The impact of mathematical modeling languages on model quality in systems biology: A software engineering perspective

Christopher Schölzel[1,*], Valeria Blesius[1], Gernot Ernst[2,3], and Andreas Dominik[1]

[1]THM University of Applied Sciences, Giessen, Germany,
{christopher.schoelzel,andreas.dominik}@mni.thm.de
[2]Vestre Viken Hospital Trust, Kongsberg, Norway
[3]University of Oslo, Norway
[*]corresponding author

April 8, 2020

Abstract

Reproducible, understandable models that can be reused and combined to true multi-scale systems are required to solve the present and future challenges of systems biology. However, many mathematical models are still built for a single purpose and reusing them in a different context is challenging. To overcome these challenges model quality needs to be addressed at the (software-)engineering level. Instead of just declaring standard modeling languages, researchers need to be aware of the characteristics that make these languages desirable and they need to utilize them consistently. We therefore propose a list of desirable language characteristics and provide guidelines how to incorporate them in a model: In our opinion, a mathematical modeling language used in systems biology should be modular, human-readable, hybrid (i.e. support multiple formalisms), open, declarative, and allow to represent models graphically. We demonstrate the benefits of these characteristics by translating a monolithic model of the human cardiac conduction system to a modular version and extending it with a trigger for premature ventricular contractions. For this task we use the modeling language Modelica as an example, that has all the aforementioned characteristics, but is not well known in systems biology. Our experiment illustrates how each characteristic can have a substantial effect on the quality and reusability of the resulting model. When applied consistently, they facilitate and simplify the creation and especially the extension of the modular model. We therefore recommend to consider these guidelines when choosing a programming language for any biological modeling task.

1 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 that 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 that 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 independently designed models. These models 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 submodels 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, 7]. As an extreme example, Topalidou *et al.* [8] 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) of the human baroreflex to the modeling language Modelica [9, 10]. 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. However, when we wanted to extend the model with a trigger for premature ventricular contraction (PVC), it turned out that even the Modelica 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.

From this example, it becomes apparent 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.* [8], since the original model was implemented in Delphi, which is also an imperative language that is not well suited for mathematical modeling.

Even though the need for guidelines on the engineering level is apparent, most publications about model quality and 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 [11–16], b) high-level choices of modeling formalisms and techniques [17–19], or c) model documentation, annotation and distribution [7, 20, 21]. When modeling languages are discussed, it is in the form of stating accepted standard languages. Both COmputational Modeling in BIology NEtwork (COMBINE) [20] and Minimal Information Required In the Annotation of Models (MIRIAM) [22] suggest to use CellML and Systems Biology Markup Language (SBML), but neither go into detail which characteristics make these languages more desirable than other choices. Our previous example of the translation of the SHM shows that using a suitable language is a necessary but not sufficient criterion for the model to actually be reusable. Additionally these languages cannot cover all use cases, especially for multi-class models that combine entirely different model formalisms that may not even be representable in a single language [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 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 towards a specific level of organization. MATLAB is a proprietary domain-specific programming language designed for scientific computing in general that 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`). While the use of MATLAB is slowly declining, the open-source programming language Python gains increasing interest in the community. Usually models are not built in Python itself, but researchers have created packages such as PySB [23] and python simulator for cellular systems (PySCes) [24] that define embedded domain-specific languages (DSLs) which facilitate the creation of mathematical models for specific use cases. With Tellurium [25] there also exists a broader python-based environment that supports multiple COMBINE standards and uses the declarative modeling language Antimony [26]. 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 [27]. Finally, Modelica is an open-source declarative modeling language primarily used in engineering [28]. It has a large user base both in industry and research, but is still largely unknown in systems biology. Notable exception include the Physiolibrary [29], a Modelica library for physiological models and SBML2Modelica [30], a tool that translates SBML models to Modelica. This extensive list of language candidates makes it apparent that researches will always 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 [31–33]. Hellerstein *et al.* [31] 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 paper we expand on the idea of model engineering in two 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 how these characteristics can be exploited to increase the code quality and thus reproducibility and reusability of a particular model.

To demonstrate the reasoning behind those characteristics and their benefits in a concrete example we use the language Modelica. Because it is an industrial standard and still little known in systems biology, we think that Modelica can help to stress the engineering aspect of the discussion and provides a fresh perspective on existing challenges. We find that Modelica is the best language to demonstrate our reasoning for our specific example, but this does not mean that Modelica is superior to other alternatives for all purposes. A different situation might warrant a different choice and with this paper we want to give our readers additional arguments to make this choice well informed. A detailed comparison of the strengths and weaknesses of Modelica and available alternatives can be found in the supplement.

To give an application example of our guidelines we transform the aforementioned monolithic model of the human cardiac conduction system into a modular structure and show that this version can easily be extended by a trigger for PVCs. Finally, this example is utilized to reflect on the choice of language characteristics and the impact that a modeling language can have on the usefulness of the model.

