	✓	✓	✓	✓	✓	✓	(✓)	✓	✓
open-source language specification	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
open-source compiler	×	✓	✓	✓	✓	✓	✓	✓	(✓)	✓	✓
open-source tools and editors	×	✓	✓	✓	✓	✓	✓	✓	(✓)	✓	✓
platform independent	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
graphical	✓	✓	(✓)	×	×	×	×	×	✓	×	×
image annotation	✓	✓	✓	×	×	×	×	×	✓	×	×
vector graphics annotation	×	✓	×	×	×	×	×	×	✓	×	×
annotations tied to model structure	✓	✓	×	×	×	×	×	×	✓	×	×
drag and drop composition	✓	(✓)	×	×	×	×	×	×	✓	×	×
hybrid	✓	(✓)	(✓)	×	(✓)	(✓)	✓	(✓)	✓	✓	✓
ordinary differential equations (ODE)	✓	✓	✓	(✓)	✓	✓	✓	✓	✓	✓	✓
differential/algebraic equations (DAE)	✓	(✓)	(✓)	×	(✓)	×	✓	×	✓	✓	✓
reinitialization due to discrete events	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓
explicit declaration of discrete variables	✓	×	×	×	×	×	×	×	✓	✓	×
other formalisms (Petri nets, FSA, ...)	✓	×	×	×	×	×	×	×	✓	(✓)	×
cross-language import/export (FMI)	(✓)	×	×	×	×	×	×	×	✓	(✓)	×

[1] using the Simulink environment and the Simscape language

[2] while Simscape is human-readable, Simulink uses a proprietary binary file format

Supplementary Table 1: Evaluation of language candidates with respects to the desirable characteristics established in this paper broken up into individual features. A check mark in parentheses means the language has the respective feature in principle, but not to its full extent or with noticeable drawbacks. This table is an extended version of Table 1 in the main article.

Supplementary Listings: Full code for monolithic and modular conduction models

The code snippets showcased in this paper where simplified for the convenience of the reader. This section of the supplement contains the full code that can also be obtained from GitHub [1]. The online version is the preferred source since it may contain fixes and updates applied after the publication of this paper. However, GitHub may not always be there in the future and therefore we provide this version to be archived along with the paper. Please note that most of the additional lines of code are introduced due to the graphical annotations.

Package structure

Modelica organizes code in packages. These packages can be defined within a single file or, as in this case, in a folder structure where each folder contains a file called `package.mo` that contains the package metadata. This project contains the following packages:

Supplementary Listing 1: `SHMConduction/package.mo`

```
package SHMConduction "modular and monolithic models of the human cardiac conduction system based on
      the PhD thesis of H. Seidel"
annotation(Documentation(info="<html>
  <p>Contains a modular version of the cardiac conduction system in the
  Seidel-Herzel-model (SHM).</p>

  <p>The SHM is a macro-level model of the human baroreflex that was
  originally written by Henrik Seidel in his PhD thesis in the language C
  <a href=\"#ref1\">[1]</a>.
  One part of this model is the cardiac conduction system that transfers
  signals from the sinus node to the ventricles where they trigger a
  ventricular contraction and thus the switch from diastole to systole.
  The cardiac conduction system is mainly controlled by the AV node that
  propagates sinus signals to the ventricles.</p>

  <p>The cardiac conduction submodel incorporates the following physiological
  effects.</p>

  <ul>
    <li><b>Refractory effect</b>: The atrioventricular (AV) node that reveives
    signals from the sinus node has a refractory period during which no
    excitation of the AV node can take place.</li>
    <li><b>Pacemaker effect</b>: If the sinus node does not send any signals
    for a prolonged time period, the AV node sends a signal on its own.</li>
    <li><b>Delay effect</b>: Signals traveling through the AV node are delayed.
    The duration of the delay depends on the time that has passed since the
    last signal has left the AV node.</li>
  </ul>

  <p>This package contains a monolithic version of the original model of the
  cardiac conduction system that was translated to Modelica, a modular version
  with small structural differences and an extension of this modular version
  that features a trigger for premature ventricular contractions (PVCs).</p>

  <p>The structural differences between the monolithic and modular versions
  are the following:</p>

  <table>
    <tr>
      <th>Monolithic version</th><th>Modular version</th>
    </tr>
```

[1] `https://github.com/CSchoel/shm-conduction`

```
    <tr>
      <td>Refractory time starts right after the contraction</td>
      <td>Refractory time starts right after sinus signal</td>
    </tr>
    <tr>
      <td>Pacemaker timer is reset after contraction</td>
      <td>Pacemaker timer is reset when sinus signal is received and AV node
      is not refractory</td>
    </tr>
    <tr>
      <td>Compares scheduled time stamps for next contraction that would be
      triggered by a delayed sinus signal or an intrinsic AV signal</td>
      <td>Every connection between components carries an actual signal;
      no scheduled time stamps are required</td>
    </tr>
  </table>

  <p>The Modelica version of the SHM model is described in detail in
  <a href=\"#ref2\">[2]</a>.
  The modular versions will be published in an upcoming paper
  <a href=\"#ref3\">[2]</a>.</p>

  <p><a name=\"ref1\">[1]</a> H. Seidel, &ldquo;Nonlinear dynamics of physiological
  rhythms,&rdquo; PhD thesis, Technische Universität Berlin, Berlin, Germany, 1997.
  </p>

  <p><a name=\"ref2\">[2]</a> C. Schölzel, A. Goesmann, G. Ernst, and A. Dominik,
  &ldquo;Modeling biology in Modelica: The human baroreflex,&rdquo; in Proceedings of the
  11th International Modelica Conference, Versailles, France, 2015, pp.-
  367376.</p>

  <p><a name=\"ref3\">[3]</a> C. Schölzel, V. Blesius, G. Ernst and
  A. Dominik, &ldquo;Required characteristics for modeling languages in systems
  biology: A software engineering perspective,&rdquo; unpublished.</p>
</html>"));
end SHMConduction;
```

Supplementary Listing 2: `SHMConduction/Components/package.mo`

```
within SHMConduction;
package Components "contains individual modules"
  type InstantSignal = Boolean(quantity="sum of Kronecker deltas")
    "signal that is only true for exact time instants (i.e. that behaves as a sum of Kronecker
        deltas)";
annotation(Documentation(info="<html>
  <p>Contains component models used in the examples.</p>
</html>"));
end Components;
```

Supplementary Listing 3: `SHMConduction/Components/Connectors/package.mo`

```
within SHMConduction.Components;
package Connectors "connector classes used as interfaces between components"
annotation(Documentation(info="<html>
  <p>Contains connector classes that define the interface between
  components.</p>
</html>"));
end Connectors;
```

Supplementary Listing 4: `SHMConduction/Components/PVC/package.mo`

```
within SHMConduction.Components;
package PVC "extended variants of components that allow to simulate premature ventricular
     contractions"
annotation(Documentation(info="<html>
  <p>Contains modified and additional models for simulating premature
```

```
  ventricular contractions (PVCs).</p>
</html>"));
end PVC;
```

Supplementary Listing 5: `SHMConduction/Examples/package.mo`

```
within SHMConduction;
package Examples "contains complete examples that are ready for simulation"
annotation(Documentation(info="<html>
  <p>Contains complete example models that can be simulated.</p>
</html>"));
end Examples;
```

Supplementary Listing 6: `SHMConduction/Icons/package.mo`

```
within SHMConduction;
package Icons "base models that contain no logic but only icon definitions"
annotation(Documentation(info="<html>
  <p>Contains empty classes with annotations that can be extended to use
  the respective icons in a model without cluttering the code.</p>
</html>"));
end Icons;
```

Monolithic model

Supplementary Listing 7: `SHMConduction/Components/MonolithicConduction.mo`

```
within SHMConduction.Components;
model MonolithicConduction "cardiac conduction system of the human heart adapted from the doctorate
    thesis of H. Seidel"
  import SI = Modelica.SIunits;
  input InstantSignal inp(start=false, fixed=true) "the sinus signal";
  output InstantSignal outp(start=false, fixed=true) "true when a contraction is triggered";
  parameter SI.Duration d_refrac = 0.22 "refractory period that has to pass until a signal from the
      sinus node can take effect again";
  parameter SI.Period av_period = 1.7 "av-node cycle duration";
  parameter SI.Duration k_avc_t = 0.78 "sensitivity of the atrioventricular conduction time to the
      time passed since the last ventricular conduction";
  parameter SI.Duration d_avc0 = 0.09 "base value for atrioventricular conduction time";
  parameter SI.Duration tau_avc = 0.11 "reference time for atrioventricular conduction time"; //TODO
       find better description
  parameter SI.Period initial_d_sinus_sinus = 1 "initial value for d_sinus_sinus";
  parameter SI.Period initial_d_interbeat = 1 "initial value for d_interbeat";
  parameter SI.Time initial_cont_last = 0 "initial value for last ventricular contraction time";
  parameter SI.Duration initial_d_delay = 0.15 "initial value for atrioventricular conduction time";
  output Boolean av_contraction "true when the av-node triggers a contraction";
  output Boolean sinus_contraction "true when the sinus node triggers a contraction";
  output Boolean refrac_passed(start=false, fixed=true) "true when the refractory period has passed"
      ;
  discrete output SI.Period d_sinus_sinus "time between the last two sinus signals that did trigger
      a contraction";
  discrete output SI.Period d_interbeat "time between the last two contractions";
protected
  discrete SI.Time cont_last "time of last contraction";
  discrete SI.Duration d_delay "atrioventricular conduction time (delay for sinus signal to trigger
      contraction)";
  SI.Duration since_cont "helper variable; time passed since last contraction";
  Boolean signal_received(start=false, fixed=true) "true, if a sinus signal has already been
      received since the last contraction";
  discrete SI.Time sinus_last "time of last received sinus signal";
  Boolean contraction_event(start=false, fixed=true);
initial equation
  cont_last = initial_cont_last;
  sinus_last = 0;
```

```
  d_sinus_sinus = initial_d_sinus_sinus;
  d_interbeat = initial_d_interbeat;
  d_delay = initial_d_delay;
equation
  signal_received = sinus_last > cont_last;
  refrac_passed = since_cont > d_refrac;
  contraction_event = (av_contraction or sinus_contraction) and refrac_passed "contraction can come
      from av-node or sinus node";
  outp = edge(contraction_event);
  av_contraction = since_cont > av_period "av-node contracts when av_period has passed since last
      contraction";
  sinus_contraction = signal_received and time > sinus_last + d_delay "sinus node contracts when
      d_delay has passed since last sinus signal";
  since_cont = time - cont_last;
  //sinus signal is recognized if refractory period has passed and there is no other sinus signal
      already in effect
  when inp and pre(refrac_passed) and not pre(signal_received) then
    d_delay = d_avc0 + k_avc_t * exp(-since_cont / tau_avc) "schedules next sinus_contraction";
    sinus_last = time "record timestamp of recognized sinus signal";
    d_sinus_sinus = time - pre(sinus_last);
  end when;
  when pre(outp) then
    cont_last = time "record timestamp of contraction";
    d_interbeat = time - pre(cont_last);
  end when;
  annotation(Documentation(info = "<html>
  <p>Models the contraction of the heart as described in Seidel's thesis.</p>
  <p>The model takes into account the following effects:</p>
  <ul>
    <li>There is a refractory period of duration <b>d_refrac</b> after a contraction during which
        signals of the sinus node are ignored.
    <li>If no sinus-induced contraction occurs for a prolonged time span (namely <b>av_period</b>)
        the av-node initiates a contraction by itself.
    <li>When a sinus signal is received, the upper heart contracts pumping the blood from the atrium
         into the ventricles. The systole does only
    begin with the second contraction of the heart. The time period between these two events is
        called the &amp;quot;atrioventricular conduction time&amp;quot;.
  </ul>
  <p><i>Note: The formulas in this model differ from the formulas found in the c-implementation by
      Seidel because OpenModelica is currently
  not capable of handling discrete equation systems. Therefore it was necessary to introduce the
      continuous phases <b>av_phase</b>,
  <b>sinus_phase</b> and <b>refrac_countdown</b>, as well as the continuous variable condition <b>
      signal_received_cont</b>.</i></p>
</html>"));
end MonolithicConduction;
```

Modular model

Supplementary Listing 8: `SHMConduction/Components/Connectors/InstantInput.mo`

```
within SHMConduction.Components.Connectors;
connector InstantInput = input InstantSignal "input with Kronecker delta behavior"
  annotation(
    Icon(
      coordinateSystem(
        preserveAspectRatio= false,
        extent= {{-100,-100},{100,100}}
      ),
      graphics= {
        Polygon(
          origin= {-100,100},
          lineThickness= 5,
          pattern= LinePattern.None,
          points= {{194.32, -100}, {200, -200}, {0, -200}, {108.45, -100}, {0, 0}, {200, 0}},
          fillPattern= FillPattern.Solid,
          fillColor= {0,0,0},
          rotation= 0
```

```
        )
      }
    )
  )
;
```

Supplementary Listing 9: `SHMConduction/Components/Connectors/InstantOutput.mo`

```
within SHMConduction.Components.Connectors;
connector InstantOutput = output InstantSignal "output with Kronecker delta behavior"
  annotation(
    Icon(
      coordinateSystem(
        preserveAspectRatio= false,
        extent= {{-100,-100},{100,100}}
      ),
      graphics= {
        Polygon(
          origin= {-100,100},
          lineThickness= 5,
          pattern= LinePattern.None,
          points= {{200, -100}, {91.55, 0}, {0, 0}, {0, -200}, {91.55, -200}, {200, -100}},
          fillPattern= FillPattern.Solid,
          fillColor= {0,0,0},
          rotation= 0
        )
      }
    )
  )
;
```

Supplementary Listing 10: `SHMConduction/Components/UnidirectionalConductionComponent.mo`

```
within SHMConduction.Components;
partial model UnidirectionalConductionComponent "basic interface class with single input and output"
  import SHMConduction.Components.Connectors.{InstantInput, InstantOutput};
  InstantInput inp "input connector" annotation(
    Placement(
      visible = true,
      transformation(
        origin = {-100, 0},
        extent = {{-10, -10}, {10, 10}}, rotation = 0
      ),
      iconTransformation(
        origin = {-108, 0},
        extent = {{-10, -10}, {10, 10}}, rotation = 0
      )
    )
  );
  InstantOutput outp "output connector" annotation(
    Placement(
      visible = true,
      transformation(
        origin = {102, 0},
        extent = {{-10, -10}, {10, 10}}, rotation = 0
      ),
      iconTransformation(
        origin = {108, 0},
        extent = {{-10, -10}, {10, 10}}, rotation = 0
      )
    )
  );
  annotation(
    Icon(
      coordinateSystem(
        preserveAspectRatio= false,
        extent= {{-100,-100},{100,100}}
```

```
      ),
      graphics= {
        Ellipse(
          origin= {-100,100},
          lineThickness= 2,
          pattern= LinePattern.Solid,
          fillPattern= FillPattern.None,
          extent= {{2,-2},{198,-198}},
          rotation= 0.0
        )
      }
    )
  );
end UnidirectionalConductionComponent;
```

Supplementary Listing 11: `SHMConduction/Components/Resettable.mo`

```
within SHMConduction.Components;
partial model Resettable "base class for all components that need a reset input"
  import SHMConduction.Components.Connectors.InstantInput;
  InstantInput reset "signal that resets internal variables" annotation(
    Placement(
      visible = true,
      transformation(
        origin = {-2, -98},
        extent = {{-10, -10}, {10, 10}}, rotation = 0
      ),
      iconTransformation(
        origin = {0, -108},
        extent = {{-10, -10}, {10, 10}}, rotation = 90
      )
    )
  );
  annotation(
    Icon(
      coordinateSystem(
        preserveAspectRatio= false,
        extent= {{-100,-100},{100,100}}
      ),
      graphics= {
        Line(
          origin= {-100,100},
          pattern= LinePattern.Solid,
          thickness= 1.5,
          arrowSize= 5,
          points= {{82.25, -192.58}, {114.50, -192.58}, {114.50, -180.64}, {81.84, -180.64}},
          arrow= {Arrow.None, Arrow.Open},
          rotation= 0
        )
      }
    )
  );
end Resettable;
```

Supplementary Listing 12: `SHMConduction/Components/Pacemaker.mo`

```
within SHMConduction.Components;
model Pacemaker "pacemaker that can elicit spontaneous signals and transmit incoming signals"
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Metronome;
  extends SHMConduction.Icons.Constant;
  extends Resettable(reset.fixed=true); // resets internal clock
  import SI = Modelica.SIunits;
  parameter SI.Period period = 1 "pacemaker period";
protected
  discrete SI.Time t_next(start=period, fixed=true)
    "scheduled time of next spontaneous beat";
```

```
  InstantSignal spontaneous_signal = time > pre(t_next)
    "signal generated spontaneously by this pacemaker";
equation
  outp = inp or spontaneous_signal;
  when spontaneous_signal or pre(reset) then
    t_next = time + period;
  end when;
end Pacemaker;
```

Supplementary Listing 13: SHMConduction/Components/RefractoryGate.mo

```
within SHMConduction.Components;
model RefractoryGate "refractory gate that can block or transmit signals"
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Gate;
  extends SHMConduction.Icons.Constant;
  import SI = Modelica.SIunits;
  parameter SI.Time t_first = 0 "time of first signal";
  parameter SI.Duration d_refrac = 1 "duration of refractory period";
  Boolean refrac_passed = time - pre(t_last) > d_refrac "true if component is ready to receive a
      signal";
protected
  discrete SI.Time t_last(start=t_first, fixed=true) "time of last output";
equation
  outp = inp and refrac_passed;
  when outp then
    t_last = time;
  end when;
end RefractoryGate;
```

Supplementary Listing 14: SHMConduction/Components/ConductionDelay.mo

```
within SHMConduction.Components;
partial model ConductionDelay "cardiac conduction delay that depends on previous cycle duration"
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Hourglass;
  import SI = Modelica.SIunits;
  discrete SI.Duration d_delay "delay duration";
  Boolean delay_passed(start=false, fixed=true) = time > t_next "if false, there is still a signal
      currently put on hold";
protected
  discrete SI.Duration d_outp_inp(start=0, fixed=true) "time between last output and following
      signal";
  discrete SI.Time t_last(start=0, fixed=true) "time of last output";
  discrete SI.Time t_next(start=-1, fixed=true) "scheduled time of next output";
equation
  outp = edge(delay_passed);
  when inp and pre(delay_passed) then
    d_outp_inp = time - pre(t_last);
    t_next = time + d_delay;
  end when;
  when outp then
    t_last = time;
  end when;
end ConductionDelay;
```

Supplementary Listing 15: SHMConduction/Components/AVConductionDelay.mo

```
within SHMConduction.Components;
model AVConductionDelay "conduction delay between SA node and ventricles"
  extends ConductionDelay;
  import SI = Modelica.SIunits;
  parameter SI.Duration k_avc_t = 0.78 "maximum increase in delay duration";
  parameter SI.Duration d_avc0 = 0.09 "minimal delay duration";
  parameter SI.Duration tau_avc = 0.11 "reference time for delay duration";
  parameter SI.Duration initial_d_avc = 0.15 "initial value for delay duration";
```

```
initial equation
  d_delay = initial_d_avc;
equation
  when inp and pre(delay_passed) then
    d_delay = d_avc0 + k_avc_t * exp(-d_outp_inp/tau_avc);
  end when;
  annotation(
    Icon(
      coordinateSystem(
        preserveAspectRatio= false,
        extent= {{-100,-100},{100,100}}
      ),
      graphics= {
        Text(
          origin = {-55, 1},
          extent = {{-21, 19}, {27, -23}},
          textString = "AVN"
        )
      }
    )
  );
end AVConductionDelay;
```

Supplementary Listing 16: SHMConduction/Components/ModularConduction.mo

```
within SHMConduction.Components;
model ModularConduction "modular version of the model of the cardiac conduction system by H. Seidel"
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Heart;
  import SI = Modelica.SIunits;
  RefractoryGate refrac_av(d_refrac=0.364) "refractory component for AV node" annotation(
    Placement(
      visible = true,
      transformation(
        origin = {6, 5.77316e-15},
        extent = {{-20, -20}, {20, 20}}, rotation = 0
      )
    )
  );
  Pacemaker pace_av(period=1.7) "pacemaker effect of AV node" annotation(
    Placement(
      visible = true,
      transformation(
        origin = {-52, 3.77476e-15},
        extent = {{-20, -20}, {20, 20}}, rotation = 0
      )
    )
  );
  AVConductionDelay delay_sa_v "total delay between SA node and ventricles" annotation(
    Placement(
      visible = true,
      transformation(
        origin = {62, 3.55271e-15},
        extent = {{-20, -20}, {20, 20}}, rotation = 0
      )
    )
  );
  discrete SI.Duration d_interbeat(start=1, fixed=true) "duration of last heart cycle (interbeat
      interval)";
  discrete SI.Time cont_last(start=0, fixed=true) "time of last contraction";
equation
  connect(inp, pace_av.inp) annotation(
    Line(thickness = 1, points = {{-74, 0}, {-96, 0}, {-96, 0}, {-100, 0}})
  );
  connect(pace_av.outp, refrac_av.inp) annotation(
    Line(thickness = 1, points = {{-30, 0}, {-16, 0}, {-16, 0}, {-16, 0}})
  );
  connect(refrac_av.outp, pace_av.reset) annotation(
```

```
    Line(thickness = 1, points = {{28, 0}, {34, 0}, {34, -44}, {-52, -44}, {-52, -22}, {-52, -22}})
  );
  connect(refrac_av.outp, delay_sa_v.inp) annotation(
    Line(thickness = 1, points = {{28, 0}, {42, 0}, {42, 0}, {40, 0}})
  );
  connect(delay_sa_v.outp, outp) annotation(
    Line(thickness = 1, points = {{84, 0}, {102, 0}, {102, 0}, {102, 0}})
  );
  when outp then
    d_interbeat = time - pre(cont_last);
    cont_last = time;
  end when;
  annotation(
    Icon(
      graphics = {
        Line(
          origin = {-75, 5},
          points = {{-17, -5}, {17, 5}},
          arrow = {Arrow.None, Arrow.Open},
          thickness = 1,
          arrowSize = 5
        ),
        Line(
          origin = {75, -20},
          points = {{-19, -18}, {19, 18}},
          arrow = {Arrow.None, Arrow.Open},
          thickness = 1,
          arrowSize = 5
        )
      }
    )
  );
end ModularConduction;
```

PVC extension

Supplementary Listing 17: `SHMConduction/Components/PVC/ConductionDelayX.mo`

```
within SHMConduction.Components.PVC;
partial model ConductionDelayX "resettable variant of ConductionDelay (resetting cancels delayed
     signals)"
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Hourglass;
  extends Resettable(reset.fixed=true); // cancels a signal that is currently on hold
  import SI = Modelica.SIunits;
  discrete SI.Duration d_delay "delay duration";
  Boolean delay_passed(start=false, fixed=true) = time > t_next or t_next > 1e99 "if false, there is
       still a signal currently put on hold";
protected
  discrete SI.Duration d_outp_inp(start=0, fixed=true) "time between last output and following
      signal";
  discrete SI.Time t_last(start=0, fixed=true) "time of last output";
  discrete SI.Time t_next(start=-1, fixed=true) "scheduled time of next output";
equation
  outp = edge(delay_passed);
  when pre(reset) or (inp and pre(delay_passed)) then
    d_outp_inp = time - pre(t_last);
  end when;
  when pre(reset) then
    t_next = 1e100;
  elsewhen inp and pre(delay_passed) then
    t_next = time + d_delay;
  end when;
  when outp or reset then
    t_last = time;
  end when;
end ConductionDelayX;
```