2 Results

2.1 Desirable characteristics for a mathematical modeling language for systems biology

The following characteristics were developed from our personal experience with Modelica and the SHM and/or from literature review. Each characteristic will be introduced with arguments for its usefulness, a brief set of guidelines how it may be applied to full effect and references to other authors that advocate this feature.

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. Modularization (and information hiding) 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. One could even make the case that modularization is the *only* way to handle this complexity since monolithic models will become unmaintainable quite fast. Even the Guyton model, which is probably among the most complicated monolithic models in systems biology, tries to explain the circuits by grouping them in (unfortunately tightly coupled) modules [34]. 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. Some languages also allow to reuse models across different languages, tools and platforms, which is especially interesting for multi-class models. One prominent example of this is the Functional Mock-up Interface (FMI), a model exchange format maintained by the Modelica Association [35, 36].

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 interfaces that explicitly state possible connection points to the outside world. If possible, each module should be tested individually (this is called a "unit test" in software engineering).

References: There is a general consensus that multi-scale modeling requires some form of modularity for hierarchical composition [33, 37–41]. More specifically, Hellerstein *et al.* [31] and Mulugeta *et al.* [32] both suggest that object-oriented programming might be an especially promising way to implement modularity.

(Human-)readable Models may not be reused or reproduced in the same language in which they are originally written. Therefore the code should be expressive enough that one can extract the relevant mathematical definitions without detailed knowledge about the tools that are associated with the modeling language. This will also make it more likely that a reader can spot errors that can be corrected before or after publication and allow to easily keep the model files under version control. Readable code is largely the responsibility of its author, but a language may facilitate a clean coding style by providing expressive language constructs and documentation features or hinder it by introducing visual clutter. For example, this is a disadvantage of purely XML-based markup languages, as they are typically very hard to understand in text form, requiring a researcher that is not familiar with the language to first install an appropriate tool before inspecting a model. On the other hand, in the case of widely accepted languages like SBML this may not be such a big issue since it can be expected that most researchers in the field will already use such a tool.

Guidelines: Variables and parameters should have speaking names instead of single-letter identifiers. They should also be documented with a short sentence that explains their meaning. The number of variables per equation should be kept low, complex formulas should be split up. Optimizations and workarounds that are not intuitive should be documented in the code. 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.

References: As mentioned in the introduction, Hellerstein *et al.* [31] also advocate for speaking variable names.

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 ODEs and discrete events, but other combinations are possible. 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 coupling of models across languages with standardized interfaces such as the FMI [35, 36]. Additionally, there is a trade-off between fully supporting a modeling formalism, such as ODEs 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 well beyond the biochemical level.

Guidelines: A model should clearly indicate which variables are discrete and which are continuous. Event triggers that define the transition between discrete and continuous parts of the model should be examined and tested with extra care.

References: In their 2017 review Bardini *et al.* [2] argue that multi-scale models in systems biology in general should strive towards a hybrid approach.

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 quality. 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 and suggest corrections.

References: Many large projects and databases such as Physiome [16], the virtual liver network [42], Plants *in silico* [37] and SEEK [21] already provide open-source implementations of models. Mulugeta *et al.* [32] also specifically advocate for more version control and changelogs (in the form of e-notebooks).

Declarative The mathematical formalism for biological models is complicated enough. 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. As a consequence, errors reported by the compiler can also focus on meaning rather than just grammar, increasing the soundness of the model. For example, the language should allow the declaration and automatic checking of proper units for variables and parameters. 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 to connect model parts to standardized ontological terms. For the latter it is preferable if the support for ontologies is already included in the language itself, which is usually not the case for general mathematical modeling languages such as MATLAB, Julia packages or Modelica.

Guidelines: Models should follow strict mathematical rules to fit nicely into the chosen formalism. If a model needs workarounds, 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.

References: Few researchers in systems biology make the distinction between imperative and declarative languages explicitly. Loew & Schaff [43] mention that the language that they chose for the Virtual Cell environment is declarative, but do not go into detail why this is important. Zhu *et al.* [44] state that declarative languages are desirable, because it allows to describe the biological processes "in a natural way".

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 abstract structural diagram definitions like Unified Modeling Language (UML) and Systems Biology Graphical Notation (SBGN) [45] or automatically generated graphs of variable interactions and actual drawings of the biological interactions. Abstract diagrams are an exact representation of the model and can sometimes be generated automatically or with little effort, but they tend to quickly become confusing with increasing model size. On the other hand, drawings are less accurate and require manual effort, but they can capture the essence of the information required to understand the model. The modeling language should provide a way to describe the model with at least one form of diagrams. Preferably the diagram should be directly coupled to the model equations to ensure that the graphical documentation always stays up to date when the code changes, but it should also be possible to at least manually position elements for increased clarity.

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.

References: The Physiolibrary is a Modelica library for physiological models that has graphical representations for each component [29]. ProMoT is a modeling tool that allows to compose modular models in a graphical way [39]. Alves *et al.* [46] compare 12 different simulator tools, giving higher ratings to those that have graphical representations for model components.

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.

2.2 Modularizing a model of the human cardiac conduction system increases understandability

The Seidel-Herzel model (SHM) describes the autonomic control of the heart rate in humans at a high level of abstraction [9]. It can be classified as a hybrid (discrete and continuous), deterministic, quantitative, macro-level model. As such, it is not representative of lower-level metabolism and cell-signaling models that are currently the most common type encountered in systems biology, but it is well suited to showcase what is needed for future multi-scale models that inevitably have to leave these well-explored levels behind to generate new insights. Henrik Seidel developed and implemented the SHM in his PhD Thesis using the programming language C. In a previous paper we translated it to Modelica [10]. All effects in the model are described on the organ level, including a blood pressure curve 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[1] 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. In the original model, these effects where tightly coupled within a single piece of code comprising five parameters, 13 variables, and 12 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.

The modular version separates the code into three components `RefractoryGate` , `Pacemaker` and `AVConductionDelay`. These components are connected via an unifying interface using a base class `UnidirectionalConductionComponent` that 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 where a signal is issued (i.e., they behave as Kronecker deltas). For the `RefractoryGate`, the output equals the input except that after each signal there is a time period `T_refrac` in which incoming signals are ignored. The `Pacemaker` 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 `T`. The `AVConductionDelay` delays an incoming signal by a duration that depends on the elapsed time since the last output signal has been issued. To give an example of the Modelica syntax, the resulting code for `RefractoryGate` looks as follows:

```
model RefractoryGate
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Gate;
```

[1]Seidel probably meant to include the refractory behavior of the ventricles and not the SA node. The actual implementation, however, checks the refractory state *before* the delay between SA node and ventricles is applied.

```
    parameter Modelica.SIunits.Time t_first = 0;
    parameter Modelica.SIunits.Period T_refrac = 1;
    Boolean refrac_passed = time - pre(t_last) > T_refrac;
  protected
    Real t_last(start=t_first, fixed=true);
  equation
    outp = inp and refrac_passed;
    when outp then
      t_last = time;
    end when;
  end RefractoryGate;
```

To explain the interaction between the described models, a graphical representation is introduced for each component. This diagram is not a separate image file, but directly implemented in Modelica, using **annotation()** statements that define, place and connect model icons as vector graphics. To keep the actual model code simple and short we defined these annotations in separate icon classes whose code can be found in Supplementary Listing 23–27. Models can extend these classes to inherit the icon definition, e.g. by the statement **extends** `SHMConduction.Icons.Gate;` in the above listing. As seen in Figure 1 we chose an open garden 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. This is defined in a composite model `ModularConduction` that looks as follows:

```
model ModularConduction
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Heart;
  RefractoryGate refrac_av(T_refrac=0.364);
  Pacemaker pace_av(T=1.7);
  AVConductionDelay delay_sa_v;
  discrete Modelica.SIunits.Period T(start=initial_T, fixed=true);
  discrete Modelica.SIunits.Time cont_last(start=0, fixed=true);
equation
  connect(inp, pace_av.inp);
  connect(pace_av.outp, refrac_av.inp);
  connect(refrac_av.outp, pace_av.reset);
  connect(refrac_av.outp, delay_sa_v.inp);
  connect(delay_sa_v.outp, outp);
  when outp then
    T = time - pre(cont_last);
    cont_last = time;
  end when;
end ModularConduction;
```

Here, components are used by defining variables of the types `RefractoryGate`, `Pacemaker`, and `AVConductionDelay` whose inputs and outputs are connected via **connect()** equations. Both the variables and **connect()** equations can have **annotation()** statements that define the placement of the components and the lines drawn to represent the connections in the diagram as seen in Supplementary Listing 16.

The structure defined in this model (and seen in 1) 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, we will show in the following sections that this simplified structure closely replicates the behavior of the SHM.

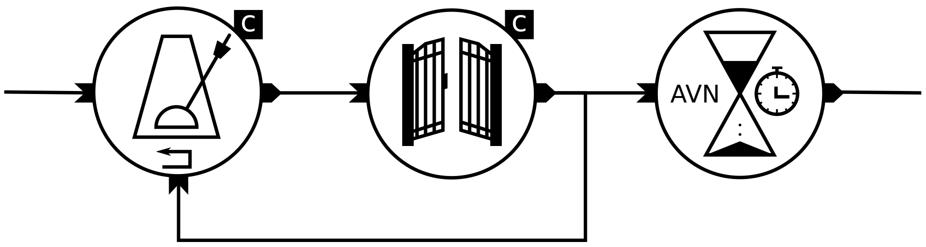

Figure 1: Diagram of the modular conduction model with symbols for the components. From left to right: `Pacemaker` for the pacemaker effect of the AV node, `RefractoryGate` for the refractory behavior of the AV node and `AVConductionDelay` for the combined delay between the SA node and the ventricles. 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.