Supplementary Listing 18: SHMConduction/Components/PVC/AVConductionDelayX.mo

```
within SHMConduction.Components.PVC;
model AVConductionDelayX "resettable variant of AVConductionDelay (resetting cancels delayed signals
    )"
  extends ConductionDelayX;
  import SI = Modelica.SIunits;
  parameter SI.Duration k_avc_t = 0.78 "maximum increase in delay duration";
  parameter SI.Duration d_avc0 = 0.09 "minimal delay duration";
  parameter SI.Duration tau_avc = 0.11 "reference time for delay duration";
  parameter SI.Duration initial_d_avc = 0.15 "initial value for delay duration";
initial equation
  d_delay = initial_d_avc;
equation
  when inp and pre(delay_passed) then
    d_delay = d_avc0 + k_avc_t * exp(-d_outp_inp/tau_avc);
  end when;
  annotation(
    Icon(
      coordinateSystem(
        preserveAspectRatio= false,
        extent= {{-100,-100},{100,100}}
      ),
      graphics= {
        Text(
          origin = {-55, 1},
          extent = {{-21, 19}, {27, -23}},
          textString = "AVN"
        )
      }
    )
  );
end AVConductionDelayX;
```

Supplementary Listing 19: SHMConduction/Components/PVC/RefractoryGateX.mo

```
within SHMConduction.Components.PVC;
model RefractoryGateX "resettable variant of RefractoryGate"
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Gate;
  extends SHMConduction.Icons.Constant;
  extends Resettable; // resets internal clock
  import SI = Modelica.SIunits;
  parameter SI.Time t_first = 0 "time of first signal";
  parameter SI.Duration d_refrac = 1 "refractory period";
  Boolean refrac_passed = time - pre(t_last) > d_refrac "true if component is ready to receive a
      signal";
protected
  discrete SI.Time t_last(start=t_first, fixed=true) "time of last output";
equation
  outp = inp and refrac_passed;
  when outp or reset then
    t_last = time;
  end when;
end RefractoryGateX;
```

Supplementary Listing 20: SHMConduction/Components/PVC/ModularConductionX.mo

```
within SHMConduction.Components.PVC;
model ModularConductionX "cardiac conduction system with trigger for PVCs"
  extends UnidirectionalConductionComponent(outp.fixed=true);
  // outp is used in a when equation, so we need an initial value
  extends SHMConduction.Icons.Heart;
  import SHMConduction.Components.Connectors.InstantInput;
  import SI = Modelica.SIunits;
  RefractoryGateX refrac_av(d_refrac=0.364) "refractory component for AV node" annotation(
    Placement(
      visible = true,
```

```
    transformation(
      origin = {-20, 0},
      extent = {{-10, -10}, {10, 10}}, rotation = 0
    )
  )
);
Pacemaker pace_av(period=1.7) "pacemaker effect of AV node" annotation(
  Placement(
    visible = true,
    transformation(
      origin = {-60, 0},
      extent = {{-10, -10}, {10, 10}}, rotation = 0
    )
  )
);
AVConductionDelayX delay_sa_v "total delay between SA node and ventricles" annotation(
  Placement(
    visible = true,
    transformation(
      origin = {20, 0},
      extent = {{-10, -10}, {10, 10}}, rotation = 0
    )
  )
);
RefractoryGate refrac_v(d_refrac=0.2) "refractory component for ventricles" annotation(
  Placement(
    visible = true,
    transformation(
      origin = {60, 0},
      extent = {{-10, -10}, {10, 10}}, rotation = 0
    )
  )
);
discrete SI.Period d_interbeat(start=1, fixed=true) "duration of last heart cycle";
discrete SI.Time cont_last(start=0, fixed=true) "time of last contraction";
InstantInput pvc(fixed=true) "trigger signal for a PVC" annotation(
  Placement(
    visible = true,
    transformation(
      origin = {76, -76},
      extent = {{-10, -10}, {10, 10}}, rotation = 135
    ),
    iconTransformation(
      origin = {76, -76},
      extent = {{-10, -10}, {10, 10}}, rotation = 135
    )
  )
);
Modelica.Blocks.Logical.Or vcont "groups inputs for refrac_v" annotation(
  Placement(
    visible = true,
    transformation(
      origin = {42, -36},
      extent = {{-10, -10}, {10, 10}}, rotation = 0
    )
  )
);
Modelica.Blocks.Logical.Or rpace "groups reset signals of pace_av" annotation(
  Placement(
    visible = true,
    transformation(
      origin = {-60, -38},
      extent = {{-10, -10}, {10, 10}}, rotation = 0
    )
  )
);
Modelica.Blocks.Logical.And pvc_upward "true if we have PVC that travels upward" annotation(
  Placement(
    visible = true,
```

```
      transformation(
        origin = {20, -72},
        extent = {{10, -10}, {-10, 10}}, rotation = 0
      )
    )
  );
equation
  connect(inp, pace_av.inp) annotation(
    Line(points = {{-100, 0}, {-72, 0}, {-72, 0}, {-70, 0}})
  );
  connect(pace_av.outp, refrac_av.inp) annotation(
    Line(points = {{-50, 0}, {-30, 0}, {-30, 0}, {-30, 0}})
  );
  connect(refrac_av.outp, delay_sa_v.inp) annotation(
    Line(points = {{-10, 0}, {10, 0}, {10, 0}, {10, 0}})
  );
  connect(delay_sa_v.outp, vcont.u1) annotation(
    Line(points = {{30, 0}, {34, 0}, {34, -22}, {16, -22}, {16, -36}, {30, -36}, {30, -36}})
  );
  connect(refrac_av.outp, rpace.u2) annotation(
    Line(points = {{-10, 0}, {-6, 0}, {-6, -58}, {-78, -58}, {-78, -46}, {-72, -46}, {-72, -46}})
  );
  connect(vcont.y, refrac_v.inp) annotation(
    Line(points = {{54, -36}, {60, -36}, {60, -12}, {42, -12}, {42, 0}, {50, 0}, {50, 0}}, color =
        {255, 0, 255})
  );
  connect(refrac_v.outp, outp) annotation(
    Line(points = {{70, 0}, {98, 0}, {98, 0}, {102, 0}})
  );
  connect(outp, pvc_upward.u1) annotation(
    Line(points = {{102, 0}, {84, 0}, {84, -54}, {50, -54}, {50, -72}, {32, -72}, {32, -72}})
  );
  connect(pvc, pvc_upward.u2) annotation(
    Line(points = {{76, -76}, {50, -76}, {50, -80}, {32, -80}, {32, -80}})
  );
  connect(pvc_upward.y, refrac_av.reset) annotation(
    Line(points = {{10, -72}, {-20, -72}, {-20, -10}, {-20, -10}}, color = {255, 0, 255})
  );
  connect(pvc_upward.y, delay_sa_v.reset) annotation(
    Line(points = {{10, -72}, {2, -72}, {2, -18}, {20, -18}, {20, -10}, {20, -10}}, color = {255, 0,
        255})
  );
  connect(pvc_upward.y, rpace.u1) annotation(
    Line(points = {{10, -72}, {-84, -72}, {-84, -38}, {-72, -38}, {-72, -38}}, color = {255, 0,
        255})
  );
  connect(rpace.y, pace_av.reset) annotation(
    Line(points = {{-48, -38}, {-44, -38}, {-44, -16}, {-60, -16}, {-60, -10}, {-60, -10}}, color =
        {255, 0, 255})
  );
  connect(pvc, vcont.u2) annotation(
    Line(points = {{76, -76}, {70, -76}, {70, -50}, {16, -50}, {16, -44}, {30, -44}})
  );
  when outp then
    d_interbeat = time - pre(cont_last);
    cont_last = time;
  end when;
  annotation(
    Icon(
      graphics = {
        Line(
          origin = {-75, 5},
          points = {{-17, -5}, {17, 5}},
          arrow = {Arrow.None, Arrow.Open},
          thickness = 1,
          arrowSize = 5
        ),
        Line(
          origin = {75, -20},
```

```
          points = {{-19, -18}, {19, 18}},
          arrow = {Arrow.None, Arrow.Open},
          thickness = 1,
          arrowSize = 5
        ),
        Line(
          origin = {60, -64},
          points = {{6, 0}, {-6, 0}},
          arrow = {Arrow.None, Arrow.Open},
          thickness = 1,
          arrowSize = 5
        )
      }
    )
  );
end ModularConductionX;
```

Simulation examples

Supplementary Listing 21: `SHMConduction/Examples/ModularExample.mo`

```
within SHMConduction.Examples;
// TODO: add ontolgy links with annotation(__Ontology(foo="bar"))
model ModularExample "experiment to compare Conduction and ModularConduction"
  SHMConduction.Components.ModularConduction modC "modular contraction model";
  SHMConduction.Components.MonolithicConduction monC "original monolithic contraction model";
equation
  modC.inp = monC.inp;
  if time < 5 then
    monC.inp = sample(0,1);
  elseif time < 15 then
    monC.inp = sample(0,3);
  elseif time < 20 then
    // during Afib, atrial impulses occur at up to 600/min => with distance 0.1s
    // source: https://my.clevelandclinic.org/health/diseases/16765-atrial-fibrillation-afib
    monC.inp = sample(0,0.05);
  elseif time < 30 then
    monC.inp = sample(0,0.8);
  elseif time < 40 then
    monC.inp = sample(0,0.2);
  else
    monC.inp = sample(0,1.8);
  end if;
  annotation(
    experiment(StartTime = 0, StopTime = 50, Tolerance = 1e-6, Interval = 0.002),
    __OpenModelica_simulationFlags(lv = "LOG_STATS", s = "dassl")
  );
end ModularExample;
```

Supplementary Listing 22: `SHMConduction/Examples/PVCExample.mo`

```
within SHMConduction.Examples;
model PVCExample "experiment to test response of ModularConductionX to PVCs"
  SHMConduction.Components.PVC.ModularConductionX con;
  import SI = Modelica.SIunits;
  discrete SI.Time sig_last(start=0, fixed=true) "time where last SA/AV signal was received";
  Integer count_sig(start=0, fixed=true) "counts SA/AV signals";
  parameter Boolean with_sinus = true "if true, a sinus signal is applied, otherwise only the AV
      node is active";
  parameter SI.Period normal_interval = if with_sinus then 0.8 else con.pace_av.period "normal cycle
       duration without PVC";
  SI.Duration t_since_sig = time - pre(sig_last) "time since last signal from SA/AV node";
  Boolean pvc_a = pre(count_sig) == 5 and t_since_sig > con.delay_sa_v.d_avc0 / 2
    "timer for PVC a): while 6th beat is delayed";
  Boolean pvc_b = pre(count_sig) == 12 and t_since_sig > con.refrac_av.d_refrac / 2
    "timer for PVC b): after 12th beat within refractory period";
  Boolean pvc_c = pre(count_sig) == 19 and t_since_sig > normal_interval / 2
```

```
    "timer for PVC c): between 19th and 20th beat (after refractory period)";
  Boolean pvc_d = pre(count_sig) == 26 and
    t_since_sig > normal_interval - con.delay_sa_v.d_avc0 / 2
    "timer for PVC d): just before the 27th beat was signalled";
  Boolean trigger(start=false, fixed=true) = pvc_a or pvc_b or pvc_c or pvc_d "pvc trigger signal";
equation
  con.pvc = edge(trigger);
  if with_sinus then
    con.inp = sample(0, normal_interval) "undisturbed normal sinus rhythm";
  else
    con.inp = false "no sinus, only AV node";
  end if;
  when con.refrac_av.outp then
    count_sig = pre(count_sig) + 1;
    sig_last = time;
  end when;
  annotation(
    experiment(StartTime = 0, StopTime = 55, Tolerance = 1e-6, Interval = 0.002),
    __OpenModelica_simulationFlags(lv = "LOG_STATS", s = "dassl")
  );
end PVCExample;
```

Icons

Supplementary Listing 23: `SHMConduction/Icons/Constant.mo`

```
within SHMConduction.Icons;
model Constant "small box with a C in upper right corner"
    annotation(
        Icon(
            coordinateSystem(
                preserveAspectRatio= false,
                extent= {{-100,-100},{100,100}}
            ),
            graphics= {
                Polygon(
                    origin= {-100,100},
                    lineThickness= 0.56,
                    pattern= LinePattern.None,
                    points= {{163.21, -24.39}, {163.21, -2.15}, {196.78, -2.15}, {196.78, -36.36},
                         {174.62, -36.36}},
                    fillPattern= FillPattern.Solid,
                    fillColor= {0,0,0},
                    rotation= 0
                ),
                Text(
                    origin = {81, 84},
                    lineColor = {255, 255, 255},
                    extent = {{-11, 22}, {11, -22}},
                    textString = "c"
                )
            }
        )
    );
end Constant;
```

Supplementary Listing 24: `SHMConduction/Icons/Gate.mo`

```
within SHMConduction.Icons;
model Gate "gate icon for refractory period"
  annotation(
    Icon(
      coordinateSystem(
        preserveAspectRatio= false,
        extent= {{-100,-100},{100,100}}
      ),
      graphics= {
```

```
Rectangle(
  origin= {-100,100},
  lineThickness= 1,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {0,0,0},
  extent= {{42.33,-42.01},{55.18,-160.69}},
  rotation= 0
),
Polygon(
  origin= {-100,100},
  lineThickness= 2,
  pattern= LinePattern.Solid,
  points= {{69.55, -41.25}, {49.33, -57.13}, {49.33, -153.89}, {89.79, -142.17}, {89.79,
      -35.96}},
  fillPattern= FillPattern.None,
  rotation= 0
),
Rectangle(
  origin= {-100,100},
  lineThickness= 1,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {0,0,0},
  extent= {{145.47,-42.01},{158.32,-160.69}},
  rotation= 0
),
Polygon(
  origin= {-100,100},
  lineThickness= 2,
  pattern= LinePattern.Solid,
  points= {{131.10, -41.25}, {151.33, -57.13}, {151.33, -153.89}, {110.86, -142.17},
      {110.86, -35.96}},
  fillPattern= FillPattern.None,
  rotation= 0
),
Polygon(
  origin= {-100,100},
  lineThickness= 1,
  pattern= LinePattern.None,
  points= {{90.34, -77}, {94.87, -75.65}, {94.87, -93.03}, {89.96, -94.54}},
  fillPattern= FillPattern.Solid,
  fillColor= {0,0,0},
  rotation= 0
),
Line(
  origin= {-100,100},
  pattern= LinePattern.Solid,
  rotation= 0,
  points= {{69.55, -41.25}, {69.55, -148.03}},
  thickness= 2
),
Line(
  origin= {-100,100},
  pattern= LinePattern.Solid,
  rotation= 0,
  points= {{79.67, -37.80}, {79.67, -144.02}},
  thickness= 2
),
Line(
  origin= {-100,100},
  pattern= LinePattern.Solid,
  rotation= 0,
  points= {{59.56, -48.82}, {59.56, -150.20}},
  thickness= 2
),
Line(
  origin= {-100,100},
  pattern= LinePattern.Solid,
```

```
          rotation= 0,
          points= {{49.33, -62.89}, {89.79, -50.86}},
          thickness= 2
        ),
        Line(
          origin= {-100,100},
          pattern= LinePattern.Solid,
          rotation= 0,
          points= {{89.79, -132.65}, {49.33, -145.21}},
          thickness= 2
        ),
        Line(
          origin= {-100,100},
          pattern= LinePattern.Solid,
          rotation= 0,
          points= {{131.10, -41.25}, {131.10, -148.03}},
          thickness= 2
        ),
        Line(
          origin= {-100,100},
          pattern= LinePattern.Solid,
          rotation= 0,
          points= {{120.98, -37.80}, {120.98, -144.02}},
          thickness= 2
        ),
        Line(
          origin= {-100,100},
          pattern= LinePattern.Solid,
          rotation= 0,
          points= {{141.10, -48.82}, {141.10, -150.20}},
          thickness= 2
        ),
        Line(
          origin= {-100,100},
          pattern= LinePattern.Solid,
          rotation= 0,
          points= {{151.33, -62.89}, {110.86, -50.86}},
          thickness= 2
        ),
        Line(
          origin= {-100,100},
          pattern= LinePattern.Solid,
          rotation= 0,
          points= {{110.86, -132.65}, {151.33, -145.21}},
          thickness= 2
        )
      }
    )
  );
end Gate;
```

Supplementary Listing 25: `SHMConduction/Icons/Heart.mo`

```
within SHMConduction.Icons;
model Heart "heart icon for full model of cardiac conduction system"
  annotation(
    Icon(
      coordinateSystem(
        preserveAspectRatio= false,
        extent= {{-100,-100},{100,100}}
      ),
      graphics= {
        Polygon(
          origin= {-100,100},
          lineThickness= 0.53,
          pattern= LinePattern.Solid,
          points= {{63.36, -149.76}, {77.13, -163.94}, {94.56, -172.20}, {114.76, -177.40}, {132.19,
                -179.84}, {143.62, -175.56}, {149.94, -164.14}, {152.69, -148.64}, {150.65,
```

```
      -132.02}, {145.04, -114.89}, {138.62, -102.65}, {130.15, -100.71}, {119.96, -96.02},
      {106.90, -89.39}, {90.08, -90.72}, {77.53, -96.02}, {69.07, -106.52}, {60.71,
      -121.62}, {56.94, -135.18}, {63.36, -149.76}},
  fillPattern= FillPattern.Solid,
  fillColor= {192,192,192},
  lineColor= {0,0,0},
  rotation= 0,
  smooth=Smooth.Bezier
),
Polygon(
  origin= {-100,100},
  lineThickness= 0.53,
  pattern= LinePattern.Solid,
  points= {{55.91, -142.11}, {52.04, -135.48}, {48.37, -122.43}, {46.94, -107.03}, {48.37,
      -93.26}, {51.53, -83.37}, {56.63, -78.07}, {63.76, -77.35}, {68.35, -78.48}, {68.46,
      -82.56}, {69.27, -86.63}, {77.33, -87.76}, {84.67, -89.49}, {83.04, -93.67}, {76.21,
      -102.75}, {67.64, -115.70}, {61.42, -127.32}, {58.87, -135.38}, {58.05, -140.89},
      {55.91, -142.11}},
  fillPattern= FillPattern.Solid,
  fillColor= {219,219,219},
  lineColor= {0,0,0},
  rotation= 0,
  smooth=Smooth.Bezier
),
Polygon(
  origin= {-100,100},
  lineThickness= 0.53,
  pattern= LinePattern.Solid,
  points= {{117.61, -84.91}, {115.67, -87.76}, {114.04, -92.09}, {115.85, -94.90}, {120.77,
      -97.35}, {126.28, -101.94}, {133.42, -101.83}, {138.25, -102.31}, {138.35, -97.63},
      {135.76, -94.19}, {133.58, -87.53}, {129.38, -82.02}, {122.20, -82.25}, {117.61,
      -84.91}},
  fillPattern= FillPattern.Solid,
  fillColor= {191,191,191},
  lineColor= {0,0,0},
  rotation= 0,
  smooth=Smooth.Bezier
),
Polygon(
  origin= {-100,100},
  lineThickness= 0.53,
  pattern= LinePattern.Solid,
  points= {{106.70, -59.41}, {111.59, -64.72}, {116.08, -65.23}, {128.42, -63.90}, {144.23,
      -65.53}, {155.81, -67.21}, {158.98, -68.69}, {160.48, -72.87}, {160.38, -79.81},
      {158.10, -84.09}, {149.94, -84.09}, {135.56, -81.34}, {123.32, -81.44}, {116.80,
      -86.33}, {114.97, -93.31}, {110.58, -91.53}, {102.42, -90.72}, {93.34, -91.64},
      {92.01, -89.19}, {94.77, -82.15}, {96.50, -75.93}, {95.48, -68.49}, {98.44, -59.82},
      {106.70, -59.41}},
  fillPattern= FillPattern.Solid,
  fillColor= {231,231,231},
  lineColor= {0,0,0},
  rotation= 0,
  smooth=Smooth.Bezier
),
Polygon(
  origin= {-100,100},
  lineThickness= 0.53,
  pattern= LinePattern.Solid,
  points= {{92.72, -84.60}, {95.48, -71.86}, {98.33, -62.68}, {103.33, -58.80}, {108.84,
      -58.50}, {112.30, -61.45}, {113.53, -65.43}, {119.54, -65.12}, {131.27, -64.51},
      {134.77, -64.90}, {137.26, -65.02}, {136.54, -61.49}, {135.86, -56.66}, {133.01,
      -46.46}, {129.33, -39.32}, {129.23, -35.86}, {134.23, -31.17}, {140.35, -26.17},
      {141.88, -22.80}, {139.74, -20.36}, {137.59, -19.34}, {131.99, -23.01}, {125.36,
      -29.43}, {125.66, -29.03}, {129.13, -24.13}, {131.37, -20.87}, {132.09, -18.42},
      {130.56, -16.18}, {128.01, -14.75}, {125.66, -15.16}, {121.07, -19.85}, {114.96,
      -25.56}, {109.65, -27.60}, {104.45, -26.99}, {100.68, -25.15}, {98.44, -21.58},
      {96.60, -15.87}, {94.87, -10.67}, {93.95, -8.12}, {93.03, -7.92}, {90.58, -8.99},
      {87.12, -10.72}, {85.89, -14.54}, {86.91, -20.15}, {87.47, -24.84}, {87.57, -28.92},
      {84.26, -36.78}, {78.14, -48.50}, {73.55, -59.82}, {70.39, -72.06}, {68.35, -81.85},
```

```
      {67.43, -86.34}, {75.08, -88.17}, {83.55, -89.29}, {82.57, -93.21}, {85.06, -94.36},
      {89.26, -92.25}, {92.72, -84.60}},
  fillPattern= FillPattern.Solid,
  fillColor= {176,176,176},
  lineColor= {0,0,0},
  rotation= 0,
  smooth=Smooth.Bezier
),
Polygon(
  origin= {-100,100},
  lineThickness= 0.53,
  pattern= LinePattern.Solid,
  points= {{135.53, -93.01}, {139.82, -92.83}, {144.10, -92.60}, {144.39, -91.39}, {144.39,
      -88.38}, {144.10, -85.78}, {143.63, -84.79}, {142.36, -84.33}, {138.83, -84.33},
      {136.57, -84.04}, {137.21, -83.58}, {140.45, -83}, {140.10, -82.01}, {133.74,
      -81.15}, {125.68, -81.06}, {121.31, -81.82}, {124.45, -82.51}, {128.65, -83.26},
      {130.96, -84.79}, {132.58, -87.34}, {133.62, -89.83}, {134.55, -91.97}, {135.53,
      -93.01}},
  fillPattern= FillPattern.Solid,
  fillColor= {161,161,161},
  lineColor= {0,0,0},
  rotation= 0,
  smooth=Smooth.Bezier
),
Polygon(
  origin= {-100,100},
  lineThickness= 0.53,
  pattern= LinePattern.Solid,
  points= {{55.58, -75.66}, {56.49, -72.12}, {57.65, -65.18}, {59.04, -53.43}, {60.14,
      -43.07}, {60.83, -36.25}, {61.76, -29.94}, {62.86, -24.04}, {63.96, -17.85}, {64.71,
      -13.97}, {66.27, -12.87}, {68.99, -12.35}, {71.89, -12.64}, {74.32, -13.62}, {75.30,
      -14.43}, {75.65, -19.76}, {76.05, -26.93}, {76.63, -29.48}, {79.23, -28.49}, {83.63,
      -25.54}, {87.10, -23.57}, {89.75, -22.53}, {91.93, -23.74}, {93.64, -26.82}, {94.22,
      -29.71}, {93.53, -32.02}, {87.16, -36.65}, {80.22, -42.38}, {78.66, -46.20}, {76.40,
      -53.26}, {72.93, -63.27}, {70.56, -72.47}, {69.28, -78.60}, {62.11, -79.41}, {55.20,
      -79.63}},
  fillPattern= FillPattern.Solid,
  fillColor= {225,225,225},
  lineColor= {0,0,0},
  rotation= 0,
  smooth=Smooth.Bezier
),
Ellipse(
  origin= {-100,100},
  lineThickness= 0.53,
  extent= {{45.36,-82.07},{55.94,-92.65}},
  pattern= LinePattern.Solid,
  fillPattern= FillPattern.Solid,
  fillColor= {17,89,255},
  lineColor= {17,89,255},
  rotation= 0
),
Ellipse(
  origin= {-100,100},
  lineThickness= 0.53,
  extent= {{69.17,-98.32},{79.75,-108.91}},
  pattern= LinePattern.Solid,
  fillPattern= FillPattern.Solid,
  fillColor= {17,89,255},
  lineColor= {17,89,255},
  rotation= 0
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{50.65, -87.74}, {53.67, -99.08}, {59.72, -105.88}, {68.04, -107.02}, {74.84,
      -105.13}},
```

```
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{52.16, -84.72}, {63.88, -85.85}, {73.33, -91.14}, {74.84, -100.59}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{53.29, -88.87}, {65.39, -94.54}, {74.08, -102.48}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{74.84, -103.99}, {77.49, -123.65}, {68.41, -140.28}, {68.04, -146.33}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{76.35, -102.10}, {95.25, -117.22}, {110.37, -117.98}, {112.75, -115.78},
      {120.20, -108.91}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{74.84, -104.37}, {86.93, -124.78}, {90.71, -162.96}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{76.35, -101.73}, {97.14, -109.29}, {102.43, -105.13}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{75.60, -103.99}, {85.23, -113.44}, {94.87, -122.89}, {104.51, -132.34}, {112.64,
       -141.04}, {116.04, -143.21}, {117.55, -140.47}, {120.57, -135.08}, {130.02,
      -128.94}},
  rotation= 0,
  smooth=Smooth.Bezier
),
```

```
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{104.51, -132.34}, {108.57, -136.69}, {111.88, -154.26}, {109.61, -165.23}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{105.44, -117.73}, {110.37, -117.98}, {116.41, -119.12}, {121.71, -119.87}, {127,
      -120.62}, {133.05, -116.85}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{88.56, -141.19}, {88.82, -143.87}, {81.26, -154.64}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{117.55, -140.47}, {119.06, -137.78}, {136.45, -145.19}, {147.03, -141.79}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{112.64, -141.04}, {116.04, -143.21}, {122.28, -150.67}, {126.34, -155.30},
      {130.40, -159.93}, {128.13, -173.92}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Line(
  origin= {-100,100},
  color= {17,89,255},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{124.31, -152.99}, {126.34, -155.30}, {137.21, -156.15}, {143.63, -152.75}},
  rotation= 0,
  smooth=Smooth.Bezier
),
Ellipse(
  origin= {-100,100},
  lineThickness= 0.53,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {107,107,107},
  extent= {{152.88,-69.07},{159.03,-82.90}},
  rotation= 0
),
Ellipse(
  origin= {-100,100},
  lineThickness= 0.53,
  pattern= LinePattern.None,
```

```
          fillPattern= FillPattern.Solid,
          fillColor= {107,107,107},
          extent= {{72.89,-54.28},{76.36,-62.10}},
          rotation= 20.90
        )
      }
    )
  );
end Heart;
```