2.3 The modular model behaves similar to the original version

Although the modular version of the conduction model 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 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.

Figure 2 shows a comparison of the resulting RR values (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 you can see a difference in the plots. In these areas the RR values of the modular model fluctuate with a higher amplitude and lower frequency compared to the original SHM.

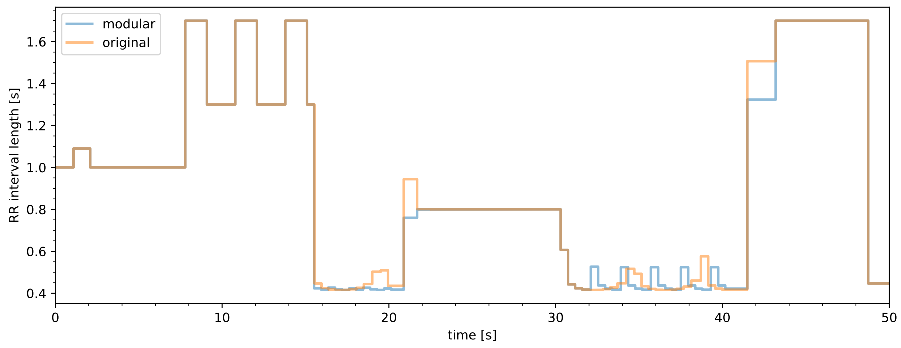

Figure 2: Comparison of the RR values of the original conduction model (orange) and the modular version (blue). The plot was obtained with an artificial sinus signal that switches its cycle duration every ten seconds according to the following schedule: 1 s, 3 s, 0.05 s, 0.8 s, 0.2 s, 1.8 s. This schedule 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.

2.4 Extending the modular model with a trigger for premature ventricular contractions is easy

To show that the modular structure and the use of a MoDROGH language facilitate reuse and extension we added a trigger for premature ventricular contractions (PVCs) to the model. 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. [47].

An ectopic beat in the ventricles leads to a stimulation of the AV node that travels back upwards to the 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 3.

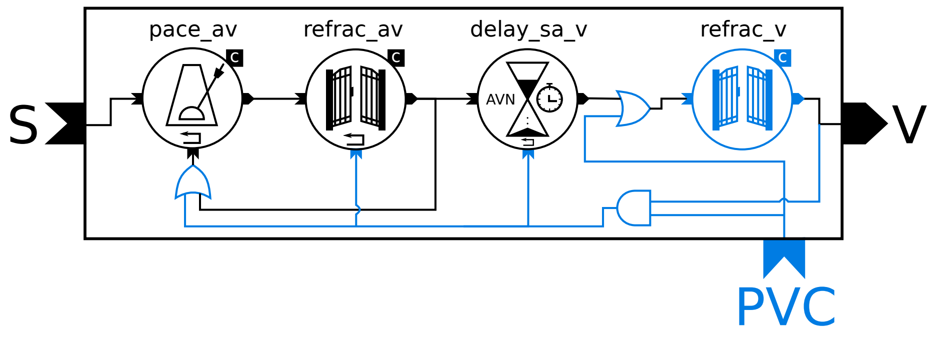

Figure 3: Diagram of the PVC model. The components are the same as in Figure 1 with additional components and connections highlighted in blue: reset inputs, second `RefractoryGate` (right) for the refractory period of the ventricles, two logical OR gates and one AND gate. The letters on the outside of 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).

2.5 The PVC extension shows plausible results

The behavior of the resulting model is shown in Figure 4. 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 RR 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 RR 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

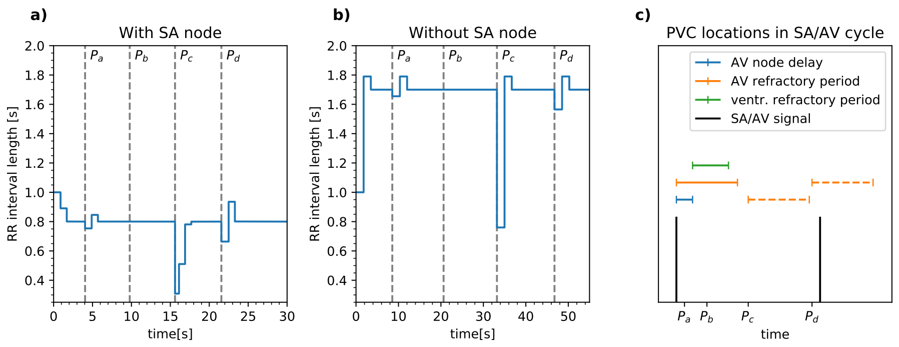

Figure 4: RR intervals of the modular model with PVCs with a) a normal sinus rhythm of 75 bpm and b) without sinus signal. Part c) shows the location of the ectopic beats in the normal cycle duration. PVCs are triggered with different prematurity: P_a while a signal is delayed, P_b within the ventricular refractory period, P_c just between two beats, P_d just before a beat would be triggered by the SA or AV node.