Supplementary Listing 26: `SHMConduction/Icons/Hourglass.mo`

```
within SHMConduction.Icons;
model Hourglass "hourglass icon for delay"
  annotation(
    Icon(
      coordinateSystem(
        preserveAspectRatio= false,
        extent= {{-100,-100},{100,100}}
      ),
      graphics= {
        Polygon(
          origin= {-100,100},
          lineThickness= 2,
          pattern= LinePattern.Solid,
          points= {{60.25, -28.44}, {139.75, -28.44}, {60.25, -171.56}, {139.75, -171.56}},
          fillPattern= FillPattern.None,
          lineColor= {0,0,0},
          rotation= 0
        ),
        Polygon(
          origin= {-100,100},
          lineThickness= 1,
          pattern= LinePattern.None,
          points= {{78.80, -61.84}, {121.20, -61.84}, {100, -100}},
          fillPattern= FillPattern.Solid,
          fillColor= {0,0,0},
          rotation= 0
        ),
        Polygon(
          origin= {-100,100},
          lineThickness= 1,
          pattern= LinePattern.None,
          points= {{60.25, -171.56}, {100, -155.65}, {139.75, -171.56}},
          fillPattern= FillPattern.Solid,
          fillColor= {0,0,0},
          rotation= 0
        ),
        Ellipse(
          origin= {-100,100},
          lineThickness= 1,
          pattern= LinePattern.None,
          fillPattern= FillPattern.Solid,
          fillColor= {0,0,0},
          extent= {{98.41,-112.72},{101.59,-115.90}},
          rotation= 0
        ),
        Ellipse(
          origin= {-100,100},
          lineThickness= 1,
          pattern= LinePattern.None,
          fillPattern= FillPattern.Solid,
          fillColor= {0,0,0},
          extent= {{98.41,-123.85},{101.59,-127.03}},
          rotation= 0
        ),
        Ellipse(
          origin= {-100,100},
```

```
  lineThickness= 1,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {0,0,0},
  extent= {{98.41,-134.98},{101.59,-138.16}},
  rotation= 0
),
Ellipse(
  origin= {-100,100},
  lineThickness= 1,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {0,0,0},
  extent= {{98.41,-146.11},{101.59,-149.29}},
  rotation= 0
),
Ellipse(
  origin= {-100,100},
  lineThickness= 2,
  extent= {{2.05,-2.05},{197.95,-197.95}},
  pattern= LinePattern.Solid,
  fillPattern= FillPattern.None,
  lineColor= {0,0,0},
  rotation= 0
),
Ellipse(
  origin= {-107,112.51},
  lineThickness= 2,
  extent= {{130.21,-88.38},{178.47,-136.64}},
  pattern= LinePattern.Solid,
  fillPattern= FillPattern.None,
  lineColor= {0,0,0},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 2,
  points= {{153.25, -104.27}, {153.25, -116.19}, {168.83, -116.19}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{154.34, -130.60}, {154.34, -136.64}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{154.34, -88.38}, {154.34, -94.42}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{130.21, -112.51}, {136.25, -112.51}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
```

```
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{172.42, -112.51}, {178.47, -112.51}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{166.40, -91.61}, {164.39, -95.09}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{175.23, -100.44}, {171.76, -102.45}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{175.23, -124.57}, {171.76, -122.57}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{166.40, -133.41}, {164.39, -129.93}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{142.27, -133.41}, {144.28, -129.93}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{133.44, -124.57}, {136.92, -122.57}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{133.44, -100.44}, {136.92, -102.45}},
  rotation= -0
),
Line(
  origin= {-107,112.51},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 1,
  points= {{142.27, -91.61}, {144.28, -95.09}},
  rotation= -0
),
```

```
        Ellipse(
          origin= {-107,112.51},
          pattern= LinePattern.None,
          fillPattern= FillPattern.Solid,
          fillColor= {0,0,0},
          extent= {{153.49,-87.53},{155.18,-89.23}},
          rotation= -0
        ),
        Ellipse(
          origin= {-107,112.51},
          pattern= LinePattern.None,
          fillPattern= FillPattern.Solid,
          fillColor= {0,0,0},
          extent= {{153.49,-87.53},{155.18,-89.23}},
          rotation= -0
        ),
        Line(
          origin= {-107,112.51},
          color= {0,0,0},
          pattern= LinePattern.Solid,
          thickness= 2,
          points= {{154.34, -88.38}, {154.34, -82.29}},
          rotation= -0
        ),
        Rectangle(
          origin= {-107,112.51},
          lineThickness= 2.12,
          pattern= LinePattern.None,
          fillPattern= FillPattern.Solid,
          fillColor= {0,0,0},
          extent= {{148.50,-80.83},{160.18,-83.75}},
          rotation= -0
        )
      }
    )
  );
end Hourglass;
```

Supplementary Listing 27: `SHMConduction/Icons/Metronome.mo`

```
within SHMConduction.Icons;
model Metronome "metronome icon for pacemaker"
  annotation(
    Icon(
      coordinateSystem(
        preserveAspectRatio= false,
        extent= {{-100,-100},{100,100}}
      ),
      graphics= {
        Line(
          origin= {-100,100},
          pattern= LinePattern.Solid,
          rotation= 0,
          points= {{98.41, -140.61}, {160.53, -33.61}},
          thickness= 2
        ),
        Polygon(
          origin= {-100,100},
          lineThickness= 2,
          pattern= LinePattern.Solid,
          points= {{82.31, -34.98}, {115.02, -34.98}, {147.73, -152.74}, {49.60, -152.74}},
          fillPattern= FillPattern.None,
          rotation= 0
        ),
        Ellipse(
          origin= {-100,100},
          lineThickness= 2,
          pattern= LinePattern.Solid,
```

```
          endAngle= 179.999997341,
          fillPattern= FillPattern.Solid,
          fillColor= {255,255,255},
          extent= {{74.88,-117.33},{122.44,-164.88}},
          startAngle= 0.0,
          rotation= 0
        ),
        Polygon(
          origin= {-100,100},
          lineThickness= 2,
          pattern= LinePattern.None,
          points= {{143.95, -55.42}, {149.79, -58.93}, {159.31, -49.78}, {147.56, -42.71}},
          fillPattern= FillPattern.Solid,
          fillColor= {0,0,0},
          rotation= 0
        )
      }
    )
  );
end Metronome;
```

A.2. An understandable, extensible and reusable implementation of the Hodgkin-Huxley equations using Modelica

frontiers

Data Supplement

An understandable, extensible and reusable implementation of the Hodgkin-Huxley equations using Modelica

Christopher Schölzel, Valeria Blesius, Gernot Ernst, and Andreas Dominik
Thursday 17th September, 2020

RATIONALE FOR PUBLISHING CODE AS PDF

This data supplement contains the full model code. This code is also available on GitHub[1], which is the preferred source since it may contain fixes and updates applied after the publication of this paper. However, GitHub may not always be there in the future and therefore we provide this version to be archived along with the paper.

PACKAGE STRUCTURE

Modelica organizes code in packages. These packages can be defined within a single file or, in a folder structure where each folder contains a file called `package.mo` that contains the package metadata. This project uses a mix of both styles: Since the components themselves are very small, they are grouped in the single file `HHModelica/Components/package.mo`. Full models and Icons, however, are created in their own separate files.

PACKAGE METADATA

Listing 1: `HHmodelica/package.mo`

```
package HHmodelica "Modelica implementations of the Hogdkin-Huxley model"
end HHmodelica;
```

Listing 2: `HHmodelica/CompleteModels/package.mo`

```
within HHmodelica;
package CompleteModels
end CompleteModels;
```

Listing 3: `HHmodelica/Icons/package.mo`

```
within HHmodelica;
package Icons "contains icon classes that are used to inherit annotations"
end Icons;
```

COMPONENTS

Listing 4: `HHmodelica/Components/package.mo`

```
within HHmodelica;
package Components "components for the two-pin modular version of the
   Hodgkin-Huxley model"
  connector TemperatureInput = input Real(unit="degC") "membrane
     temperature"
    annotation(
      Icon(
        coordinateSystem(
          preserveAspectRatio=true,
          extent={{-100,-100},{100,100}}
        ),
        graphics={
          Ellipse(
            extent={{-100,100},{100,-100}},
```

[1] https://github.com/CSchoel/hh-modelica

```
            lineColor={255,0,0},
            fillColor={255,255,255},
            fillPattern=FillPattern.Solid
          )
        }
      )
    );
  connector TemperatureOutput = output Real(unit="degC") "membrane
     temperature"
    annotation(
      Icon(
        coordinateSystem(
          preserveAspectRatio=true,
          extent={{-100,-100},{100,100}}
        ),
        graphics={
          Ellipse(
            extent={{-100,100},{100,-100}},
            lineColor={255,0,0},
            fillColor={255,0,0},
            fillPattern=FillPattern.Solid
          )
        }
      )
    );

  connector ElectricalPin "electrical connector for membrane currents"
    flow Real i(unit="uA/cm2") "ionic current through membrane";
    Real v(unit="mV") "membrane potential (as displacement from resting
       potential)";
  end ElectricalPin;

  connector PositivePin "electrical pin with filled square icon for visual
      distinction"
    extends ElectricalPin;
    annotation(
      Icon(
        coordinateSystem(
          preserveAspectRatio=true,
          extent={{-100,-100},{100,100}}
        ),
        graphics={
          Rectangle(
            extent={{-100,100},{100,-100}},
            lineColor={0,0,255},
            fillColor={0,0,255},
            fillPattern=FillPattern.Solid
          )
        }
      )
    );
  end PositivePin;

  connector NegativePin "electrical pin with open square icon for visual
     distinction"
    extends ElectricalPin;
    annotation(
      Icon(
        coordinateSystem(
          preserveAspectRatio=true,
          extent={{-100,-100},{100,100}}
        ),
```

```
        graphics={
          Rectangle(
            extent={{-100,100},{100,-100}},
            lineColor={0,0,255},
            fillColor={255,255,255},
            fillPattern=FillPattern.Solid
          )
        }
      )
    );
  end NegativePin;

  partial model TwoPinComponent "component with two directly connected
     electrical pins"
    PositivePin p "positive extracellular pin" annotation (Placement(
       transformation(extent={{-10, 90},{10, 110}})));
    NegativePin n "negative intracellular pin" annotation (Placement(
       transformation(extent={{-10, -90},{10, -110}})));
    Real v(unit="mV") "voltage as potential difference between positive
       and negative pin";
    Real i(unit="uA/cm2") "outward current flowing through component from
       negative to positive pin";
  equation
    i = p.i;
    0 = p.i + n.i;
    v = p.v - n.v;
    annotation(
      Documentation(info="
        <html>
          <p>This is a base model for components with two electrical pins
             which
          are directly connected. It establishes the connection and
             defines a
          voltage between the positive and negative pin, but does not
             specify
          the current-voltage relationship.</p>
          <p>This base component establishes the convention that positive
          currents are outward currents and negative currents are inward
          currents.</p>
        </html>
      ")
    );
  end TwoPinComponent;

  function expFit "exponential function with scaling parameters for x and
     y axis"
    input Real x "input value";
    input Real sx "scaling factor for x axis (fitting parameter)";
    input Real sy "scaling factor for y axis (fitting parameter)";
    output Real y "result";
  algorithm
    y := sy * exp(sx * x);
  end expFit;

  function goldmanFit "fitting function related to Goldmans formula for
     the movement of a charged particle in a constant electrical field"
    input Real x "membrane potential (as displacement from resting
       potential)";
    input Real x0 "offset for x (fitting parameter)";
    input Real sx "scaling factor for x (fitting parameter)";
    input Real sy "scaling factor for y (fitting parameter)";
    output Real y "rate of change of the gating variable at given V=x";
```

```
  protected
    Real x_adj "adjusted x with offset and scaling factor";
  algorithm
    x_adj := sx * (x - x0);
    if abs(x - x0) < 1e-6 then
      y := sy; // using L'Hôpital to find limit for x_adj->0
    else
      y := sy * x_adj / (exp(x_adj) - 1);
    end if;
    annotation(
      Documentation(info="
        <html>
          <p>Hodgkin and Huxley state that this formula was (in part) used
          because it &quot;bears a close resemblance to the equation
             derived
          by Goldman (1943) for the movements of a charged particle in a
             constant
          field&quot;.</p>
          <p>We suppose that this statement refers to equation 11 of
             Goldman
          (1943), which is also called the Goldman-Hodgkin-Katz flux
             equation:</p>
          <blockquote>
            j_i = u_i * F / a * dV * (n'_i * exp(-z_i * beta * dV - n0_i))
                / exp(-z_i * beta * dV)
          </blockquote>
          <p>Factoring out n0_i from the denominator, substituting
          n'_i/n0_i = exp(V_0 * beta * z_i) and grouping and renaming
          variables, the GHK flux equation can be written as</p>
          <blockquote>
            y := sy * sx * x * (exp((x - x0) * sx) - 1) / (exp(x * sx) -
               1)
          </blockquote>
          <p>with sx = -z_i * beta, x = dV, x0 = V_0, and
          sy = n0_i * u_i * F / a * 1 / sx.</p>
          <p>With this notation, the similarity becomes apparent, as
             omitting
          the denominator (exp((x - x0) * sx) - 1) and using x-x0 instead
             of
          x in the rest of the formula gives exactly the goldmanFit used
             by
          Hodgkin and Huxley.</p>
        </html>
      ")
    );
  end goldmanFit;

  function logisticFit "logistic function with sigmoidal shape"
    input Real x "input value";
    input Real x0 "x-value of sigmoid midpoint (fitting parameter)";
    input Real sx "growth rate/steepness (fitting parameter)";
    input Real y_max "maximum value";
    output Real y "result";
  protected
    Real x_adj "adjusted x with offset and scaling factor";
  algorithm
    x_adj := sx * (x - x0);
    y := y_max / (exp(-x_adj) + 1);
  end logisticFit;

  model Gate "gating molecule with an open conformation and a closed
     conformation"
```

```
  replaceable function fopen = expFit(sx=1, sy=1) "rate of transfer from
      closed to open conformation";
  replaceable function fclose = expFit(sx=1, sy=1) "rate of transfer
     from open to closed conformation";
  Real n(start=fopen(0)/(fopen(0) + fclose(0)), fixed=true) "ratio of
     molecules in open conformation";
  input Real v(unit="mV") "membrane potential (as displacement from
     resting potential)";
  TemperatureInput temp "membrane temperature";
protected
  Real phi = 3^((temp-6.3)/10) "temperature-dependent factor for rate of
      transfer calculated with Q10 = 3";
equation
  der(n) = phi * (fopen(v) * (1 - n) - fclose(v) * n);
end Gate;

partial model IonChannel "ionic current through the membrane"
  extends TwoPinComponent;
  extends HHmodelica.Icons.IonChannel;
  Real g(unit="mmho/cm2") "ion conductance, needs to be defined in
     subclasses";
  parameter Real v_eq(unit="mV") "equilibrium potential (as displacement
      from resting potential)";
  parameter Real g_max(unit="mmho/cm2") "maximum conductance";
equation
  i = g * (v - v_eq);
end IonChannel;

partial model GatedIonChannel "ion channel that has voltage-dependent
    gates"
  extends IonChannel;
  TemperatureInput temp "membrane temperature to determine reaction
     coefficient"
    annotation (Placement(transformation(extent={{-40, 48},{-60, 68}})))
       ;
end GatedIonChannel;

model PotassiumChannel "channel selective for K+ cations"
  extends GatedIonChannel(g_max=36, v_eq=12);
  extends HHmodelica.Icons.Activatable;
  Gate gate_act(
    redeclare function fopen= goldmanFit(x0=-10, sy=100, sx=0.1),
    redeclare function fclose= expFit(sx=1/80, sy=125),
    v=v, temp=temp
  ) "activation gate";
equation
  g = g_max * gate_act.n ^ 4;
end PotassiumChannel;

model SodiumChannel "channel selective for Na+ cations"
  extends GatedIonChannel(g_max=120, v_eq=-115);
  extends HHmodelica.Icons.Activatable;
  extends HHmodelica.Icons.Inactivatable;
  Gate gate_act(
    redeclare function fopen= goldmanFit(x0=-25, sy=1000, sx=0.1),
    redeclare function fclose= expFit(sx=1/18, sy=4000),
    v=v, temp=temp
  ) "activation gate";
  Gate gate_inact(
    redeclare function fopen= expFit(sx=1/20, sy=70),
    redeclare function fclose= logisticFit(x0=-30, sx=-0.1, y_max=1000),
    v=v, temp=temp
```

```
  ) "inactivation gate";
equation
  g = g_max * gate_act.n ^ 3 * gate_inact.n;
end SodiumChannel;

model LeakChannel "constant leakage current of ions through membrane"
  extends IonChannel(g_max=0.3, v_eq=-10.613);
  extends HHmodelica.Icons.OpenChannel;
equation
  g = g_max;
end LeakChannel;

model LipidBilayer "lipid bilayer separating external and internal
   potential (i.e. acting as a capacitor)"
  extends TwoPinComponent;
  extends HHmodelica.Icons.LipidBilayer;
  TemperatureOutput temp = temp_m annotation (Placement(transformation(
     extent={{40, 48},{60, 68}})));
  parameter Real temp_m(unit="degC") = 6.3 "constant membrane
     temperature";
  parameter Real c(unit="uF/cm2") = 1 "membrane capacitance";
  parameter Real v_init(unit="mV") = -90 "short initial stimulation";
initial equation
  v = v_init;
equation
  der(v) = 1000 * i / c "multiply with 1000 to get mV/s instead of v/s";
end LipidBilayer;

model ConstantCurrent "applies current to positive pin regardless of
   voltage"
  extends TwoPinComponent;
  parameter Real i_const(unit="uA/cm2") "current applied to positive pin
     ";
equation
  i = i_const;
end ConstantCurrent;

model Ground "sets voltage to zero, acting as a reference for measuring
   potential"
  PositivePin p;
equation
  p.v = 0;
end Ground;

model Membrane "full membrane model that can be used in current clamp
   experiments"
  extends HHmodelica.Icons.LipidBilayer;
  PositivePin p "positive extracellular pin" annotation (Placement(
     transformation(extent={{-10, 90},{10, 110}})));
  NegativePin n "negative intracellular pin" annotation (Placement(
     transformation(extent={{-10, -90},{10, -110}})));
  PotassiumChannel c_pot;
  SodiumChannel c_sod;
  LeakChannel c_leak;
  LipidBilayer l2 "lipid bilayer as capacitor";
equation
  connect(c_pot.p, l2.p);
  connect(c_pot.n, l2.n);
  connect(c_sod.p, l2.p);
  connect(c_sod.n, l2.n);
  connect(c_leak.p, l2.p);
  connect(c_leak.n, l2.n);
```

```
    connect(p, l2.p);
    connect(n, l2.n);
    connect(c_pot.temp, l2.temp);
    connect(c_sod.temp, l2.temp);
  end Membrane;

  model CurrentClamp "current clamp that applies constant current to the
     membrane"
    extends HHmodelica.Icons.CurrentClamp;
    PositivePin p "extracellular electrode" annotation (Placement(
       transformation(extent={{-10, 90},{10, 110}})));
    NegativePin n "intracellular electrode(s)" annotation (Placement(
       transformation(extent={{-10, -90},{10, -110}})));
    parameter Real i_const(unit="uA/cm2") = 40 "current applied to
       membrane";
    ConstantCurrent cur(i_const=i_const) "external current applied to
       membrane";
    Ground g "reference electrode";
    Real v(unit="mV") = -n.v "measured membrane potential";
  equation
    connect(p, cur.p);
    connect(n, cur.n);
    connect(g.p, p);
  end CurrentClamp;

end Components;
```

COMPLETE MODELS

Listing 5: `HHmodelica/CompleteModels/HHmono.mo`

```
within HHmodelica.CompleteModels;
partial model PotentialAdapter "base class that converts membrane
   potential to current standards"
  parameter Real e_r(unit="mV") = -75 "resting potential";
  Real v_m(unit="mV") = e_r - v "absolute membrane potential (v_in - v_out
     )";
  Real v(unit="mV") "membrane potential as displacement from resting
     potential (out - in)";
  annotation(
    experiment(StartTime = 0, StopTime = 30, Tolerance = 1e-6, Interval =
       0.01),
    __OpenModelica_simulationFlags(s = "dassl"),
    __MoST_experiment(variableFilter="v_m|v|gK|gNa|n|m|h"),
    Documentation(info="
      <html>
        <p>The variable e_r in this adapter can be used to plot the
           absolute
        membrane potential as difference between the potential on the
           inside
        and the potential on the outside of the cell.
        This conforms with current standards, but not to the original
           equations
        by Hodgkin and Huxley, which define V as the displacement from the
        resting potential with opposite sign.</p>
        <p>For this conversion, a value for the resting potential e_r must
            be
        assumed, which is not given in the original article. We use e_r =
           -75 mV,
        because this is the value that is used by the BioModels
           implementation of
        the Hodgkin-Huxley model and corresponds to the resting potential
```

```
        measured for the squid giant axon <i>in vivo</i>
        (cf. Moore and Cole, 1960, https://doi.org/10.1085/jgp.43.5.961)
           .</p>
      </html>
    ")
  );
end PotentialAdapter;
```