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 RR 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.

3 Discussion

The separation of the model of the human cardiac conduction system into individual components with a clear interface required structural changes. Therefore, before discussing the impact of our language and design choices on the understandability and reproducibility of the model, possible implications for its accuracy need to be addressed.

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 RR 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 the results, 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 `T_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 [48]. It can therefore be said that the modular design both helps identifying plausibility issues and can be more easily adapted to the biological reality.

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 RR intervals that are shorter than the sinus cycle length. The third RR interval is also a little bit 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 RR 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.

With the question of the validity of the example models out of the way, the focus can now return 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.

Modular We have shown that extending the modular model to simulate PVCs was quite simple. Using the component diagram as a basis, the discussion about where to include the trigger signal for an ectopic beat became much easier than with the tightly coupled model. When we originally tried to implement the same extension 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. Those were again followed by technological questions, since at the end some variables had to be changed and introduced, but due to the separation in small components each change only required the review of a few lines of code. In fact, the code for the `RefractoryGate` could be reused by simply adding another instance of this component to represent the refractory period of the ventricles. In the original model, several variables and equations would have to be added, making the already complicated system almost unmanageable.

This becomes even more apparent if we consider a bidirectional conduction model, which is probably only feasible using a modular structure.

Extending each component with an additional input and output in the opposite direction is very possible, if one only has to consider one effect at a time. The diagram view would even require less changes, mainly doubling each connection line between two subsequent components. Doing the same for the original model in C or even the non-modular version in Modelica would increase the amount of variables and equations substantially and, therefore, further diminish the already low readability and instead introduce multiple possible sources for errors that would be hard to track down.

It has to be said that the resulting components are only reusable within their physiological context. It is, for example, not possible to apply any insights from this model to a model of the liver or the lung. This can only be the case for areas in biology that are well enough understood to form a unifying theory, such as cell metabolism.

(Human-)readable The code of the models presented in this paper needed little additional words for explanation. We think that readers that are not familiar with Modelica will also be able to understand the general idea, since the code directly represents meaningful concepts with little additional clutter. The same could probably not be said about a piece of C-code or an XML file generated by an SBML tool. The full version of the model code that can be found in Supplementary Listing 1–27 and at `https://github.com/CSchoel/shm-conduction` also contains additional documentation. In Modelica, documentation strings can be attached to models, variables, parameters, and even individual equations. Models can also have a more rich documentation in the form of an embedded HTML string. Graphical tools can interpret this information to, for example, automatically generate a user-friendly form for changing parameter values. This documentation style is possible with most declarative languages and should be preferred over unstructured comments, which help when reading the raw code but usually cannot be further processed by tools.

Hybrid The example model is (almost) purely discrete. This does not allow us to showcase the hybrid nature of Modelica directly. However, since this model of the cardiac conduction system has been developed as a replacement of the original, it can be integrated into the full SHM, which also has continuous variables, by simply changing the name of the referenced submodel from `MonolithicConduction` to `ModularConduction`. We could also show that although Modelica is not purely built for discrete modeling, formulation of discrete events requires little effort. You only have to be careful to use `pre()` when you are referencing the value of a variable *before* an event.

Open Since Modelica is an open-source language with open-source tools, the reader can try out the models discussed here at `https://github.com/CSchoel/shm-conduction`. You simply have to download the latest release from the Github repository, download and install OpenModelica and load the models using the "Load library" option in the "File" menu. This is possible whether you are using Windows, Linux or macOS as your operating system. With JModelica there also exists a second free and platform-independent compiler that offers optimization features [49]. If you want to combine Modelica models with models written in different languages, this may be possible using the FMI specification [35, 36].

However, engineers that use Modelica for industrial applications mostly turn towards 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 the aforementioned open-source solutions. In cases where this is a concern, languages with fully open environments like SBML, CellML or the Python-based solutions PySB, PySCes or Tellurium should be preferred.

Declarative The separation of models into components with clear interfaces enhances the benefits of a declarative language. Subjecting the component interfaces to strict mathematical rules makes existing design flaws apparent. The original model contained some design choices that were probably taken because they were convenient for programming, but they showed to be harmful for understanding and maybe even for the physiological plausibility of the model. For example, the original model mixed variables that represent actual signals and time stamp variables that schedule signals for the future (see Supplementary Figure 1). An incoming sinus signal was first translated to a scheduled contraction time which then would only take effect if there was not already a contraction that was scheduled to happen before the current signal. Upon the actual contraction, a scheduled time for a contraction triggered by the AV node was calculated. 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 (unless in the case of an ectopic beat). 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 where missing. A correct implementation was easier when the different effects where clearly separated.