Listing 6: `HHmodelica/CompleteModels/HHmono.mo`

```
within HHmodelica.CompleteModels;
model HHmono "monolithic version of the Hodgkin-Huxley model"
  extends PotentialAdapter;
  parameter Real e_r(unit="mV") = -75 "resting potential";
  Real v_m(unit="mV") = e_r - v "absolute membrane potential (v_in - v_out
    )";
  parameter Real Cm(unit = "uF/cm2")       = 1;
  parameter Real gbarNa(unit = "mmho/cm2")   = 120 "max sodium conductance
    ";
  parameter Real gbarK(unit = "mmho/cm2")    = 36 "max potassium
    conductance";
  parameter Real gbar0(unit = "mmho/cm2")    = 0.3;
  parameter Real VNa(unit = "mV")      = -115;
  parameter Real VK(unit = "mV")       = 12;
  parameter Real Vl(unit = "mV")       = -10.613;
  parameter Real Temp     = 6.3 ;
  parameter Real phi      = 3^((Temp-6.3)/10);
  parameter Real Vdepolar(unit = "mV") = -90;
  parameter Real Vnorm(unit = "mV")    = 1 "for non-dimensionalizing v in
    function expressions, i.e. exp(v/Vnorm) replaces exp(v).";
  parameter Real msecm1(unit = "1/msec") = 1 "for adding units to alpha
    and beta variables";
  parameter Real alphan0 (unit="1/msec") = 0.1/(exp(1)-1) "always use v=0
    to calculate i.c.";
  parameter Real betan0  (unit="1/msec") = 0.125;
  parameter Real alpham0 (unit="1/msec") = 2.5/(exp(2.5)-1);
  parameter Real betam0  (unit="1/msec") = 4;
  parameter Real alphah0 (unit="1/msec") = 0.07;
  parameter Real betah0  (unit="1/msec") = 1/(exp(3)+1);

  parameter Real minusI(unit = "nA/cm2") = 40;
  input Real Vclamp(unit = "mV");

  parameter Real clamp_0no_1yes = 0;

  //Variables for all the algebraic equations
  Real INa(unit = "nA/cm2") "Ionic currents";
  Real IK(unit = "nA/cm2") "Ionic currents";
  Real Il(unit = "nA/cm2") "Ionic currents";
  Real alphan(unit = "1/msec")  "rate constant of particles from out to in
    ";
  Real betan(unit = "1/msec")   "rate constant from in to out";
  Real alpham(unit = "1/msec")  "rate constant of activating molecules
    from out to in";
  Real betam(unit = "1/msec")   "rate constant of activating molecules
    from in to out";
  Real alphah(unit = "1/msec")  "rate constant of inactivating molecules
    from out to in";
  Real betah(unit = "1/msec")   "rate constant of inactivating molecules
    from in to out";
  Real gNa(unit = "mmho/cm2")   "Sodium conductance";
  Real gK(unit = "mmho/cm2")    "potassium conductance";
```

```
  Real Iion(unit = "nA/cm2");

  //State variables for all the ODEs
  Real VV(unit="mV");
  Real v(unit="mV") "displacement of the membrane potential from its
     resting value (depolarization negative)";
  Real n "proportion of the particles in a certain position";
  Real m "proportion of activating molecules on the inside";
  Real h "proportion of inactivating molecules on the outside";

protected
  Real ninf;
  Real minf;
  Real hinf;
  Real taun(unit="msec");
  Real taum(unit="msec");
  Real tauh(unit="msec");

initial equation
  VV = if clamp_0no_1yes == 0 then Vdepolar else Vclamp;

  n = alphan0/(alphan0+betan0);
  m = alpham0/(alpham0+betam0);
  h = alphah0/(alphah0+betah0);

equation
  //if v/Vnorm == -10 then
  //  alphan = 0.1;
  //else
  alphan = 0.01 * (v / Vnorm + 10) / (exp((v / Vnorm + 10) / 10) - 1);
  //end if;
  betan = msecm1 * (0.125 * exp(v / Vnorm / 80));
  //if v/Vnorm == -25 then
  //  alpham = 1;
  //else
  alpham = 0.1 * (v / Vnorm + 25) / (exp((v / Vnorm + 25) / 10) - 1);
  //end if;
  betam = msecm1 * (4 * exp(v / Vnorm / 18));
  alphah  = msecm1*(0.07*exp((v/Vnorm)/20));
  betah   = msecm1*(1/(exp(( v/Vnorm+30)/10)+1));
  minf    = alpham/(alpham+betam);
  ninf    = alphan/(alphan+betan);
  hinf    = alphah/(alphah+betah);
  taun    =1/(alphan+betan);
  tauh    =1/(alphah+betah);
  taum    =1/(alpham+betam);
  gNa     = gbarNa * m^3 * h;
  gK      = gbarK  * n^4;
  INa     = gNa * (v-VNa);
  IK      = gK  * (v-VK);
  Il      = gbar0 * (v-Vl);
  Iion    = INa + IK + Il;
  if (clamp_0no_1yes == 0) then
    der(VV) = (-minusI-INa-IK-Il)/Cm;
    v = VV;
  else
    der(VV) = 0;
    v = Vclamp;
  end if;
  der(n) = phi*(alphan*(1-n)-betan*n);
  der(m) = phi*(alpham*(1-m)-betam*m);
  der(h) = phi*(alphah*(1-h)-betah*h);
```

```
  annotation(
    experiment(StartTime = 0, StopTime = 30, Tolerance = 1e-6, Interval =
       0.01),
    __OpenModelica_simulationFlags(s = "dassl"),
    __MoST_experiment(variableFilter="v_m|v|gK|gNa|n|m|h")
  );
  end HHmono;
```

Listing 7: `HHmodelica/CompleteModels/HHmodular.mo`

```
within HHmodelica.CompleteModels;
model HHmodular "'flat' version of the modular model (no membrane
   container)"
  extends PotentialAdapter(v = l2.v);
  HHmodelica.Components.PotassiumChannel c_pot annotation(
    Placement(visible = true, transformation(origin = {-33, 3}, extent =
       {{-17, -17}, {17, 17}}, rotation = 0)));
  HHmodelica.Components.SodiumChannel c_sod annotation(
    Placement(visible = true, transformation(origin = {1,3}, extent =
       {{-17, -17}, {17, 17}}, rotation = 0)));
  HHmodelica.Components.LeakChannel c_leak annotation(
    Placement(visible = true, transformation(origin = {35, 3}, extent =
       {{-17, -17}, {17, 17}}, rotation = 0)));
  HHmodelica.Components.LipidBilayer l2 annotation(
    Placement(visible = true, transformation(origin = {-67, 3}, extent =
       {{-17, -17}, {17, 17}}, rotation = 0)));
  HHmodelica.Components.CurrentClamp clamp annotation(
    Placement(visible = true, transformation(origin = {69, 3}, extent =
       {{-17, -17}, {17, 17}}, rotation = 0)));
equation
  connect(l2.p, c_pot.p) annotation(
    Line(points = {{-66, 20}, {-66, 40}, {-33, 40}, {-33, 20}}, color =
       {0, 0, 255}));
  connect(c_pot.p, c_sod.p) annotation(
    Line(points = {{-33, 20}, {-32, 20}, {-32, 40}, {0, 40}, {0, 20}, {2,
       20}}, color = {0, 0, 255}));
  connect(c_sod.p, c_leak.p) annotation(
    Line(points = {{2, 20}, {2, 20}, {2, 40}, {34, 40}, {34, 20}, {36,
       20}}, color = {0, 0, 255}));
  connect(c_leak.p, clamp.p) annotation(
    Line(points = {{36, 20}, {36, 20}, {36, 40}, {68, 40}, {68, 20}, {70,
       20}}, color = {0, 0, 255}));
  connect(clamp.n, c_leak.n) annotation(
    Line(points = {{70, -14}, {68, -14}, {68, -40}, {36, -40}, {36, -14},
       {36, -14}}, color = {0, 0, 255}));
  connect(c_leak.n, c_sod.n) annotation(
    Line(points = {{36, -14}, {34, -14}, {34, -40}, {2, -40}, {2, -14},
       {2, -14}}, color = {0, 0, 255}));
  connect(c_sod.n, c_pot.n) annotation(
    Line(points = {{2, -14}, {0, -14}, {0, -40}, {-32, -40}, {-32, -14},
       {-33, -14}}, color = {0, 0, 255}));
  connect(c_pot.n, l2.n) annotation(
    Line(points = {{-33, -14}, {-33, -40}, {-66, -40}, {-66, -14}}, color
       = {0, 0, 255}));
  connect(l2.temp, c_pot.temp) annotation(
    Line(points = {{-58, 12}, {-58, 16}, {-42, 16}, {-42, 13}}, color =
       {255, 0, 0}));
  connect(c_pot.temp, c_sod.temp) annotation(
    Line(points = {{-42, 13}, {-40, 13}, {-40, 16}, {-8, 16}, {-8, 12}},
       color = {255, 0, 0}));
annotation(
```

```
  experiment(StartTime = 0, StopTime = 0.03, Tolerance = 1e-6, Interval =
     1e-05),
  __OpenModelica_simulationFlags(s = "dassl"),
  __MoST_experiment(variableFilter="v_m|clamp\\.(v|i)|c_pot\\.(g|gate_act
     \\.n)|c_sod\\.(g|gate_act\\.n|gate_inact\\.n)")
);
end HHmodular;
```

Listing 8: `HHmodelica/CompleteModels/HHmodHier.mo`

```
within HHmodelica.CompleteModels;
model HHmodHier
  extends PotentialAdapter(v = m.l2.v);
  import HHmodelica.Components.Membrane;
  import HHmodelica.Components.CurrentClamp;
  Membrane m;
  // i = 40 => recurring depolarizations
  // i = 0 => v returns to 0
  CurrentClamp c(i_const=40);
equation
  connect(m.p, c.p);
  connect(m.n, c.n);
annotation(
  experiment(StartTime = 0, StopTime = 0.03, Tolerance = 1e-6, Interval =
     1e-05),
  __OpenModelica_simulationFlags(s = "dassl"),
  __MoST_experiment(variableFilter="c\\.(v|i)|m\\.c_pot\\.(g|gate_act\\.n)
     |m\\.c_sod\\.(g|gate_act\\.n|gate_inact\\.n)")
);
end HHmodHier;
```

ICONS

Listing 9: `HHmodelica/Icons/LipidBilayer.mo`

```
within HHmodelica.Icons;
model LipidBilayer "lipid bilayer with red circles on outside and black
   lines on inside"
annotation(
  Icon(
    coordinateSystem(
      preserveAspectRatio= false,
      extent= {{-100,-100},{100,100}}
    ),
    graphics= {
      Rectangle(
        origin= {-100,2145.04},
        lineThickness= 1,
        pattern= LinePattern.None,
        fillPattern= FillPattern.Solid,
        fillColor= {230,230,230},
        extent= {{0.17,-2098.02},{200,-2245.04}},
        rotation= -0
      ),
      Line(
        origin= {-100,2145.04},
        color= {0,0,0},
        pattern= LinePattern.Solid,
        thickness= 0.25,
        points= {{193.86, -2181.04}, {198.41, -2166.55}, {196.24,
           -2148.38}},
        rotation= -0
```

```
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{189.41, -2181.04}, {184.86, -2166.55}, {187.89,
       -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{185.50,-2179.95},{198.35,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{193.86, -2109.77}, {198.41, -2124.27}, {196.24,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{189.41, -2109.77}, {184.86, -2124.27}, {187.89,
       -2142.87}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{10.57, -2181.04}, {15.12, -2166.55}, {12.95, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{6.12, -2181.04}, {1.58, -2166.55}, {4.60, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{2.21,-2179.95},{15.06,-2192.80}},
    rotation= -0
  ),
```

```
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{10.57, -2109.77}, {15.12, -2124.27}, {12.95, -2142.44}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{6.12, -2109.77}, {1.58, -2124.27}, {4.60, -2142.87}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{100, -2045.04}, {100, -2098.03}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{25.85, -2181.04}, {30.39, -2166.55}, {28.23, -2148.38}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{21.39, -2181.04}, {16.85, -2166.55}, {19.88, -2147.95}},
  rotation= -0
),
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {219,0,0},
  extent= {{17.48,-2179.95},{30.33,-2192.80}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{25.85, -2109.77}, {30.39, -2124.27}, {28.23, -2142.44}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
```

```
    points= {{21.39, -2109.77}, {16.85, -2124.27}, {19.88, -2142.87}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{41.12, -2181.04}, {45.67, -2166.55}, {43.50, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{36.67, -2181.04}, {32.12, -2166.55}, {35.15, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{32.76,-2179.95},{45.60,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{41.12, -2109.77}, {45.67, -2124.27}, {43.50, -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{36.67, -2109.77}, {32.12, -2124.27}, {35.15, -2142.87}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{56.40, -2181.04}, {60.94, -2166.55}, {58.78, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{51.94, -2181.04}, {47.40, -2166.55}, {50.43, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
```

```
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{48.03,-2179.95},{60.88,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{56.40, -2109.77}, {60.94, -2124.27}, {58.78, -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{51.94, -2109.77}, {47.40, -2124.27}, {50.43, -2142.87}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{71.67, -2181.04}, {76.21, -2166.55}, {74.05, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{67.21, -2181.04}, {62.67, -2166.55}, {65.70, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{63.30,-2179.95},{76.15,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{71.67, -2109.77}, {76.21, -2124.27}, {74.05, -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{67.21, -2109.77}, {62.67, -2124.27}, {65.70, -2142.87}},
```

```
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{86.95, -2181.04}, {91.49, -2166.55}, {89.32, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{82.49, -2181.04}, {77.95, -2166.55}, {80.97, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{78.58,-2179.95},{91.43,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{86.95, -2109.77}, {91.49, -2124.27}, {89.32, -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{82.49, -2109.77}, {77.95, -2124.27}, {80.97, -2142.87}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{102.22, -2181.04}, {106.76, -2166.55}, {104.60,
       -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{97.76, -2181.04}, {93.22, -2166.55}, {96.25, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
```

```
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{93.85,-2179.95},{106.70,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{102.22, -2109.77}, {106.76, -2124.27}, {104.60,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{97.76, -2109.77}, {93.22, -2124.27}, {96.25, -2142.87}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{117.49, -2181.04}, {122.04, -2166.55}, {119.87,
       -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{113.04, -2181.04}, {108.49, -2166.55}, {111.52,
       -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{109.13,-2179.95},{121.98,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{117.49, -2109.77}, {122.04, -2124.27}, {119.87,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
```

```
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{113.04, -2109.77}, {108.49, -2124.27}, {111.52,
       -2142.87}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{132.77, -2181.04}, {137.31, -2166.55}, {135.15,
       -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{128.31, -2181.04}, {123.77, -2166.55}, {126.80,
       -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{124.40,-2179.95},{137.25,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{132.77, -2109.77}, {137.31, -2124.27}, {135.15,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{128.31, -2109.77}, {123.77, -2124.27}, {126.80,
       -2142.87}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{148.04, -2181.04}, {152.58, -2166.55}, {150.42,
       -2148.38}},
    rotation= -0
  ),
  Line(
```

```
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{143.59, -2181.04}, {139.04, -2166.55}, {142.07,
       -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{139.68,-2179.95},{152.52,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{148.04, -2109.77}, {152.58, -2124.27}, {150.42,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{143.59, -2109.77}, {139.04, -2124.27}, {142.07,
       -2142.87}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{163.32, -2181.04}, {167.86, -2166.55}, {165.69,
       -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{158.86, -2181.04}, {154.32, -2166.55}, {157.35,
       -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{154.95,-2179.95},{167.80,-2192.80}},
    rotation= -0
  ),
```

```
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{163.32, -2109.77}, {167.86, -2124.27}, {165.69,
     -2142.44}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{158.86, -2109.77}, {154.32, -2124.27}, {157.35,
     -2142.87}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{178.59, -2181.04}, {183.13, -2166.55}, {180.97,
     -2148.38}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{174.13, -2181.04}, {169.59, -2166.55}, {172.62,
     -2147.95}},
  rotation= -0
),
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {219,0,0},
  extent= {{170.22,-2179.95},{183.07,-2192.80}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{178.59, -2109.77}, {183.13, -2124.27}, {180.97,
     -2142.44}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{174.13, -2109.77}, {169.59, -2124.27}, {172.62,
     -2142.87}},
  rotation= -0
```

```
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{185.50,-2098.02},{198.35,-2110.86}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{2.21,-2098.02},{15.06,-2110.86}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{17.48,-2098.02},{30.33,-2110.86}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{32.76,-2098.02},{45.60,-2110.86}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{48.03,-2098.02},{60.88,-2110.86}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{63.30,-2098.02},{76.15,-2110.86}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{78.58,-2098.02},{91.43,-2110.86}},
```

```
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{93.85,-2098.02},{106.70,-2110.86}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{109.13,-2098.02},{121.98,-2110.86}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{124.40,-2098.02},{137.25,-2110.86}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{139.68,-2098.02},{152.52,-2110.86}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{154.95,-2098.02},{167.80,-2110.86}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{170.22,-2098.02},{183.07,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{100, -2245.04}, {100, -2192.79}},
```

```
        rotation= -0
      )
    }
  )
);
end LipidBilayer;
```

Listing 10: `HHmodelica/Icons/OpenChannel.mo`

```
within HHmodelica.Icons;
model OpenChannel "pore that is open on both sides"
annotation(
  Icon(
    coordinateSystem(
      preserveAspectRatio= false,
      extent= {{-100,-100},{100,100}}
    ),
    graphics= {
      Polygon(
        origin= {-100,2145.04},
        lineThickness= 0.25,
        pattern= LinePattern.Solid,
        points= {{62.67, -2098.02}, {62.67, -2192.80}, {76.96, -2192.80},
           {82.99, -2186.78}, {82.99, -2145.04}, {82.99, -2115.02},
           {82.99, -2098.02}},
        fillPattern= FillPattern.Solid,
        fillColor= {124,154,239},
        lineColor= {0,0,0},
        rotation= -0
      ),
      Polygon(
        origin= {-100,2145.04},
        lineThickness= 0.25,
        pattern= LinePattern.Solid,
        points= {{137.43, -2098.02}, {137.43, -2192.80}, {123.13,
           -2192.80}, {117.11, -2186.78}, {117.11, -2145.04}, {117.11,
           -2115.02}, {117.11, -2098.02}},
        fillPattern= FillPattern.Solid,
        fillColor= {124,154,239},
        lineColor= {0,0,0},
        rotation= -0
      )
    }
  )
);
end OpenChannel;
```

Listing 11: `HHmodelica/Icons/Activatable.mo`

```
within HHmodelica.Icons;
model Activatable "pore that is open on the inside and closed on the
   outside"
annotation(
  Icon(
    coordinateSystem(
      preserveAspectRatio= false,
      extent= {{-100,-100},{100,100}}
    ),
    graphics= {
      Polygon(
        origin= {-100,2145.04},
        lineThickness= 0.25,
```

```
        pattern= LinePattern.Solid,
        points= {{62.67, -2098.02}, {62.67, -2192.80}, {76.96, -2192.80},
           {82.99, -2186.78}, {82.99, -2145.04}, {100.05, -2115.02},
           {100.05, -2098.02}},
        fillPattern= FillPattern.Solid,
        fillColor= {124,154,239},
        lineColor= {0,0,0},
        rotation= -0
      ),
      Polygon(
        origin= {-100,2145.04},
        lineThickness= 0.25,
        pattern= LinePattern.Solid,
        points= {{137.43, -2098.02}, {137.43, -2192.80}, {123.13,
           -2192.80}, {117.11, -2186.78}, {117.11, -2145.04}, {100.05,
           -2115.02}, {100.05, -2098.02}},
        fillPattern= FillPattern.Solid,
        fillColor= {124,154,239},
        lineColor= {0,0,0},
        rotation= -0
      )
    }
  )
);
end Activatable;
```

Listing 12: `HHmodelica/Icons/Inactivatable.mo`

```
within HHmodelica.Icons;
model Inactivatable "hinged lid for ion channel"
annotation(
  Icon(
    coordinateSystem(
      preserveAspectRatio= false,
      extent= {{-100,-100},{100,100}}
    ),
    graphics= {
      Polygon(
        origin= {-100,2145.04},
        lineThickness= 0.25,
        pattern= LinePattern.Solid,
        points= {{62.67, -2192.80}, {58.84, -2188.42}, {2.55, -2237.69},
           {6.38, -2242.07}, {17.17, -2232.62}, {25.67, -2233.19}, {51.34,
            -2210.71}, {51.91, -2202.22}},
        fillPattern= FillPattern.Solid,
        fillColor= {180,181,183},
        lineColor= {0,0,0},
        rotation= -0
      )
    }
  )
);
end Inactivatable;
```

Listing 13: `HHmodelica/Icons/IonChannel.mo`

```
within HHmodelica.Icons;
model IonChannel "base model for ion channel with gap in lipid bilayer"
annotation(
  Icon(
    coordinateSystem(
      preserveAspectRatio= false,
```

```
    extent= {{-100,-100},{100,100}}
  ),
  graphics= {
    Rectangle(
      origin= {-100,2145.04},
      lineThickness= 1,
      pattern= LinePattern.None,
      fillPattern= FillPattern.Solid,
      fillColor= {230,230,230},
      extent= {{0.22,-2098.02},{200.05,-2245.04}},
      rotation= -0
    ),
    Line(
      origin= {-100,2145.04},
      color= {0,0,0},
      pattern= LinePattern.Solid,
      thickness= 0.25,
      points= {{193.91, -2181.04}, {198.45, -2166.55}, {196.29,
         -2148.38}},
      rotation= -0
    ),
    Line(
      origin= {-100,2145.04},
      color= {0,0,0},
      pattern= LinePattern.Solid,
      thickness= 0.25,
      points= {{189.46, -2181.04}, {184.91, -2166.55}, {187.94,
         -2147.95}},
      rotation= -0
    ),
    Ellipse(
      origin= {-100,2145.04},
      lineThickness= 0.86,
      pattern= LinePattern.None,
      fillPattern= FillPattern.Solid,
      fillColor= {219,0,0},
      extent= {{185.55,-2179.95},{198.39,-2192.80}},
      rotation= -0
    ),
    Line(
      origin= {-100,2145.04},
      color= {0,0,0},
      pattern= LinePattern.Solid,
      thickness= 0.25,
      points= {{193.91, -2109.77}, {198.45, -2124.27}, {196.29,
         -2142.44}},
      rotation= -0
    ),
    Line(
      origin= {-100,2145.04},
      color= {0,0,0},
      pattern= LinePattern.Solid,
      thickness= 0.25,
      points= {{189.46, -2109.77}, {184.91, -2124.27}, {187.94,
         -2142.87}},
      rotation= -0
    ),
    Ellipse(
      origin= {-100,2145.04},
      lineThickness= 0.86,
      pattern= LinePattern.None,
      fillPattern= FillPattern.Solid,
```

```
    fillColor= {219,0,0},
    extent= {{185.55,-2098.02},{198.39,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{10.62, -2181.04}, {15.17, -2166.55}, {13, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{6.17, -2181.04}, {1.62, -2166.55}, {4.65, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{2.26,-2179.95},{15.10,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{10.62, -2109.77}, {15.17, -2124.27}, {13, -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{6.17, -2109.77}, {1.62, -2124.27}, {4.65, -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{2.26,-2098.02},{15.10,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{100.05, -2045.04}, {100.05, -2098.03}},
    rotation= -0
  ),
```

```
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{25.90, -2181.04}, {30.44, -2166.55}, {28.28, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{21.44, -2181.04}, {16.90, -2166.55}, {19.93, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{17.53,-2179.95},{30.38,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{25.90, -2109.77}, {30.44, -2124.27}, {28.28, -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{21.44, -2109.77}, {16.90, -2124.27}, {19.93, -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{17.53,-2098.02},{30.38,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{41.17, -2181.04}, {45.71, -2166.55}, {43.55, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
```

```
    thickness= 0.25,
    points= {{36.71, -2181.04}, {32.17, -2166.55}, {35.20, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{32.80,-2179.95},{45.65,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{41.17, -2109.77}, {45.71, -2124.27}, {43.55, -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{36.71, -2109.77}, {32.17, -2124.27}, {35.20, -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{32.80,-2098.02},{45.65,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{56.45, -2181.04}, {60.99, -2166.55}, {58.82, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{51.99, -2181.04}, {47.45, -2166.55}, {50.47, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{48.08,-2179.95},{60.93,-2192.80}},
    rotation= -0
```

```
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{56.45, -2109.77}, {60.99, -2124.27}, {58.82, -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{51.99, -2109.77}, {47.45, -2124.27}, {50.47, -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{48.08,-2098.02},{60.93,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{148.09, -2181.04}, {152.63, -2166.55}, {150.47,
       -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{143.63, -2181.04}, {139.09, -2166.55}, {142.12,
       -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{139.72,-2179.95},{152.57,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{148.09, -2109.77}, {152.63, -2124.27}, {150.47,
       -2142.44}},
    rotation= -0
  ),
```

```
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{143.63, -2109.77}, {139.09, -2124.27}, {142.12,
     -2142.87}},
  rotation= -0
),
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {219,0,0},
  extent= {{139.72,-2098.02},{152.57,-2110.86}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{163.36, -2181.04}, {167.91, -2166.55}, {165.74,
     -2148.38}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{158.91, -2181.04}, {154.36, -2166.55}, {157.39,
     -2147.95}},
  rotation= -0
),
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {219,0,0},
  extent= {{155,-2179.95},{167.85,-2192.80}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{163.36, -2109.77}, {167.91, -2124.27}, {165.74,
     -2142.44}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{158.91, -2109.77}, {154.36, -2124.27}, {157.39,
     -2142.87}},
  rotation= -0
```

```
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{155,-2098.02},{167.85,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{178.64, -2181.04}, {183.18, -2166.55}, {181.02,
       -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{174.18, -2181.04}, {169.64, -2166.55}, {172.67,
       -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{170.27,-2179.95},{183.12,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{178.64, -2109.77}, {183.18, -2124.27}, {181.02,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{174.18, -2109.77}, {169.64, -2124.27}, {172.67,
       -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{170.27,-2098.02},{183.12,-2110.86}},
```

```
        rotation= -0
      ),
      Line(
        origin= {-100,2145.04},
        color= {0,0,0},
        pattern= LinePattern.Solid,
        thickness= 0.25,
        points= {{100.05, -2245.04}, {100.05, -2192.79}},
        rotation= -0
      )
    }
  )
);
end IonChannel;
```

Listing 14: `HHmodelica/Icons/CurrentClamp.mo`

```
within HHmodelica.Icons;
model CurrentClamp "two electrodes sticking through lipid bilayer"
annotation(
  Icon(
    coordinateSystem(
      preserveAspectRatio= false,
      extent= {{-100,-100},{100,100}}
    ),
    graphics= {
      Line(
        origin= {-100,2145.04},
        color= {0,0,0},
        pattern= LinePattern.Solid,
        thickness= 0.25,
        points= {{132.77, -2181.04}, {137.31, -2166.55}, {135.15,
           -2148.38}},
        rotation= -0
      ),
      Rectangle(
        origin= {-100,2145.04},
        lineThickness= 1,
        pattern= LinePattern.None,
        fillPattern= FillPattern.Solid,
        fillColor= {230,230,230},
        extent= {{0.17,-2098.02},{200,-2245.04}},
        rotation= -0
      ),
      Line(
        origin= {-100,2145.04},
        color= {0,0,0},
        pattern= LinePattern.Solid,
        thickness= 0.25,
        points= {{193.86, -2181.04}, {198.41, -2166.55}, {196.24,
           -2148.38}},
        rotation= -0
      ),
      Line(
        origin= {-100,2145.04},
        color= {0,0,0},
        pattern= LinePattern.Solid,
        thickness= 0.25,
        points= {{189.41, -2181.04}, {184.86, -2166.55}, {187.89,
           -2147.95}},
        rotation= -0
      ),
```

```
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {219,0,0},
  extent= {{185.50,-2179.95},{198.35,-2192.80}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{10.57, -2181.04}, {15.12, -2166.55}, {12.95, -2148.38}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{6.12, -2181.04}, {1.58, -2166.55}, {4.60, -2147.95}},
  rotation= -0
),
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {219,0,0},
  extent= {{2.21,-2179.95},{15.06,-2192.80}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{100, -2045.04}, {100, -2098.03}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{25.85, -2181.04}, {30.39, -2166.55}, {28.23, -2148.38}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{21.39, -2181.04}, {16.85, -2166.55}, {19.88, -2147.95}},
  rotation= -0
),
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
```

```
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{17.48,-2179.95},{30.33,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{41.12, -2181.04}, {45.67, -2166.55}, {43.50, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{36.67, -2181.04}, {32.12, -2166.55}, {35.15, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{32.76,-2179.95},{45.60,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{56.40, -2181.04}, {60.94, -2166.55}, {58.78, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{51.94, -2181.04}, {47.40, -2166.55}, {50.43, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{48.03,-2179.95},{60.88,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{71.67, -2181.04}, {76.21, -2166.55}, {74.05, -2148.38}},
    rotation= -0
```

```
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{67.21, -2181.04}, {62.67, -2166.55}, {65.70, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{63.30,-2179.95},{76.15,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{86.95, -2181.04}, {91.49, -2166.55}, {89.32, -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{82.49, -2181.04}, {77.95, -2166.55}, {80.97, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{78.58,-2179.95},{91.43,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{102.22, -2181.04}, {106.76, -2166.55}, {104.60,
       -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{97.76, -2181.04}, {93.22, -2166.55}, {96.25, -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
```

```
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{93.85,-2179.95},{106.70,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{117.49, -2181.04}, {122.04, -2166.55}, {119.87,
       -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{113.04, -2181.04}, {108.49, -2166.55}, {111.52,
       -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{109.13,-2179.95},{121.98,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{128.31, -2181.04}, {123.77, -2166.55}, {126.80,
       -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{124.40,-2179.95},{137.25,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{148.04, -2181.04}, {152.58, -2166.55}, {150.42,
       -2148.38}},
    rotation= -0
  ),
  Line(
```

```
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{143.59, -2181.04}, {139.04, -2166.55}, {142.07,
       -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{139.68,-2179.95},{152.52,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{163.32, -2181.04}, {167.86, -2166.55}, {165.69,
       -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{158.86, -2181.04}, {154.32, -2166.55}, {157.35,
       -2147.95}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{154.95,-2179.95},{167.80,-2192.80}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{178.59, -2181.04}, {183.13, -2166.55}, {180.97,
       -2148.38}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{174.13, -2181.04}, {169.59, -2166.55}, {172.62,
       -2147.95}},
    rotation= -0
  ),
```

```
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {219,0,0},
  extent= {{170.22,-2179.95},{183.07,-2192.80}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{100, -2245.04}, {100, -2192.79}},
  rotation= -0
),
Polygon(
  origin= {-100,2145.04},
  lineThickness= 0.25,
  pattern= LinePattern.Solid,
  points= {{198.69, -2060.29}, {187.99, -2085.82}, {113.84,
     -2218.63}, {154.72, -2071.87}, {165.42, -2046.35}},
  fillPattern= FillPattern.Solid,
  fillColor= {210,246,244},
  lineColor= {0,0,0},
  rotation= -0
),
Polygon(
  origin= {-100,2145.04},
  lineThickness= 0.25,
  pattern= LinePattern.Solid,
  points= {{1.31, -2060.29}, {12.01, -2085.82}, {86.16, -2218.63},
     {45.28, -2071.87}, {34.58, -2046.35}},
  fillPattern= FillPattern.Solid,
  fillColor= {210,246,244},
  lineColor= {0,0,0},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{193.86, -2109.77}, {198.41, -2124.27}, {196.24,
     -2142.44}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{189.41, -2109.77}, {184.86, -2124.27}, {187.89,
     -2142.87}},
  rotation= -0
),
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
```

```
    fillColor= {219,0,0},
    extent= {{185.50,-2098.02},{198.35,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{10.57, -2109.77}, {15.12, -2124.27}, {12.95, -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{6.12, -2109.77}, {1.58, -2124.27}, {4.60, -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{2.21,-2098.02},{15.06,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{25.85, -2109.77}, {30.39, -2124.27}, {28.23, -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{21.39, -2109.77}, {16.85, -2124.27}, {19.88, -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{17.48,-2098.02},{30.33,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{41.12, -2109.77}, {45.67, -2124.27}, {43.50, -2142.44}},
    rotation= -0
  ),
```

```
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{36.67, -2109.77}, {32.12, -2124.27}, {35.15, -2142.87}},
  rotation= -0
),
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {219,0,0},
  extent= {{32.76,-2098.02},{45.60,-2110.86}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{56.40, -2109.77}, {60.94, -2124.27}, {58.78, -2142.44}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{51.94, -2109.77}, {47.40, -2124.27}, {50.43, -2142.87}},
  rotation= -0
),
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
  fillPattern= FillPattern.Solid,
  fillColor= {219,0,0},
  extent= {{48.03,-2098.02},{60.88,-2110.86}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{71.67, -2109.77}, {76.21, -2124.27}, {74.05, -2142.44}},
  rotation= -0
),
Line(
  origin= {-100,2145.04},
  color= {0,0,0},
  pattern= LinePattern.Solid,
  thickness= 0.25,
  points= {{67.21, -2109.77}, {62.67, -2124.27}, {65.70, -2142.87}},
  rotation= -0
),
Ellipse(
  origin= {-100,2145.04},
  lineThickness= 0.86,
  pattern= LinePattern.None,
```

```
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{63.30,-2098.02},{76.15,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{86.95, -2109.77}, {91.49, -2124.27}, {89.32, -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{82.49, -2109.77}, {77.95, -2124.27}, {80.97, -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{78.58,-2098.02},{91.43,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{102.22, -2109.77}, {106.76, -2124.27}, {104.60,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{97.76, -2109.77}, {93.22, -2124.27}, {96.25, -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{93.85,-2098.02},{106.70,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
```

```
    points= {{117.49, -2109.77}, {122.04, -2124.27}, {119.87,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{113.04, -2109.77}, {108.49, -2124.27}, {111.52,
       -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{109.13,-2098.02},{121.98,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{132.77, -2109.77}, {137.31, -2124.27}, {135.15,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{128.31, -2109.77}, {123.77, -2124.27}, {126.80,
       -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{124.40,-2098.02},{137.25,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{148.04, -2109.77}, {152.58, -2124.27}, {150.42,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
```

```
    thickness= 0.25,
    points= {{143.59, -2109.77}, {139.04, -2124.27}, {142.07,
       -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{139.68,-2098.02},{152.52,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{163.32, -2109.77}, {167.86, -2124.27}, {165.69,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{158.86, -2109.77}, {154.32, -2124.27}, {157.35,
       -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
    pattern= LinePattern.None,
    fillPattern= FillPattern.Solid,
    fillColor= {219,0,0},
    extent= {{154.95,-2098.02},{167.80,-2110.86}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{178.59, -2109.77}, {183.13, -2124.27}, {180.97,
       -2142.44}},
    rotation= -0
  ),
  Line(
    origin= {-100,2145.04},
    color= {0,0,0},
    pattern= LinePattern.Solid,
    thickness= 0.25,
    points= {{174.13, -2109.77}, {169.59, -2124.27}, {172.62,
       -2142.87}},
    rotation= -0
  ),
  Ellipse(
    origin= {-100,2145.04},
    lineThickness= 0.86,
```

```
          pattern= LinePattern.None,
          fillPattern= FillPattern.Solid,
          fillColor= {219,0,0},
          extent= {{170.22,-2098.02},{183.07,-2110.86}},
          rotation= -0
        )
      }
    )
);
end CurrentClamp;
```

A.3. Countering reproducibility issues in mathematical models with software engineering techniques: A case study using a one-dimensional mathematical model of the atrioventricular node

Data Supplement

Countering reproducibility issues in mathematical models with software engineering techniques: A case study using a one-dimensional mathematical model of the atrioventricular node

Christopher Schölzel[1,4*] Valeria Blesius[1,4] Gernot Ernst[2, 3] Alexander Goesmann[4] Andreas Dominik[1]

1 Technische Hochschule Mittelhessen – THM University of Applied Sciences, Giessen, Germany
2 Vestre Viken Hospital Trust, Kongsberg, Norway
3 University of Oslo, Norway
4 Justus Liebig University Giessen, Germany

* christopher.schoelzel@mni.thm.de

1 Supplementary Figures

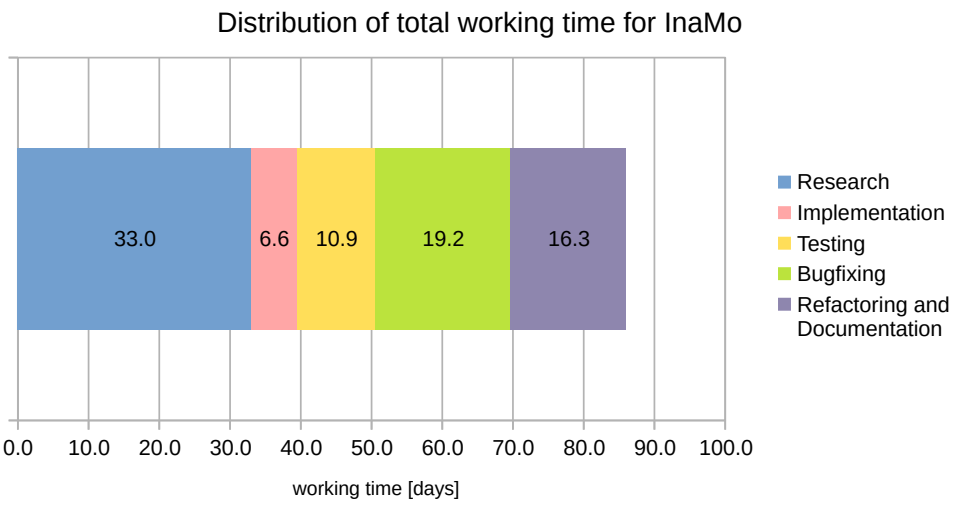


Figure 1. Estimation of working time distribution for the development of the Modelica version of the Inada model. Estimates were taken from time stamps of PDF comments in the articles listed in Figure 1 and of commits in the version control system Git. In a first step, we always assumed that a full day was spent on literature research or on the code in the repository. To correct for overlap on days where we worked both on the repository and added comments in the PDF documents, we scaled the whole dataset to the amount of unique days spend for development (86).

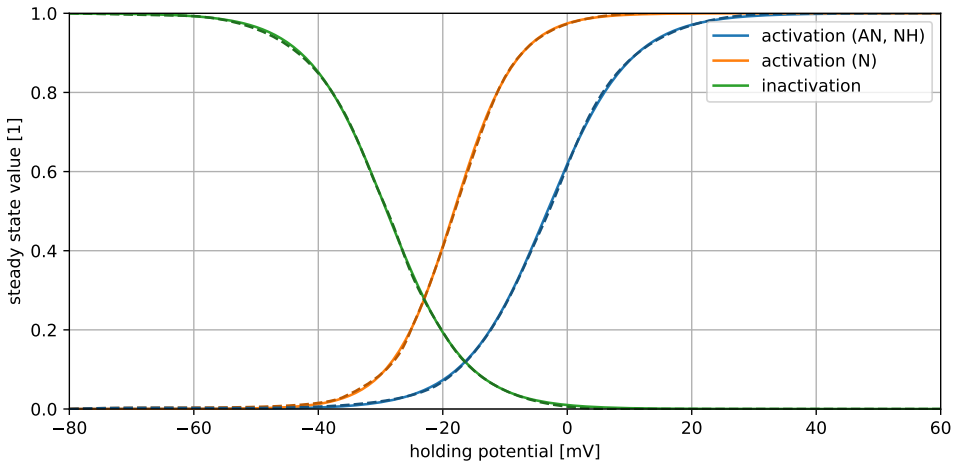


Figure 2. Steady states of gating variables in $I_{Ca,L}$. Solid lines: Simulation result of LTypeCalciumSteady. Dashed lines: Reference data extracted from Figure S1A and S1B in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

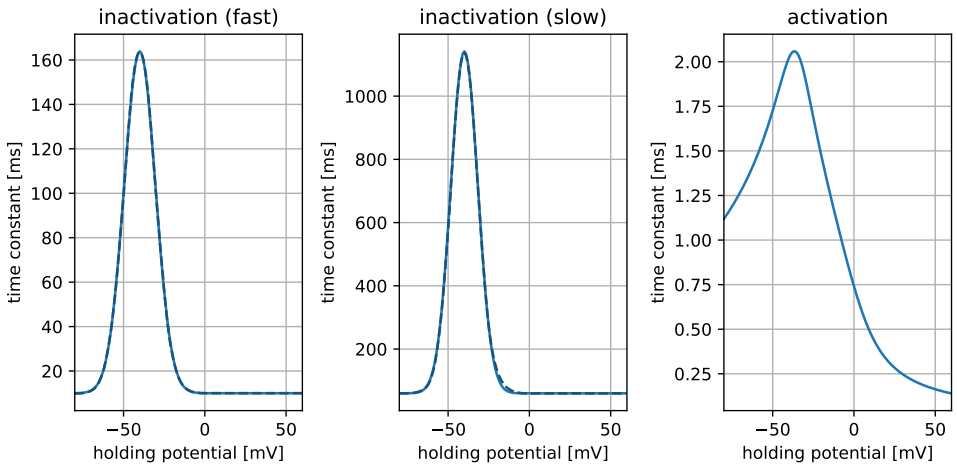


Figure 3. Time constants of gating variables in $I_{Ca,L}$. Solid lines: Simulation result of LTypeCalciumSteady. Dashed lines: Reference data extracted from Figure S1C (fast inactivation) and S1D (slow inactivation) in [1]. A reference plot for the activation gate is not provided in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

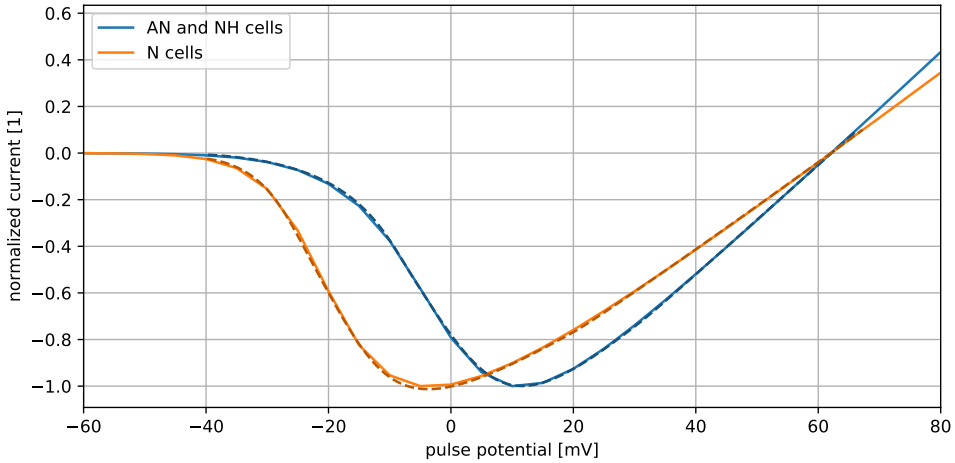


Figure 4. Reference plots for current-voltage relationship of $I_{Ca,L}$ obtained with a voltage pulse protocol with a holding potential of -70 mV, a holding duration of 5 s, and a pulse duration of 300 ms. Solid lines: Simulation result of LTypeCalciumIV. Dashed lines: Reference data extracted from Figure S1E in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

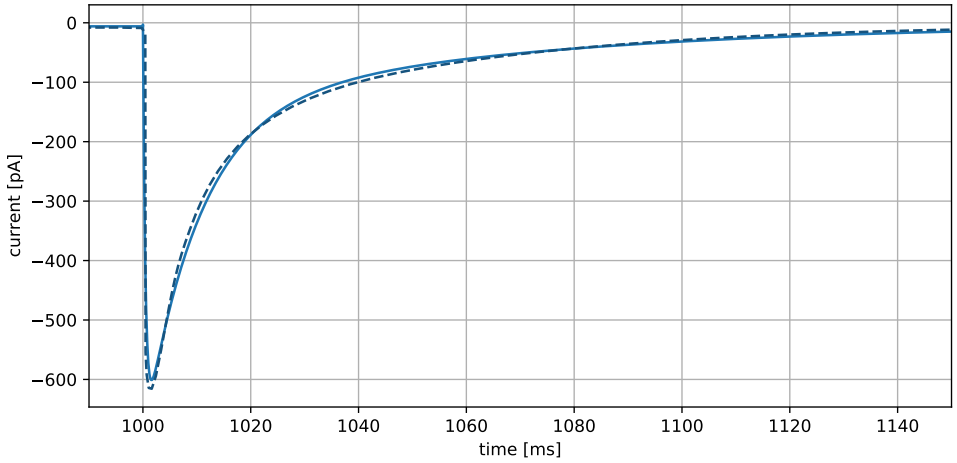


Figure 5. Time course of $I_{Ca,L}$ when switching from holding potential of -40 mV to prolonged stimulation at +10 mV. Solid line: Simulation result of LTypeCalciumStep. Dashed line: Reference data extracted from Figure S1H in [1]. The plots are in perfect agreement. Note, however, that while Inada *et al.* state that they used AN cells for plot S1H, NH cells had to be used instead to reach this agreement. Additionally, it seems that the x axis of Figure S1H is scaled differently than the measuring strip in the plot suggests. To obtain a good fit, time stamps extracted from the figure have to be multiplied by a scaling factor of 0.75. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

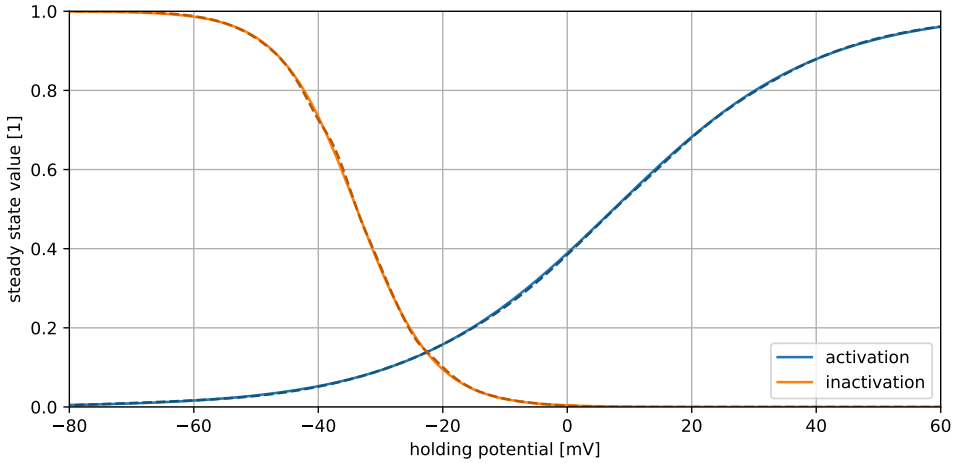


Figure 6. Steady states for activation and inactivation gates of I_{to}. Solid lines: Simulation result of TransientOutwardSteady. Dashed lines: Reference data extracted from Figures S2A and S2B in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

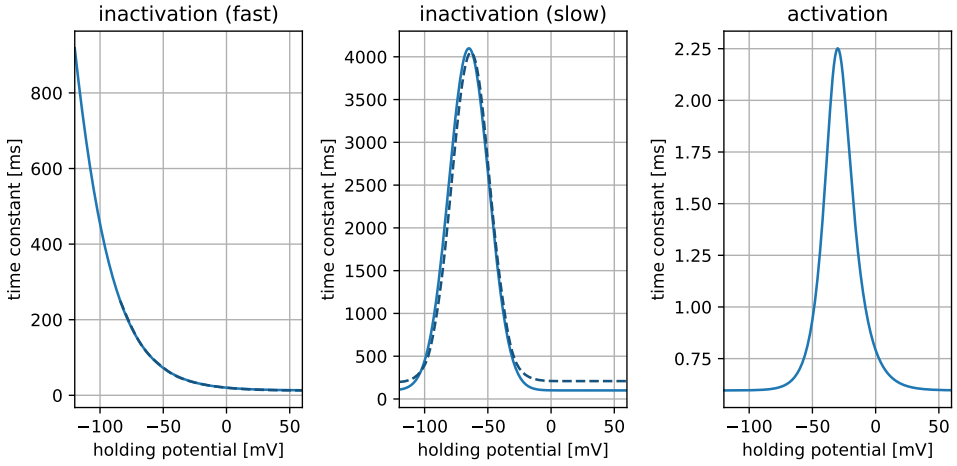


Figure 7. Time constants for gating variables in I_{to}. Solid lines: Simulation result of TransientOutwardSteady. Dashed lines: Reference data extracted from Figures S2C and S2D in [1]. A reference plot for activation is not provided in [1]. The fast inactivation is in perfect agreement, but the slow inactivation shows a lower minimum value. It seems that the minimum in Figure S2D is closer to 0.2 s than 0.1 s as given in [1] and the C++ code. Since article and C++ code agree on the value 0.1 s, we assume that Figure S2D was generated with an older version of the model. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