An additional advantage of the declarative approach is the support for multiple different numerical solvers for the model. The original version by Seidel included a hard-coded implementation of a fourth order Runge-Kutta solver, but in OpenModelica the user can simply select an appropriate solver from a drop-down menu.

Regarding the support for ontologies and terminologies, proper SI units could be introduced for each variable and the Kronecker delta-behavior of in- and outputs could be made explicit by defining the type `InstantSignal`.

It would have been possible to add full support for an ontology such as Chemical Entities of Biological Interest (ChEBI) [50] or Ontology of Physics for Biology (OPB) [51] in Modelica in two ways: First, the ontological terms could be translated to a corresponding type hierarchy, and second, the ontological terms could be assigned to variables with so called vendor-specific annotations of the form `annotation(__VendorName(key =value))`. Both approaches require manual effort and do not guarantee interoperability with other Models defined in Modelica. Here, languages with a biological focus such as SBML, or CellML have a clear advantage with tools like SemGen, a semantics-based annotation and composition software for SBML and CellML models [52].

Graphical The diagrams in Figure 1 and 3 not only help to understand the model at first glance, but they can also be defined as part of the Modelica code by an `annotation()` statement. This allows for building more complex models or small test cases using drag and drop in a graphical tool like OpenModelica [53]. The lines that connect the model components are not arbitrary, but are closely tied to the semantics of the model with `annotation()` statements that follow the `connect()` equations. These annotations can be found in the full code given both in Supplementary Listing 1–27 and at `https://github.com/CSchoel/shm-conduction`.

It can be argued that the connection in Figure 1 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 simply introducing another layer of abstraction that 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.

To sum up, we could show that using a language that is modular, declarative, readable, open, graphical and hybrid directly benefits the modeling process and leads to models that are both more understandable and easier to reuse and extend.

At this point we have to again note that it is not necessary to use Modelica to gain these benefits. Modelica is one language among several choices, some of which are presented in the supplement, that realize the MoDROGH characteristics to a great extent. What makes Modelica interesting is that it is tested and proven in an industrial setting, but only little known in systems biology. However, similar results could also, for example, be achieved using CellML or MATLAB with the Simulink environment and the Simscape language. For example, one downside of Modelica in contrast to CellML is the mostly missing support for partial differential equations (PDEs).

In general it can be said that in order to build reusable multi-scale, multi-level and multi-class models, one needs to be careful to chose a language that can handle the complexity of their models. It is, however, not sufficient just to chose a language that is suitable in principle. Researchers must be aware of the beneficial characteristics of the language and must be able to utilize them consistently. We all have to think as (software-)engineers and not only implement models for a single purpose, but as reliable part of a solution for larger challenges.

4 Methods

4.1 Material

We used Mo|E version 0.6.3 [54] to write the code of our models and OpenModelica version 1.13.0 [53] as well as Inkscape version 0.91 (`https://inkscape.org/`) to add the component icons. OpenModelica was also used for all simulations. 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 [55].

4.2 Modular conduction model

The first part of the modular model of the human cardiac conduction system is the interface component `UnidirectionalConductionComponent` that serves as a base class for all other components. 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 Kronecker deltas:

```
type InstantSignal = Boolean(quantity="InstantSignal");
connector InstantInput = input InstantSignal;
connector InstantOutput = output InstantSignal;

partial model UnidirectionalConductionComponent
  InstantInput inp "input connector";
  InstantOutput outp "output connector";
end UnidirectionalConductionComponent;
```

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
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Gate;
  extends SHMConduction.Icons.Constant;
  parameter Modelica.SIunits.Time t_first = 0;
  parameter Modelica.SIunits.Period T_refrac = 1;
  Boolean refrac_passed = time - pre(t_last) > T_refrac;
protected
  Real t_last(start=t_first, fixed=true);
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 `Pacemaker` model propagates incoming signals, but also adds an own signal if there was no input for a certain period of time. Additionally it has to be considered that during the refractory period incoming signals are ignored entirely and therefore should not reset the timer of the pacemaker. To accomplish this an external reset signal is added that will only be triggered if the signal passes not only the pacemaker but also the subsequent `RefractoryGate` component. The pacemaker component itself only 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
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Metronome;
  extends SHMConduction.Icons.Constant;
  InstantInput reset "resets internal clock";
  parameter Modelica.SIunits.Period T = 1 "pacemaker period";
  InstantSignal spontaneous_signal = time > pre(t_next)
    "signal generated spontaneously by this pacemaker";
protected
  discrete Modelica.SIunits.Time t_next(start=T, fixed=true);
equation
  outp = inp or spontaneous_signal;
  when spontaneous_signal or pre(reset) then
    t_next = time + T;
  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
    extends UnidirectionalConductionComponent;
    extends SHMConduction.Icons.Hourglass;
    discrete Modelica.SIunits.Duration duration;
    discrete Modelica.SIunits.Period T(start=0, fixed=true)
      "time between last output and following signal";
    discrete Modelica.SIunits.Time t_last(start=0, fixed=true)
      "time of last output";
    discrete Modelica.SIunits.Time t_next(start=-1, fixed=true)
      "time where next output is scheduled";
    Boolean delay_passed(start=false, fixed=true) = time > t_next
      "if false, there is still a signal currently put on hold";
  equation
    outp = edge(delay_passed);
    when inp and pre(delay_passed) then
      T = time - pre(t_last);
      t_next = time + duration;
    end when;
    when outp then
      t_last = time;
    end when;
  end ConductionDelay;
```

Note that this is only a **partial** model that does not specify the behavior of the variable `duration`. This allows to separate the general delay logic from the physiological equation for the AV node which is modeled in the `AVConductionDelay` :