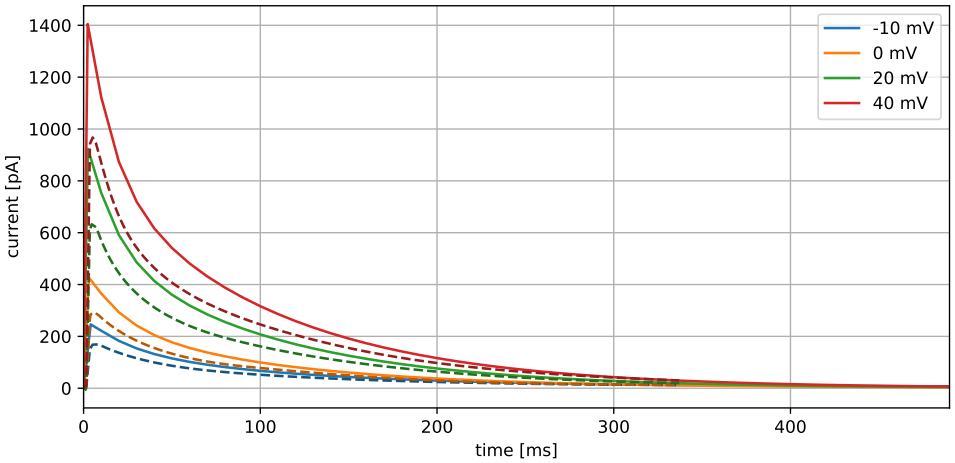


Figure 8. Time course of I_{to} after a stimulation to different voltages for 500 ms from a holding potential of -80 mV that was kept for 20 s before the stimulation. Solid lines: Simulation result of TransientOutwardIV. Dashed lines: Reference data extracted from Figure S2E in [1]. While Inada *et al.* state that they used AN cells for plot S2E, NH cells were used instead to reach a better agreement to the reference plot. The absolute values for the current are larger for InaMo than for the reference. This may be due to differences in the holding duration, which was not reported by Inada *et al.*. We chose a holding duration 20 s for a full return to the steady state, since otherwise the previous pulse would influence the following. The differences to the reference vanish when a scaling factor of 0.75 is applied to the simulation results. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

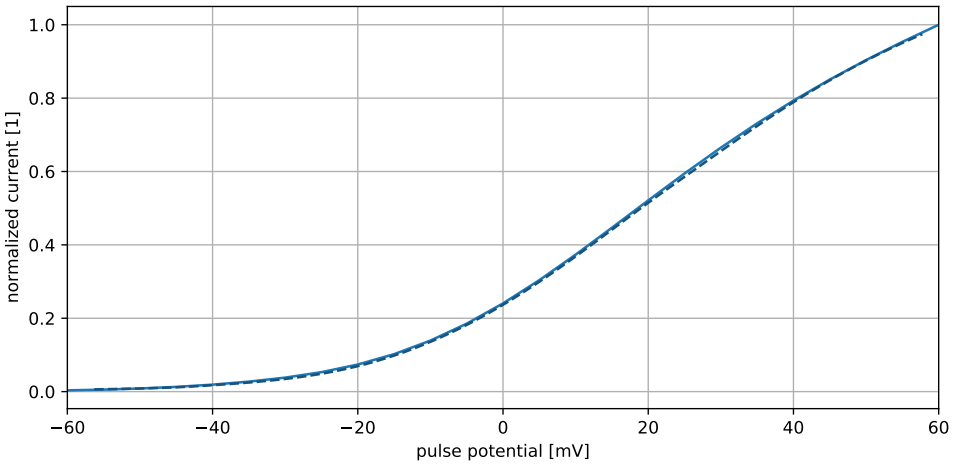


Figure 9. Current-voltage relationship of I_{to} obtained with a voltage pulse protocol with a holding potential of -80 mV, a holding duration of 20 s, and a pulse duration of 500 ms. Solid line: Simulation result of TransientOutwardIV. Dashed line: Reference data extracted from Figure S2F in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

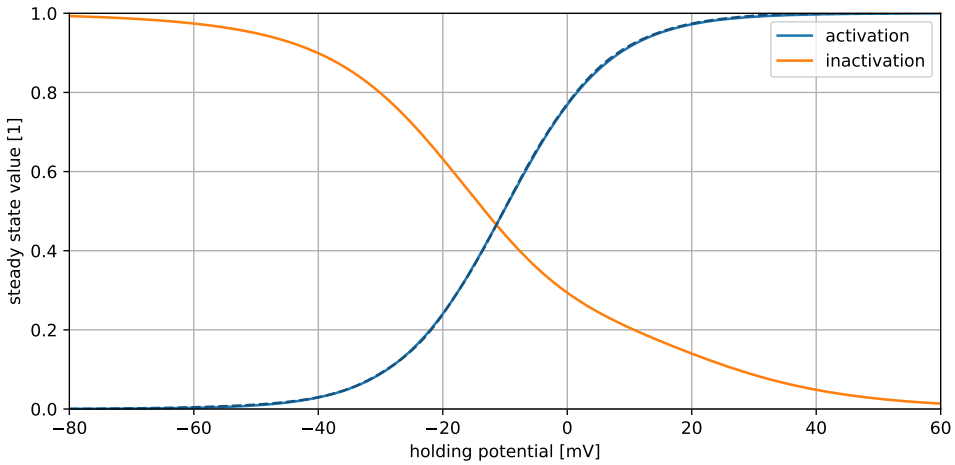


Figure 10. Steady state of the gating variables for $I_{K,r}$. Solid lines: Simulation result of RapidDelayedRectifierSteady. Dashed lines: Reference data extracted from Figure S3A in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

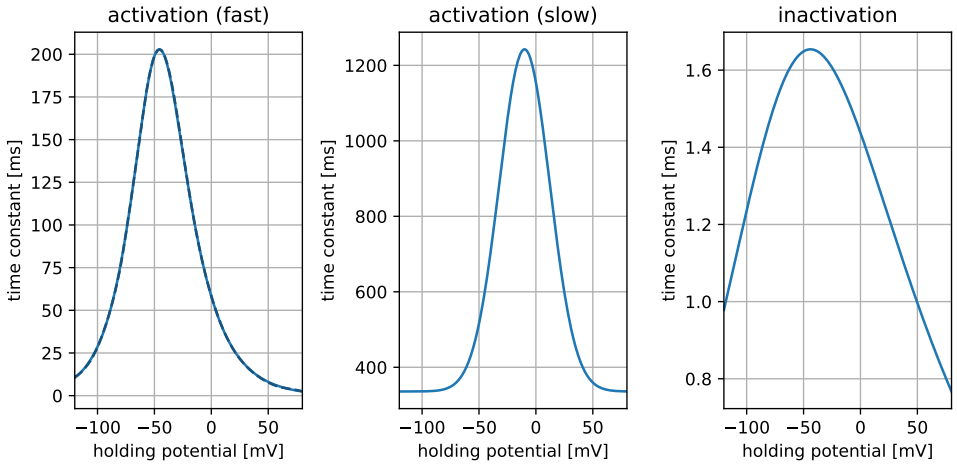


Figure 11. Time constants of the gating variables for $I_{K,r}$. Solid lines: Simulation result of RapidDelayedRectifierSteady. Dashed lines: Reference data extracted from Figure S3B in [1] (showing fast activation). Reference plots for activation are not provided in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

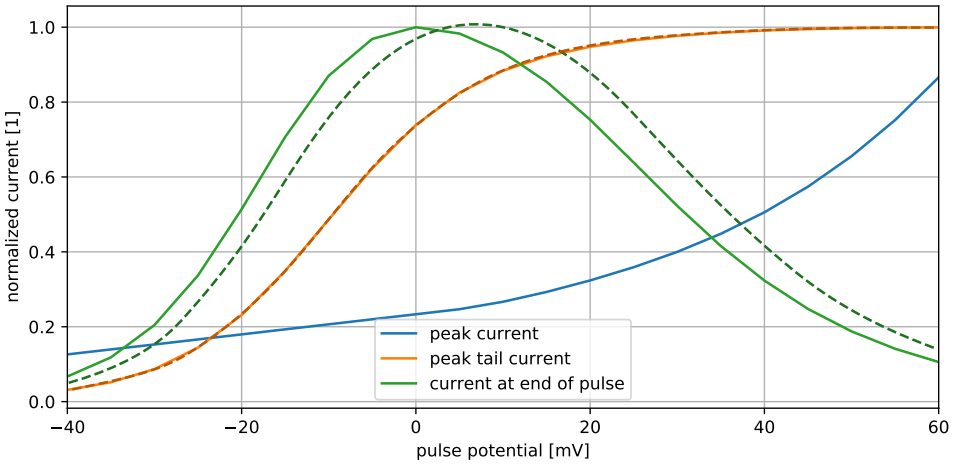


Figure 12. Current-voltage relationship of $I_{K,r}$ obtained with a voltage pulse protocol with a holding potential of -40 mV, a holding duration of 5 s, and a pulse duration of 500 ms. Solid lines: Simulation result of RapidDelayedRectifierIV. Dashed lines: Reference data extracted from Figures S3C and S3D in [1]. Figure S3C shows the current at the end of each stimulation pulse. Figure S3D shows the peak current obtained after the voltage shifts back from the pulse voltage to the holding potential. Data from Figure S3D is in perfect agreement with simulation results, but data from Figure S3C is shifted by 5 mV towards higher voltages. This could be explained if Inada *et al.* accidentally associated currents to the newly started pulse right after the current was measured instead of the previous pulse. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

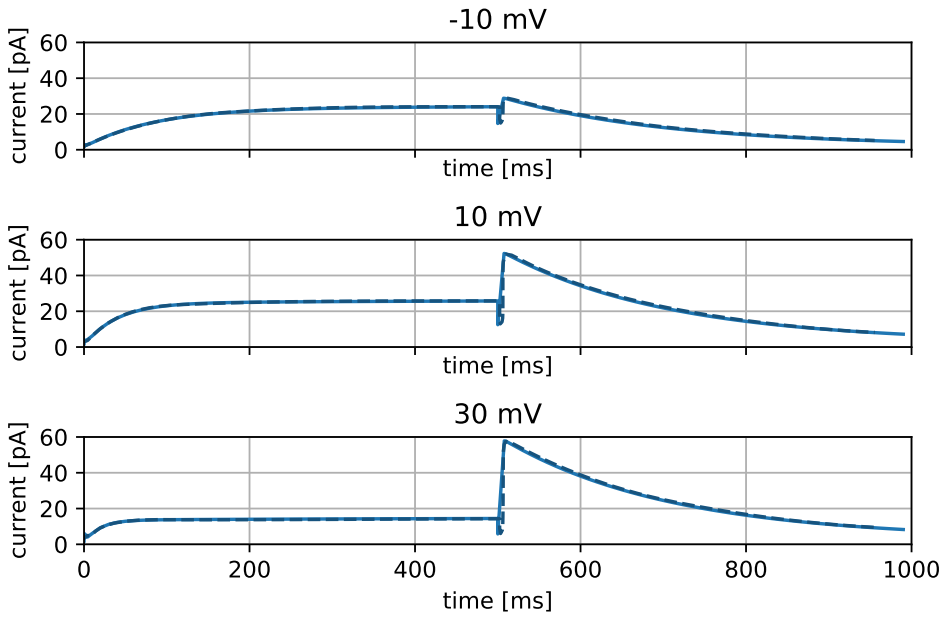


Figure 13. Current time course of $I_{K,r}$ after 500 ms pulses with different voltages from a holding potential of -40 mV which is held for 5 s. Solid lines: Simulation result of RapidDelayedRectifierIV. Dashed lines: Reference data extracted from Figure S3E in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

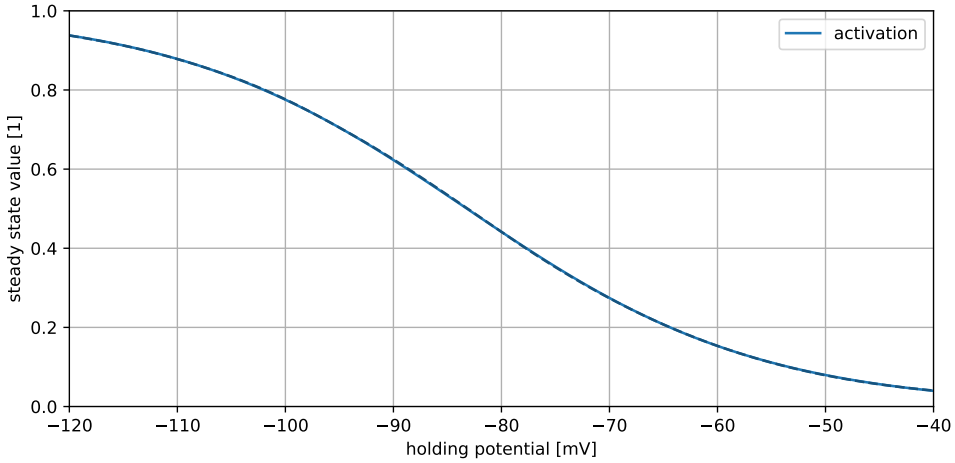


Figure 14. Steady state of activation gating variable in I_f. Solid line: Simulation result of HyperpolarizationActivatedSteady. Dashed line: Reference data extracted from Figure S4A in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

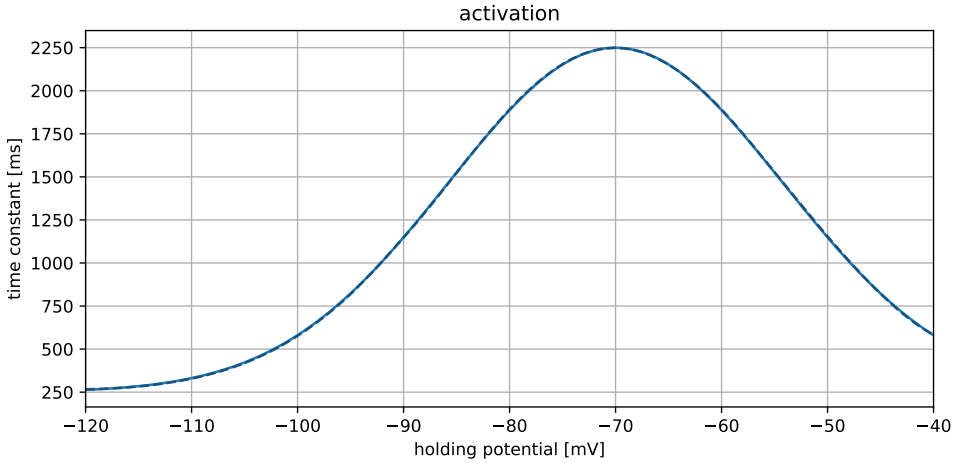


Figure 15. Time constant for activation gating variable in I_f. Solid line: Simulation result of HyperpolarizationActivatedSteady. Dashed line: Reference data extracted from Figure S4B in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

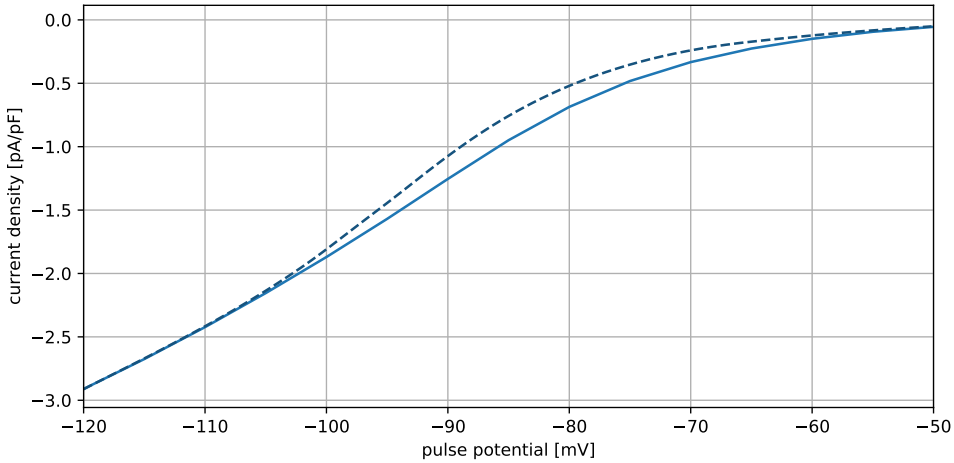


Figure 16. Current-voltage relationship of I_f obtained with a voltage pulse protocol with a holding potential of -50 mV, a holding duration of 20 s, and a pulse duration of 4 s. Solid line: Simulation result of HyperpolarizationActivatedIV. Dashed line: Reference data extracted from Figure S4C in [1]. The plots diverge for pulse voltages between -100 to -60 mV with higher current densities in the reference data. We have no good explanation for this difference, especially considering that Figure 17, which covers similar voltage ranges using the same data, shows a perfect agreement for I_f. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

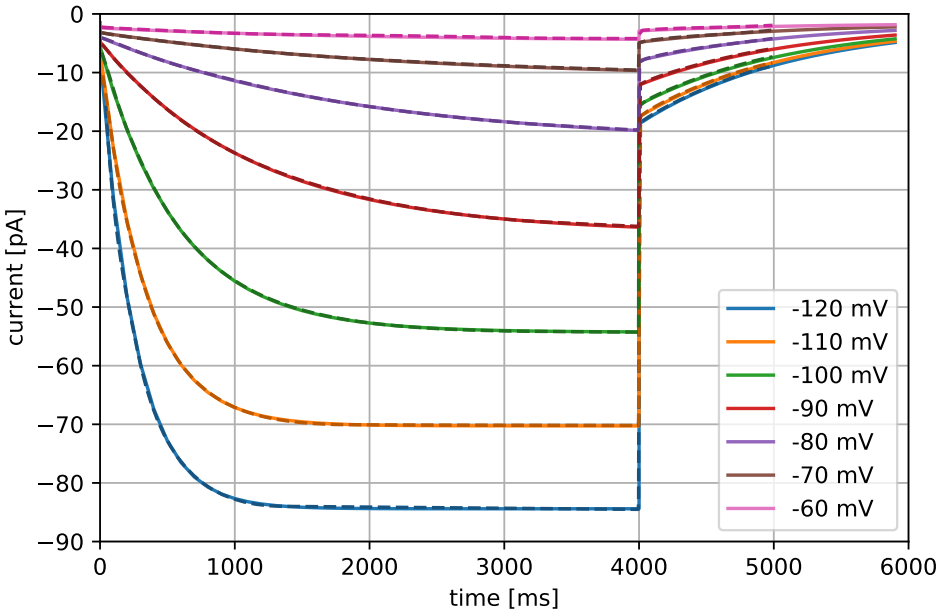


Figure 17. Time course of I_f after 4 s of stimulation to different voltages from a resting potential of -50 mV, which was held for 20 s. Solid lines: Simulation result of HyperpolarizationActivatedIV. Dashed lines: Reference data extracted from Figure S4D in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

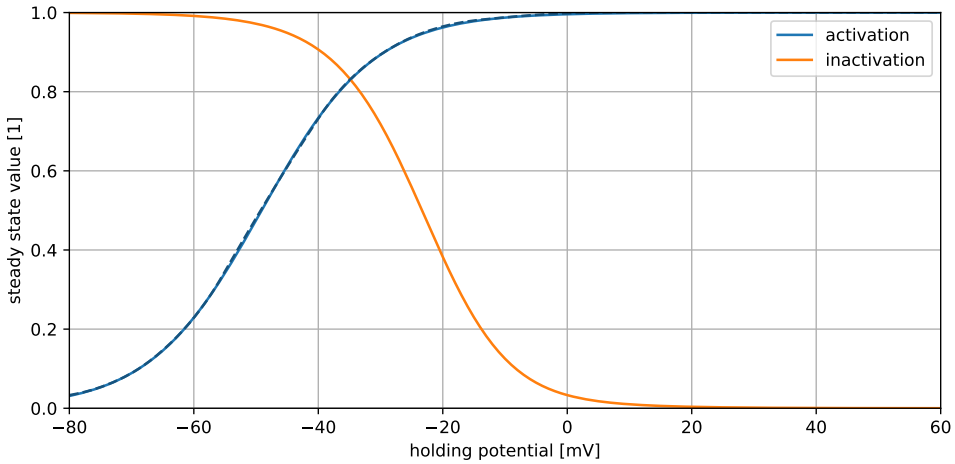


Figure 18. Steady state of gating variables in I_{st}. Solid lines: Simulation result of SustainedInwardSteady. Dashed lines: Reference data extracted from Figure S5A in [1]. A reference plot for inactivation is not provided in [1]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

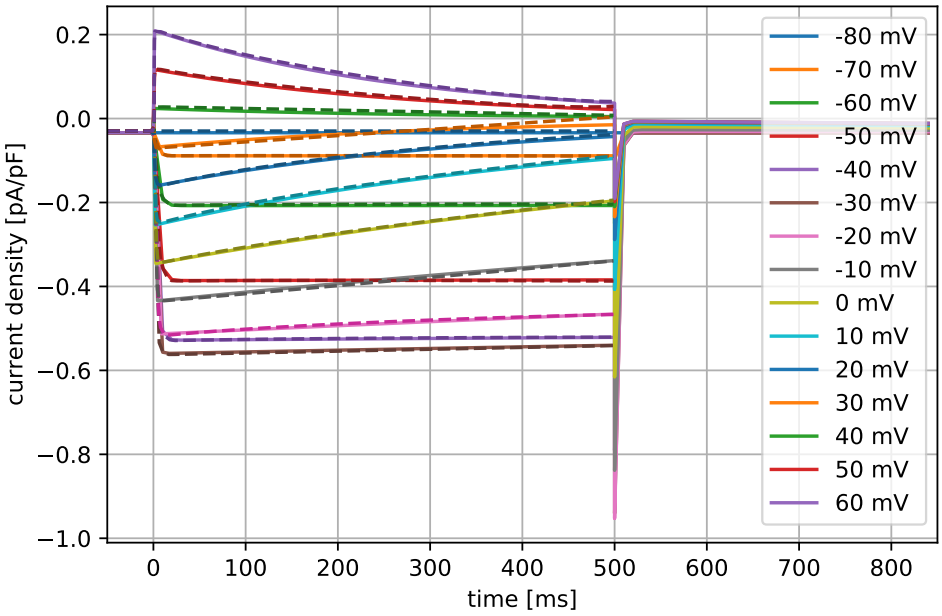


Figure 19. Current density time course of I_{st} during a voltage pulse protocol with a holding potential of -80 mV, a holding duration of 15 s, and a pulse duration of 500 ms. Solid lines: Simulation result of SustainedInwardIV. Dashed lines: Reference data extracted from Figure S5B in [1]. We had to change the parameter g_{st} to 0.27 nS instead of the default 0.1 nS reported in [1] in order to obtain a good fit to the reference data. Evidence that this is not due to an error in our implementation is provided by the good agreement between our results and those of [2] in Figures 29 and 30. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