```
model AVConductionDelay
  extends ConductionDelay;
  parameter Modelica.SIunits.Duration k_avc_t = 0.78;
  parameter Modelica.SIunits.Duration T_avc0 = 0.09;
  parameter Modelica.SIunits.Duration tau_avc = 0.11;
  parameter Modelica.SIunits.Duration initial_T_avc = 0.15;
initial equation
  duration = initial_T_avc;
equation
  when inp and pre(delay_passed) then
    duration = T_avc0 + k_avc_t * exp(-T/tau_avc);
  end when;
end AVConductionDelay;
```

The model `ModularConduction` combines the components using **connect** `()` equations to connect the input and output variables. These equations are represented as lines in the graphical representation:

```
model ModularConduction
  extends UnidirectionalConductionComponent;
  extends SHMConduction.Icons.Heart;
  RefractoryGate refrac_av(T_refrac=0.364);
  Pacemaker pace_av(T=1.7);
  AVConductionDelay delay_sa_v;
  discrete Modelica.SIunits.Period T(start=initial_T, fixed=true);
  discrete Modelica.SIunits.Time cont_last(start=0, fixed=true);
equation
  connect(inp, pace_av.inp);
  connect(pace_av.outp, refrac_av.inp);
  connect(refrac_av.outp, pace_av.reset);
  connect(refrac_av.outp, delay_sa_v.inp);
  connect(delay_sa_v.outp, outp);
  when outp then
    T = time - pre(cont_last);
    cont_last = time;
  end when;
end ModularConduction;
```

As you can see, the model `ModularConduction` is again a `UnidirectionalConductionComponent` and can therefore be used as a component in a larger model (such as the SHM).

4.3 PVC model

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 bit 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:

```
  when pre(reset) or (inp and pre(delay_passed)) then
    T = time - pre(t_last);
  end when;
  when pre(reset) then
    t_next = 1e100;
  elsewhen inp and pre(delay_passed) then
    t_next = time + duration;
  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;
  RefractoryGateX refrac_av(T_refrac=0.364);
  Pacemaker pace_av(T=1.7);
  AVConductionDelayX delay_sa_v;
  RefractoryGate refrac_v(T_refrac=0.2);
  discrete Modelica.SIunits.Period T(start=1, fixed=true);
  discrete Modelica.SIunits.Time cont_last(start=0, fixed=true);
  InstantInput pvc(fixed=true) "trigger signal for a PVC";
  Modelica.Blocks.Logical.Or vcont;
  Modelica.Blocks.Logical.Or rpace;
  Modelica.Blocks.Logical.And pvc_upward
    "true if we have PVC that travels upward";
equation
  connect(inp, pace_av.inp);
  connect(pace_av.outp, refrac_av.inp);
  connect(refrac_av.outp, delay_sa_v.inp);
  connect(delay_sa_v.outp, vcont.u1);
  connect(refrac_av.outp, rpace.u2);
  connect(vcont.y, refrac_v.inp);
  connect(refrac_v.outp, outp);
  connect(outp, pvc_upward.u1);
  connect(pvc, pvc_upward.u2);
  connect(pvc_upward.y, refrac_av.reset);
  connect(pvc_upward.y, delay_sa_v.reset);
  connect(pvc_upward.y, rpace.u1);
  connect(rpace.y, pace_av.reset);
  connect(pvc, vcont.u2);
  when outp then
    T = time - pre(cont_last);
    cont_last = time;
  end when;
end ModularConductionX;
```

4.4 Modular contraction experiment setup

Simulation experiments can also be defined directly in Modelica syntax. The following code was used to produce Figure 2:

```
model ModularExample
  ModularConduction modC;
  MonolithicConduction monC;
equation
  modC.inp = monC.signal;
  if time < 5 then
    monC.signal = sample(0,1);
  elseif time < 15 then
    monC.signal = sample(0,3);
  elseif time < 20 then
    monC.signal = sample(0,0.05);
  elseif time < 30 then
```