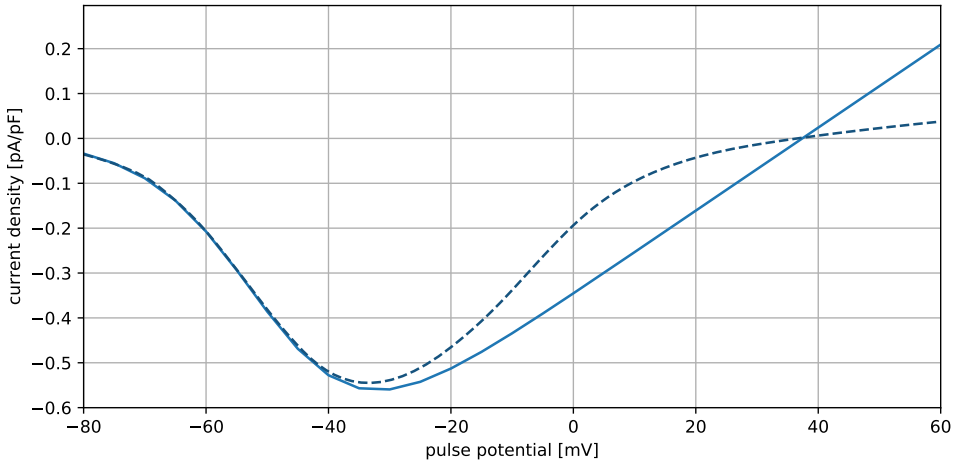


Figure 20. Current-voltage relationship of I_{st} obtained with a voltage pulse protocol with a holding potential of -80 mV, a holding duration of 15 s, and a pulse duration of 500 ms. Solid line: Simulation result of SustainedInwardIV. Dashed line: Reference data extracted from Figure S5C in [1]. As in the previous figure, we had to use $g_{st} = 0.27$ nS to obtain a good agreement. Additionally, the behavior for positive pulse voltages is different. We do not have a good explanation for this difference, but can only point to the reference plot by Kurata2002 *et al.*, which shows a behavior that is more similar to InaMo than to the plot by Inada *et al.* (see Figure 30). This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

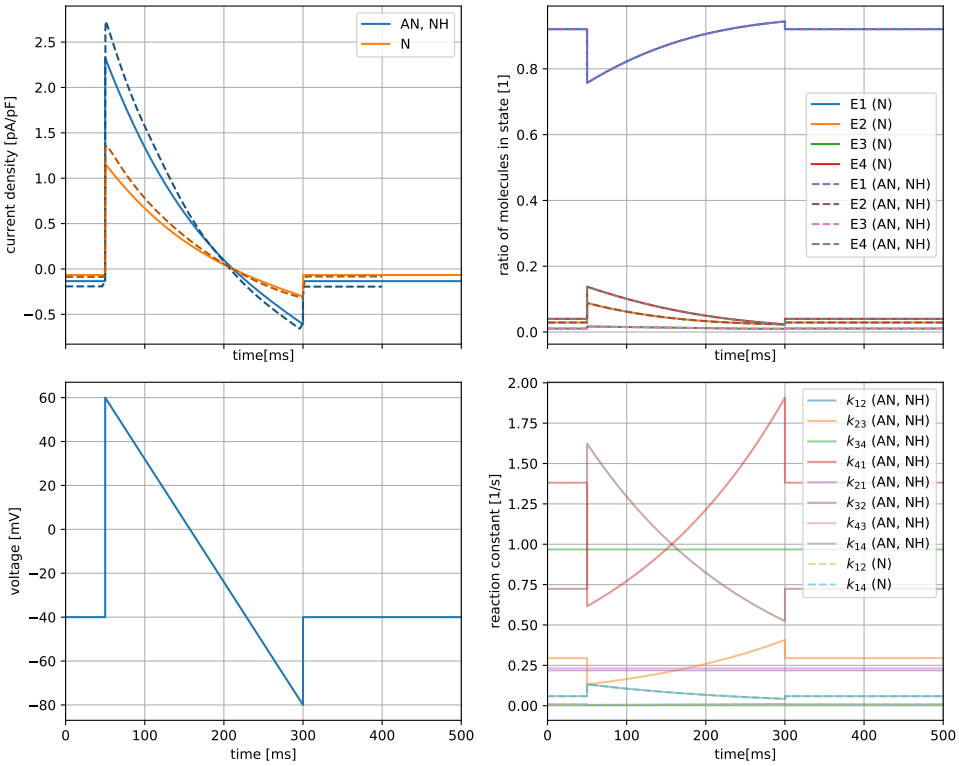


Figure 21. Current density time course of I_{NaCa} following a ramp-shaped input voltage. Solid lines: Simulation result of SodiumCalciumExchangerRampInada. Dashed lines: Reference data extracted from Figure S6A in [1]. The current density in InaMo is slightly lower than in the reference. This can be explained by the fact that Inada *et al.* do not state whether calcium concentrations were held constant for the experiment and if so, which value was assumed for $\left[Ca^{2+}\right]_{sub}$. Since they used Data from Convery *et al.* [3], we assume that the calcium and sodium concentrations should be similar to those used in this experiment ($\left[Na^{+}\right]_{i}$ = 10 mM, $\left[Na^{+}\right]_{o}$ = 140 mM, $\left[Ca^{2+}\right]_{o}$ = 2.5 mM). However, Convery *et al.* do not give a value for $\left[Ca^{2+}\right]_{sub}$ and both using all values from Table S15 of [1] and using all values from [3] does not reproduce the absolute values observed in Figure S6. We therefore used a mix of settings from both sources and manually changed $\left[Ca^{2+}\right]_{sub}$ until a good fit with the reference was achieved. The remaining differences disappear when the current density is multiplied with a scaling factor of 1.18. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

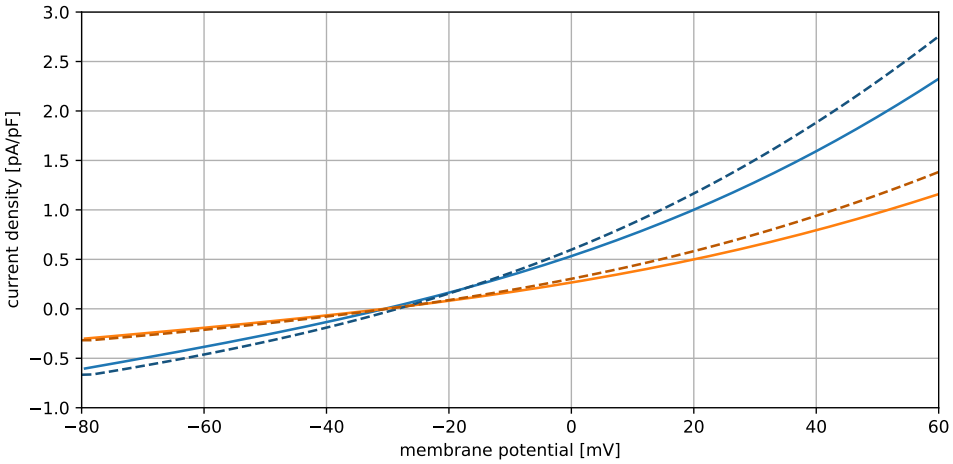


Figure 22. Current-voltage relationship of I_{NaCa} obtained following a ramp-shaped input voltage (see previous figure). Solid lines: Simulation result of SodiumCalciumExchangerRampInada. Dashed lines: Reference data extracted from Figure S6B in [1]. As in the previous figure, InaMo has slightly lower current densities, which can be explained by missing information about the experiment setup of Inada *et al.*. Like in the previous Figure, the remaining differences disappear when the current density is multiplied with a scaling factor of 1.18. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

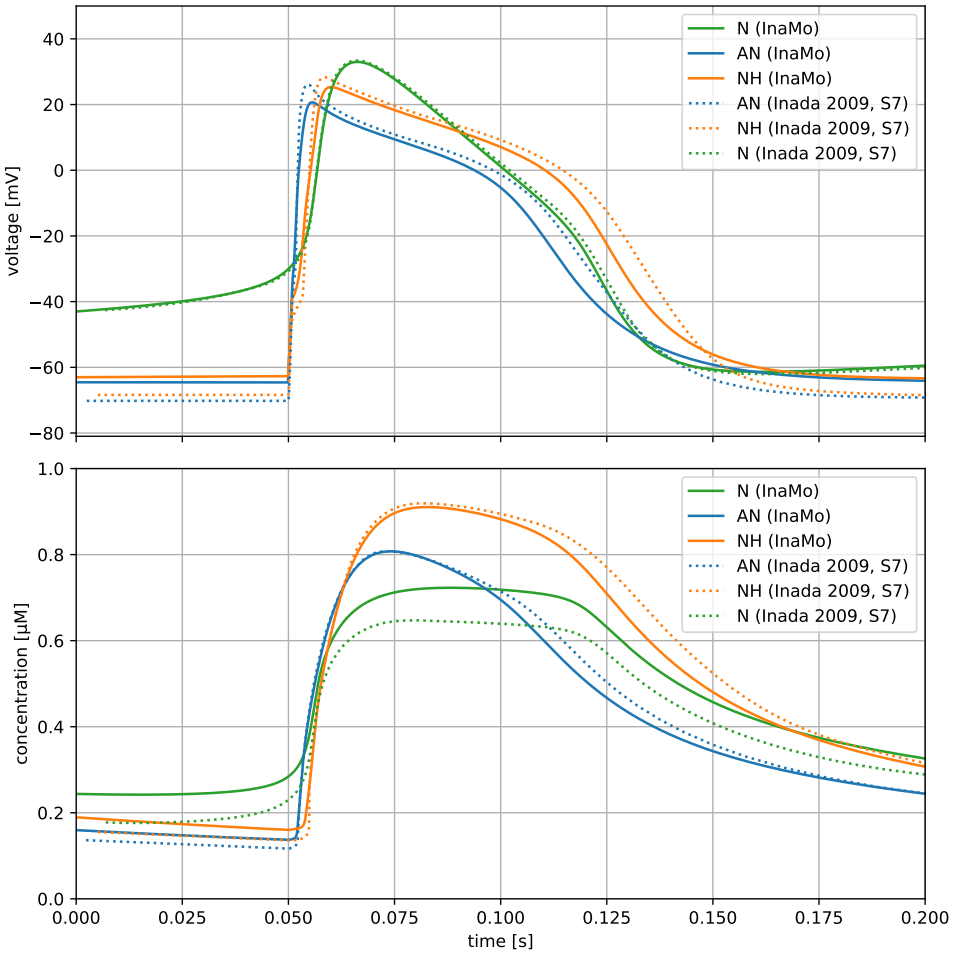


Figure 23. Time course of membrane voltage and $\left[Ca^{2+}\right]_i$ for the full cell models obtained by spontaneous activation (N cell) or stimulation with a current pulse protocol issuing pulses of -1.2 nA/-0.95 nA with a pulse duration of 1 ms after a holding period of 300 ms with a holding potential of 0 nA (AN/NH cell). Solid lines: Simulation result of AllCells. Dotted lines: Reference data extracted from Figure S7 in [1]. For AN and NH cells the voltage plot shows a lower resting potential and a slightly narrower action potential with respect to the reference. For N cells, $\left[Ca^{2+}\right]_i$ is significantly lower than in the reference. This can be explained by differences discussed in other figures. Additionally, the current pulse protocol is not given in [1]. We assume it is the same as for Figure 1 of [1], but this still only gives the stimulation current in terms of a multiplicative constant applied to an unspecified threshold current. We also changed the holding duration to 300 ms instead of 350 ms as it yields a better agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

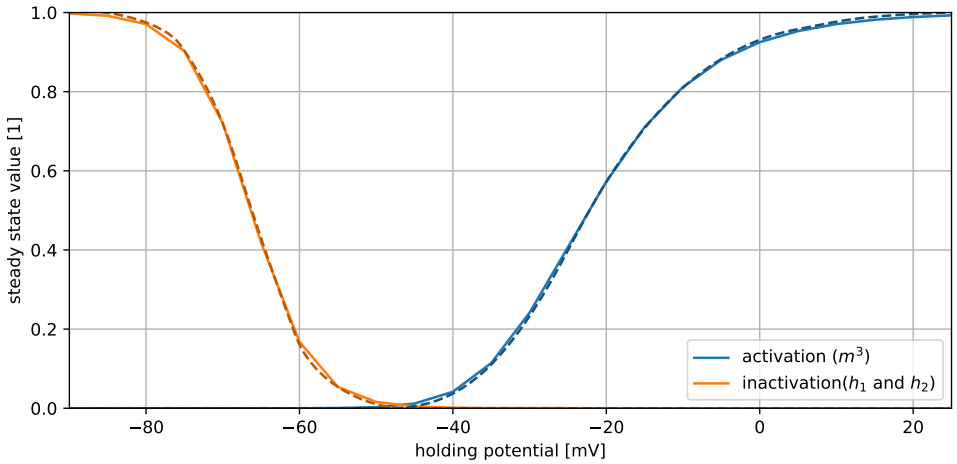


Figure 24. Steady state of gating variables for I_{Na}. Solid lines: Simulation result of SodiumChannelSteady. Dashed lines: Reference data extracted from Figure 2A in [4]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

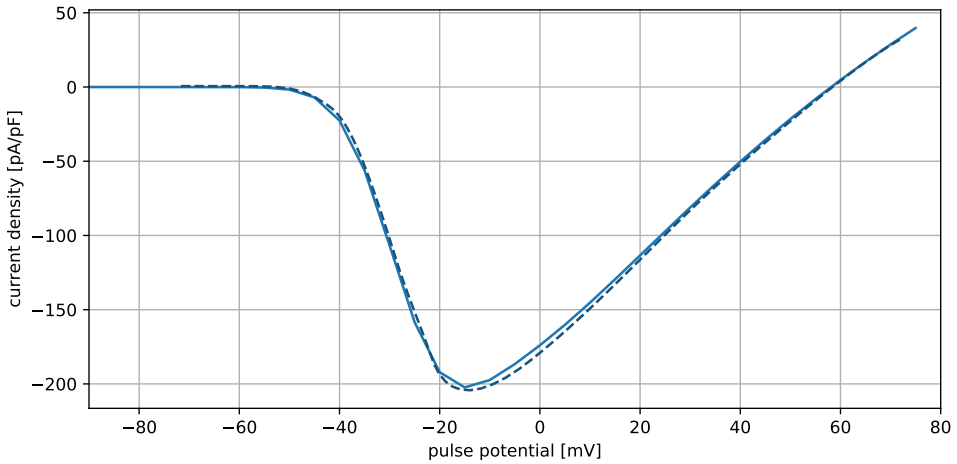


Figure 25. Current-voltage relationship of I_{Na} obtained with a voltage pulse protocol with a holding potential of -90 mV, a holding duration of 2 s, and a pulse duration of 50 ms. Solid line: Simulation result of SodiumChannelIV. Dashed line: Reference data extracted from Figure 2B in [4]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

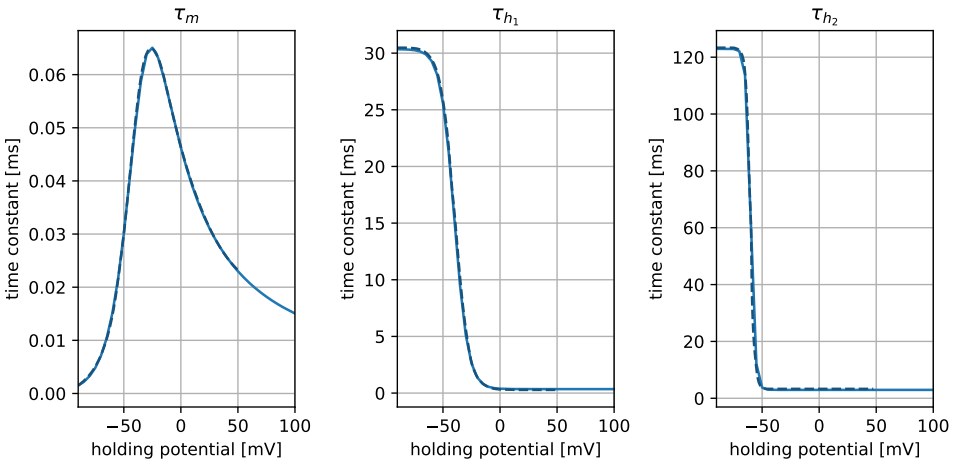


Figure 26. Time constants of gating variables for I_{Na}. Solid lines: Simulation result of SodiumChannelSteady. Dashed lines: Reference data extracted from Figures 2C–2E in [4]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

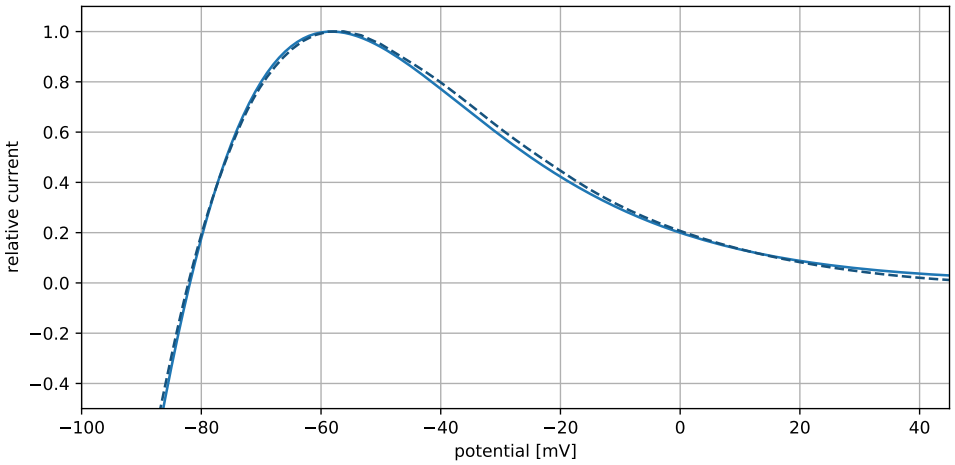


Figure 27. Current-voltage relationship of $I_{K,1}$ obtained by a voltage clamp experiment with linearly rising input voltage. Solid line: Simulation result of InwardRectifierLin. Dashed line: Reference data extracted from Figure 8 in [4]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

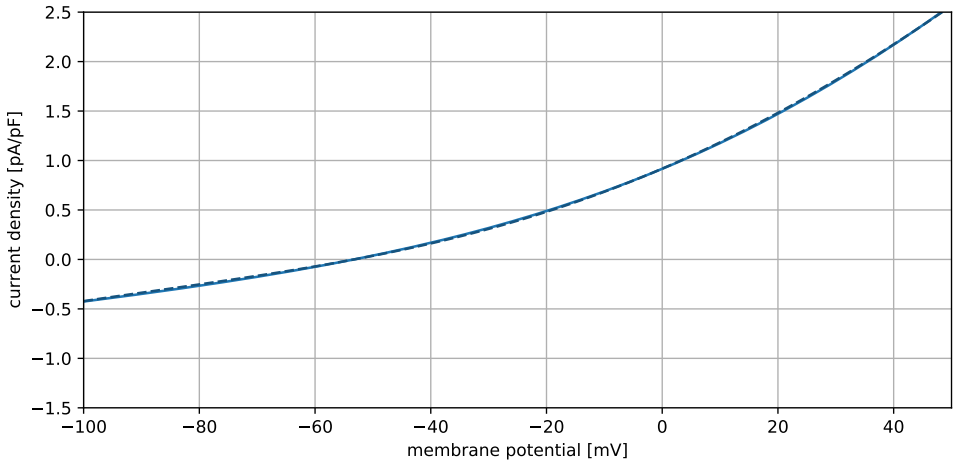


Figure 28. Current-voltage relationship of I_{NaCa} obtained from a voltage clamp experiment with a linearly rising input voltage, which uses parameter settings from Kurata *et al.* [2]. Solid line: Simulation result of SodiumCalciumExchangerLinKurata. Dashed line: Reference data extracted from Figure 17 (upper left) in [2]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

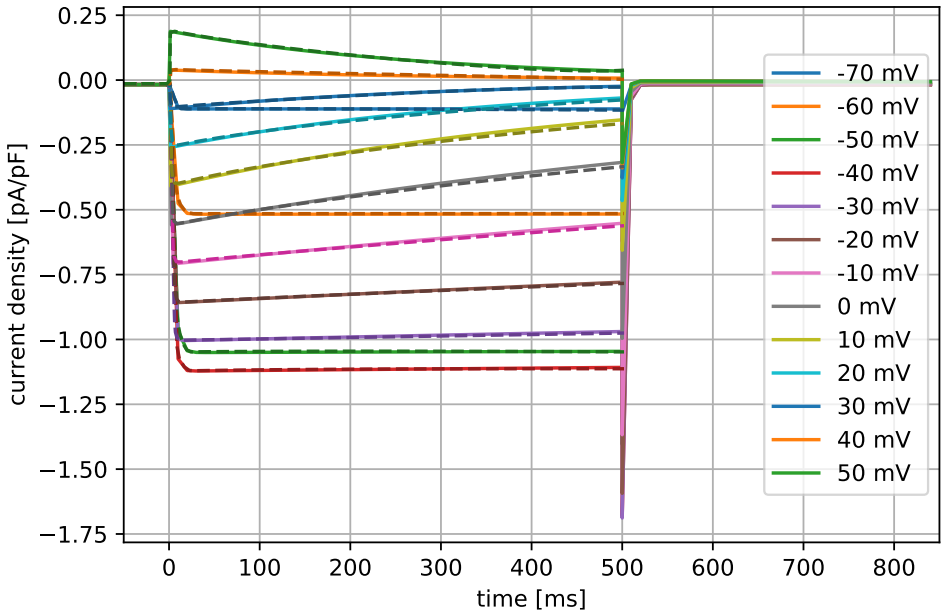


Figure 29. Current density time course of I_{st} due to a 500 ms stimulation to different voltages after holding the voltage at -80 mV for 15 s. This experiment uses parameter settings and steady state equation from Kurata *et al.* [2]. Solid line: Simulation result of SustainedInwardIVKurata. Dashed line: Reference data extracted from Figure 4 (bottom left) in [2]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

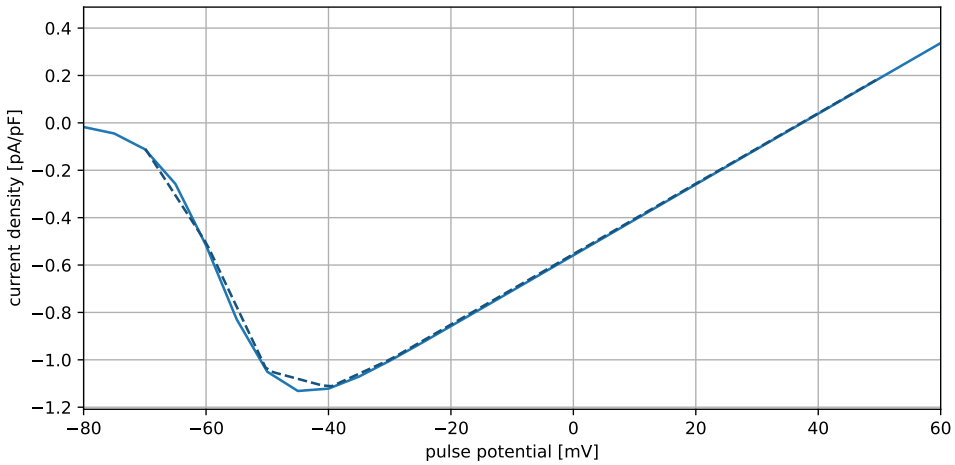


Figure 30. Current-voltage relationship of I_{st} obtained with a voltage pulse protocol with a holding potential of -80 mV, a holding duration of 15 s, and a pulse duration of 500 ms. This experiment uses parameter settings and steady state equation from Kurata *et al.* [2]. Solid line: Simulation result of SustainedInwardIV. Dashed line: Reference data extracted from Figure 4 (bottom right) in [2]. The plots are in perfect agreement. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

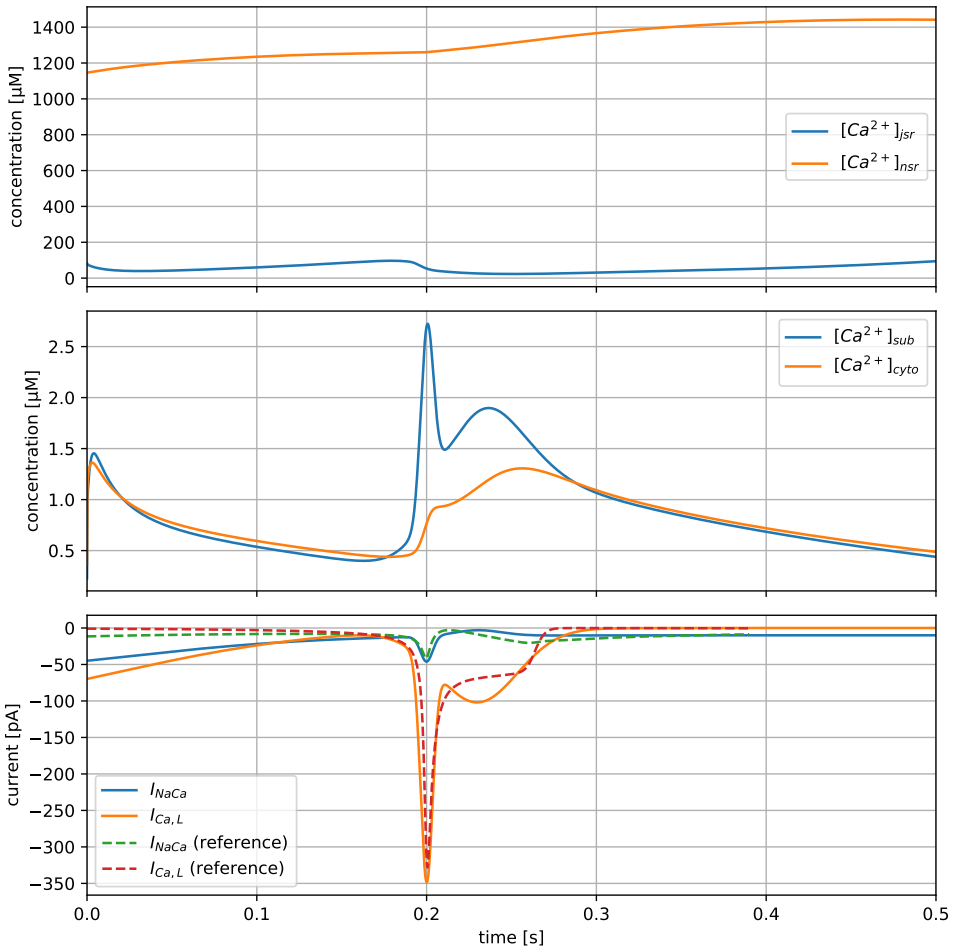


Figure 31. Time course of Ca^{2+} concentrations due to a dummy current for I_{NaCa} and $I_{Ca,L}$, which roughly resemble the true currents during an action potential (dotted lines in bottom plot). No reference plot is available for this experiment, but it allows to examine the $[Ca^{2+}]$ handling in isolation from other components. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

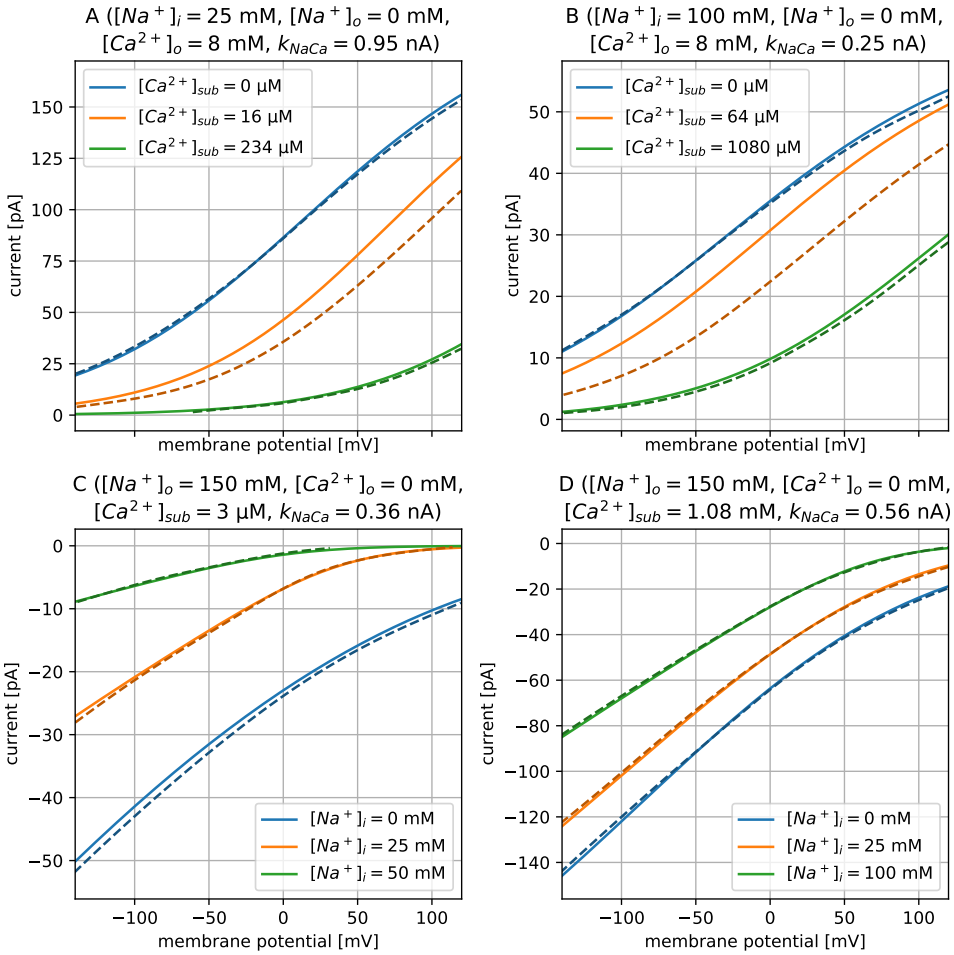


Figure 32. Current-voltage relationship of I_{NaCa} obtained from a voltage clamp experiment with linearly rising input voltage. Parts A and B show effect of variation in $\left[Ca^{2+}\right]_{sub}$ and parts C and D show effect of variation in $[Na^{+}]_i$. Solid lines: Simulation result of SodiumCalciumExchangerLinMatsuoka. Dashed lines: Reference data extracted from Figure 19 in [5]. Absolute current values are not shown in Figure 19, but can be found in Figures 15A (A), Figure 16B (B), and Figure 17 (C, D) in [5]. For part A and B, the lines with 0 and 16 µM (A) and 0 and 64 µM (B) are closer to each other, which does not agree with the reference, but does agree with the experimental data (filled and open circles). We have no good explanation for this difference, since it only occurs in two of the four experiment setups. The absolute values are not exact, since Matsuoka *et al.* used a different scaling factor for each part but do not report its value. We therefore choose the value of k_{NaCa} freely to roughly reproduce the absolute values in Figures 15–17 in [5]. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

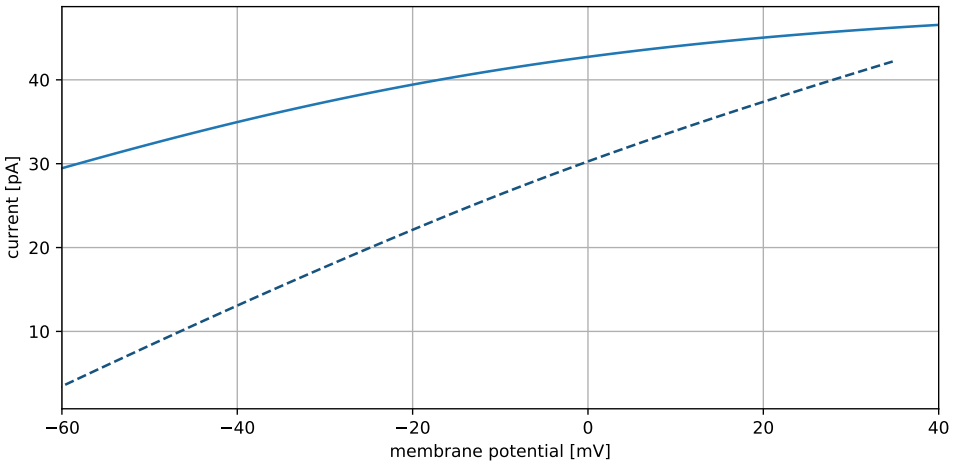


Figure 33. Current-voltage relationship of I_p obtained from a voltage clamp experiment with linearly rising input voltage. Solid line: Simulation result of SodiumPotassiumPumpLin. Dashed line: Reference data extracted from Figure 12 in [6]. The reference cannot be reproduced exactly as Demir *et al.* only report the sum of I_p and three different background currents. This figure was created with InaMo version 1.4.2, which is available under the DOI 10.5281/zenodo.4533008.

References

1. Inada S, Hancox JC, Zhang H, Boyett MR. One-Dimensional Mathematical Model of the Atrioventricular Node Including Atrio-Nodal, Nodal, and Nodal-His Cells. Biophysical Journal. 2009;97(8):2117–2127. doi:10.1016/j.bpj.2009.06.056.

2. Kurata Y, Hisatome I, Imanishi S, Shibamoto T. Dynamical Description of Sinoatrial Node Pacemaking: Improved Mathematical Model for Primary Pacemaker Cell. American Journal of Physiology-Heart and Circulatory Physiology. 2002;283(5):H2074–H2101. doi:10.1152/ajpheart.00900.2001.

3. Convery MK, Hancox JC. Na+-Ca2+ Exchange Current from Rabbit Isolated Atrioventricular Nodal and Ventricular Myocytes Compared Using Action Potential and Ramp Waveforms. Acta Physiologica Scandinavica. 2000;168(3):393–401. doi:10.1046/j.1365-201x.2000.00681.x.

4. Lindblad DS, Murphey CR, Clark JW, Giles WR. A Model of the Action Potential and Underlying Membrane Currents in a Rabbit Atrial Cell. American Journal of Physiology - Heart and Circulatory Physiology. 1996;271(4):H1666–H1696. doi:10.1152/ajpheart.1996.271.4.H1666.

5. Matsuoka S, Hilgemann DW. Steady-State and Dynamic Properties of Cardiac Sodium-Calcium Exchange: Ion and Voltage Dependencies of the Transport Cycle. Journal of General Physiology. 1992;100(6):963–1001. doi:10.1085/jgp.100.6.963.

6. Demir SS, Clark JW, Murphey CR, Giles WR. A Mathematical Model of a Rabbit Sinoatrial Node Cell. American Journal of Physiology-Cell Physiology. 1994;266(3):C832–C852. doi:10.1152/ajpcell.1994.266.3.C832.

2 Supplementary Tables

Model Part	Variable	Unit	Article	C++	CellML
Membrane	V	V	$-7.003 \cdot 10^{-2}$	$-7.022 \cdot 10^{-2}$	$-7.155 \cdot 10^{-2}$
I_{Na}	m	1	$1.227 \cdot 10^{-2}$	$1.246 \cdot 10^{-2}$	$1.048 \cdot 10^{-2}$
I_{Na}	h_1	1	$7.170 \cdot 10^{-1}$	$7.273 \cdot 10^{-1}$	$7.922 \cdot 10^{-1}$
I_{Na}	h_2	1	$6.162 \cdot 10^{-1}$	$6.378 \cdot 10^{-1}$	$7.883 \cdot 10^{-1}$
$I_{Ca,L}$	d_L	1	$4.069 \cdot 10^{-5}$	$3.952 \cdot 10^{-5}$	$3.229 \cdot 10^{-5}$
$I_{Ca,L}$	$f_{L,\text{fast}}$	1	$9.985 \cdot 10^{-1}$	$9.985 \cdot 10^{-1}$	$9.988 \cdot 10^{-1}$
$I_{Ca,L}$	$f_{L,\text{slow}}$	1	$9.875 \cdot 10^{-1}$	$9.911 \cdot 10^{-1}$	$9.988 \cdot 10^{-1}$
I_{to}	r	1	$8.857 \cdot 10^{-3}$	$8.704 \cdot 10^{-3}$	$8.029 \cdot 10^{-3}$
I_{to}	q_{fast}	1	$8.734 \cdot 10^{-1}$	$8.847 \cdot 10^{-1}$	$9.955 \cdot 10^{-1}$
I_{to}	q_{slow}	1	$1.503 \cdot 10^{-1}$	$2.026 \cdot 10^{-1}$	$5.480 \cdot 10^{-1}$
$I_{K,r}$	$p_{\text{a, fast}}$	1	$7.107 \cdot 10^{-2}$	$3.473 \cdot 10^{-2}$	$9.078 \cdot 10^{-4}$
$I_{K,r}$	$p_{\text{a, slow}}$	1	$4.840 \cdot 10^{-2}$	$3.473 \cdot 10^{-2}$	$2.899 \cdot 10^{-3}$
$I_{K,r}$	p_{i}	1	$9.866 \cdot 10^{-1}$	$9.868 \cdot 10^{-1}$	$9.879 \cdot 10^{-1}$
$[Ca^{2+}]$ handling	$[Ca^{2+}]_i$	mM	$1.206 \cdot 10^{-4}$	$1.082 \cdot 10^{-4}$	$3.104 \cdot 10^{-5}$
$[Ca^{2+}]$ handling	$[Ca^{2+}]_{sub}$	mM	$6.397 \cdot 10^{-5}$	$5.860 \cdot 10^{-5}$	$2.870 \cdot 10^{-5}$
$[Ca^{2+}]$ handling	$[Ca^{2+}]_{jsr}$	mM	$4.273 \cdot 10^{-1}$	$4.002 \cdot 10^{-1}$	$5.575 \cdot 10^{-1}$
$[Ca^{2+}]$ handling	$[Ca^{2+}]_{nsr}$	mM	$1.068 \cdot 10^{+0}$	$9.596 \cdot 10^{-1}$	$6.672 \cdot 10^{-1}$
$[Ca^{2+}]$ handling	f_{TC}	1	$2.359 \cdot 10^{-2}$	$2.120 \cdot 10^{-2}$	$6.156 \cdot 10^{-3}$
$[Ca^{2+}]$ handling	f_{TMC}	1	$3.667 \cdot 10^{-1}$	$3.395 \cdot 10^{-1}$	$1.122 \cdot 10^{-1}$
$[Ca^{2+}]$ handling	f_{TMM}	1	$5.594 \cdot 10^{-1}$	$5.834 \cdot 10^{-1}$	$7.843 \cdot 10^{-1}$
$[Ca^{2+}]$ handling	f_{CM_i}	1	$4.845 \cdot 10^{-2}$	$4.366 \cdot 10^{-2}$	$1.290 \cdot 10^{-2}$
$[Ca^{2+}]$ handling	f_{CM_s}	1	$2.626 \cdot 10^{-2}$	$2.410 \cdot 10^{-2}$	$1.192 \cdot 10^{-2}$
$[Ca^{2+}]$ handling	f_{CQ}	1	$3.379 \cdot 10^{-1}$	$3.235 \cdot 10^{-1}$	$4.007 \cdot 10^{-1}$
$[Ca^{2+}]$ handling	f_{CSL}	1	$3.936 \cdot 10^{-5}$	$3.085 \cdot 10^{-5}$	$8.911 \cdot 10^{-6}$

Table 1. Initial values for AN cell model in article, C++-code, and CellML code.

Model Part	Variable	Unit	Article	C++	CellML
Membrane	V	V	$-6.213 \cdot 10^{-2}$	$-6.213 \cdot 10^{-2}$	$-4.971 \cdot 10^{-2}$
$I_{Ca,L}$	d_L	1	$1.533 \cdot 10^{-4}$	$1.534 \cdot 10^{-4}$	$1.793 \cdot 10^{-3}$
$I_{Ca,L}$	$f_{L,\text{fast}}$	1	$6.861 \cdot 10^{-1}$	$6.809 \cdot 10^{-1}$	$9.756 \cdot 10^{-1}$
$I_{Ca,L}$	$f_{L,\text{slow}}$	1	$4.441 \cdot 10^{-1}$	$3.320 \cdot 10^{-1}$	$7.744 \cdot 10^{-1}$
$I_{K,r}$	$p_{\text{a, fast}}$	1	$6.067 \cdot 10^{-1}$	$6.061 \cdot 10^{-1}$	$1.925 \cdot 10^{-1}$
$I_{K,r}$	$p_{\text{a, slow}}$	1	$1.287 \cdot 10^{-1}$	$1.288 \cdot 10^{-1}$	$7.972 \cdot 10^{-2}$
$I_{K,r}$	p_{i}	1	$9.775 \cdot 10^{-1}$	$9.775 \cdot 10^{-1}$	$9.490 \cdot 10^{-1}$
I_f	y	1	$3.825 \cdot 10^{-2}$	$3.823 \cdot 10^{-2}$	$4.623 \cdot 10^{-2}$
I_{st}	q_a	1	$1.933 \cdot 10^{-1}$	$1.936 \cdot 10^{-1}$	$4.764 \cdot 10^{-1}$
I_{st}	q_i	1	$4.886 \cdot 10^{-1}$	$4.885 \cdot 10^{-1}$	$5.423 \cdot 10^{-1}$
$[Ca^{2+}]$ handling	$[Ca^{2+}]_i$	mM	$3.623 \cdot 10^{-4}$	$3.633 \cdot 10^{-4}$	$1.850 \cdot 10^{-4}$
$[Ca^{2+}]$ handling	$[Ca^{2+}]_{sub}$	mM	$2.294 \cdot 10^{-4}$	$2.300 \cdot 10^{-4}$	$1.603 \cdot 10^{-4}$
$[Ca^{2+}]$ handling	$[Ca^{2+}]_{jsr}$	mM	$8.227 \cdot 10^{-2}$	$8.185 \cdot 10^{-2}$	$2.963 \cdot 10^{-1}$
$[Ca^{2+}]$ handling	$[Ca^{2+}]_{nsr}$	mM	$1.146 \cdot 10^{+0}$	$1.146 \cdot 10^{+0}$	$1.111 \cdot 10^{+0}$
$[Ca^{2+}]$ handling	f_{TC}	1	$6.838 \cdot 10^{-1}$	$6.856 \cdot 10^{-2}$	$3.565 \cdot 10^{-2}$
$[Ca^{2+}]$ handling	f_{TMC}	1	$6.192 \cdot 10^{-1}$	$6.195 \cdot 10^{-1}$	$4.433 \cdot 10^{-1}$
$[Ca^{2+}]$ handling	f_{TMM}	1	$3.363 \cdot 10^{-1}$	$3.360 \cdot 10^{-1}$	$3.917 \cdot 10^{-1}$
$[Ca^{2+}]$ handling	f_{CM_i}	1	$1.336 \cdot 10^{-1}$	$1.339 \cdot 10^{-1}$	$7.230 \cdot 10^{-2}$
$[Ca^{2+}]$ handling	f_{CM_s}	1	$8.894 \cdot 10^{-2}$	$8.915 \cdot 10^{-2}$	$6.308 \cdot 10^{-2}$
$[Ca^{2+}]$ handling	f_{CQ}	1	$8.736 \cdot 10^{-2}$	$8.694 \cdot 10^{-2}$	$2.614 \cdot 10^{-1}$
$[Ca^{2+}]$ handling	f_{CSL}	1	$4.764 \cdot 10^{-5}$	$4.674 \cdot 10^{-5}$	$4.150 \cdot 10^{-5}$

Table 2. Initial values for N cell model in article, C++ code, and CellML code.

Model Part	Variable	Unit	Article	C++	CellML
Membrane	V	V	$-6.863 \cdot 10^{-2}$	$-6.867 \cdot 10^{-2}$	$-6.976 \cdot 10^{-2}$
I_{Na}	m	1	$1.529 \cdot 10^{-2}$	$1.521 \cdot 10^{-2}$	$1.322 \cdot 10^{-2}$
I_{Na}	h_1	1	$6.438 \cdot 10^{-1}$	$6.463 \cdot 10^{-1}$	$7.066 \cdot 10^{-1}$
I_{Na}	h_2	1	$5.552 \cdot 10^{-1}$	$5.638 \cdot 10^{-1}$	$7.016 \cdot 10^{-1}$
$I_{Ca,L}$	d_L	1	$5.025 \cdot 10^{-5}$	$4.996 \cdot 10^{-5}$	$4.235 \cdot 10^{-5}$
$I_{Ca,L}$	$f_{L,\text{fast}}$	1	$9.981 \cdot 10^{-1}$	$9.981 \cdot 10^{-1}$	$9.984 \cdot 10^{-1}$
$I_{Ca,L}$	$f_{L,\text{slow}}$	1	$9.831 \cdot 10^{-1}$	$9.851 \cdot 10^{-1}$	$9.984 \cdot 10^{-1}$
I_{to}	r	1	$9.581 \cdot 10^{-3}$	$9.559 \cdot 10^{-3}$	$8.948 \cdot 10^{-3}$
I_{to}	q_{fast}	1	$8.640 \cdot 10^{-1}$	$8.708 \cdot 10^{-1}$	$9.948 \cdot 10^{-1}$
I_{to}	q_{slow}	1	$1.297 \cdot 10^{-1}$	$1.343 \cdot 10^{-1}$	$4.274 \cdot 10^{-1}$
$I_{K,r}$	$p_{\text{a, fast}}$	1	$9.949 \cdot 10^{-2}$	$9.333 \cdot 10^{-2}$	$1.418 \cdot 10^{-3}$
$I_{K,r}$	$p_{\text{a, slow}}$	1	$7.024 \cdot 10^{-2}$	$6.769 \cdot 10^{-2}$	$5.398 \cdot 10^{-3}$
$I_{K,r}$	p_{i}	1	$9.853 \cdot 10^{-1}$	$9.854 \cdot 10^{-1}$	$9.864 \cdot 10^{-1}$
$\left[Ca^{2+}\right]$ handling	$\left[Ca^{2+}\right]_i$	mM	$1.386 \cdot 10^{-4}$	$1.340 \cdot 10^{-4}$	$3.733 \cdot 10^{-5}$
$\left[Ca^{2+}\right]$ handling	$\left[Ca^{2+}\right]_{sub}$	mM	$7.314 \cdot 10^{-5}$	$7.122 \cdot 10^{-5}$	$3.273 \cdot 10^{-5}$
$\left[Ca^{2+}\right]$ handling	$\left[Ca^{2+}\right]_{jsr}$	mM	$4.438 \cdot 10^{-1}$	$4.475 \cdot 10^{-1}$	$6.822 \cdot 10^{-1}$
$\left[Ca^{2+}\right]$ handling	$\left[Ca^{2+}\right]_{nsr}$	mM	$1.187 \cdot 10^{+0}$	$1.163 \cdot 10^{+0}$	$8.187 \cdot 10^{-1}$
$\left[Ca^{2+}\right]$ handling	f_{TC}	1	$2.703 \cdot 10^{-2}$	$2.615 \cdot 10^{-2}$	$7.396 \cdot 10^{-3}$
$\left[Ca^{2+}\right]$ handling	f_{TMC}	1	$4.020 \cdot 10^{-1}$	$3.930 \cdot 10^{-1}$	$1.337 \cdot 10^{-1}$
$\left[Ca^{2+}\right]$ handling	f_{TMM}	1	$5.282 \cdot 10^{-1}$	$5.362 \cdot 10^{-1}$	$7.652 \cdot 10^{-1}$
$\left[Ca^{2+}\right]$ handling	f_{CM_i}	1	$5.530 \cdot 10^{-2}$	$5.355 \cdot 10^{-2}$	$1.547 \cdot 10^{-2}$
$\left[Ca^{2+}\right]$ handling	f_{CM_s}	1	$2.992 \cdot 10^{-2}$	$2.911 \cdot 10^{-2}$	$1.358 \cdot 10^{-2}$
$\left[Ca^{2+}\right]$ handling	f_{CQ}	1	$3.463 \cdot 10^{-1}$	$3.483 \cdot 10^{-1}$	$4.500 \cdot 10^{-1}$
$\left[Ca^{2+}\right]$ handling	f_{CSL}	1	$4.843 \cdot 10^{-5}$	$4.447 \cdot 10^{-5}$	$1.217 \cdot 10^{-5}$

Table 3. Initial values for NH cell model in article, C++-code, and CellML code.