```
    monC.signal = sample(0,0.8);
  elseif time < 40 then
    monC.signal = sample(0,0.2);
  else
    monC.signal = 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;
```

For the results the variables `monC.T_cont` and `modC.T` were plotted against simulation time.

4.5 PVC experiment setup

The simulation experiment for the PVC model is also defined as a Modelica class. Here the variable `T` was plotted once with `with_sinus = true` and once with `with_sinus = false`:

```
model PVCExample
  ModularConductionX con;
  discrete Modelica.SIunits.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;
  parameter Modelica.SIunits.Period T_normal = if with_sinus then 0.8 else
      con.pace_av.T
    "normal cycle duration without PVC";
  Modelica.SIunits.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.T_avc0
      /2;
  Boolean pvc_b = pre(count_sig) == 12 and t_since_sig > con.refrac_av.T_refrac
      /2;
  Boolean pvc_c = pre(count_sig) == 19 and t_since_sig > T_normal/2;
  Boolean pvc_d = pre(count_sig) == 26 and
    t_since_sig > T_normal - con.delay_sa_v.T_avc0/2;
  Boolean trigger(start=false, fixed=true) = pvc_a or pvc_b or pvc_c or pvc_d;
equation
  con.pvc = edge(trigger);
  if with_sinus then
    con.inp = sample(0, T_normal) "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;
```

Acknowledgements

C. Schölzel would like to thank Denis Noble, Peter Hunter, James Bassingthwaighte, Maxwell Neal and Herbert Sauro for insightful discussions about the Physiome project, CellML and present and future challenges for systems biology. We also thank Alexander Goesmann for his advice regarding the focus of the paper and Jochen Blom for additional proofreading.

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.

Competing Interests

The authors declare no competing interests.

Data availability

The data for figures 2 and 4 can be generated from the model code available on GitHub, `https://github.com/CSchoel/shm-conduction`. No other datasets where generated or analyzed during the current study.

Code availability

The full code of the models and experiments used in this article can be found on GitHub, `https://github.com/CSchoel/shm-conduction`.

References

1. Hodgkin, A. L. & Huxley, A. F. A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve. *The Journal of Physiology* **117,** 500–544 (1952).
2. Bardini, R., Politano, G., Benso, A. & Di Carlo, S. Multi-Level and Hybrid Modelling Approaches for Systems Biology. *Computational and Structural Biotechnology Journal* **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, Berlin, Heidelberg, 2005).
4. Dada, J. O. & Mendes, P. Multi-Scale Modelling and Simulation in Systems Biology. *Integrative Biology* **3,** 86 (2011).

5. Yu, J. S. & Bagheri, N. Multi-Class and Multi-Scale Models of Complex Biological Phenomena. *Current Opinion in Biotechnology* **39,** 167–173 (2016).
6. Waltemath, D. & Wolkenhauer, O. How Modeling Standards, Software, and Initiatives Support Reproducibility in Systems Biology and Systems Medicine. *IEEE Transactions on Biomedical Engineering* **63,** 1999–2006 (2016).
7. Medley, J. K., Goldberg, A. P. & Karr, J. R. Guidelines for Reproducibly Building and Simulating Systems Biology Models. *IEEE transactions on bio-medical engineering* **63,** 2015–2020 (2016).
8. Topalidou, M., Leblois, A., Boraud, T. & Rougier, N. P. A Long Journey into Reproducible Computational Neuroscience. *Frontiers in Computational Neuroscience* **9,** 1–2 (2015).
9. Seidel, H. *Nonlinear Dynamics of Physiological Rhythms* PhD thesis (Technische Universität Berlin, Berlin, Germany, 1997).
10. 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), 367–376.
11. Sarma, G. P. *et al.* Unit Testing, Model Validation, and Biological Simulation. *F1000Research* **5,** 1946 (2016).
12. Miskovic, L., Tokic, M., Fengos, G. & Hatzimanikatis, V. Rites of Passage: Requirements and Standards for Building Kinetic Models of Metabolic Phenotypes. *Current Opinion in Biotechnology* **36,** 146–153 (2015).
13. 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. *Journal of Biomechanical Engineering* **137,** 020905 (2015).
14. Grimm, V. & Railsback, S. F. Pattern-Oriented Modelling: A 'multi-Scope' for Predictive Systems Ecology. *Philosophical Transactions of the Royal Society B: Biological Sciences* **367,** 298–310 (2012).
15. 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. *Clinical Pharmacology & Therapeutics* **92,** 17–20 (2012).
16. 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. *Journal of Experimental Biology* **210,** 1576–1583 (2007).
17. Goldberg, A. P. *et al.* Emerging Whole-Cell Modeling Principles and Methods. *Current Opinion in Biotechnology* **51,** 97–102 (2018).

18. Bartocci, E. & Lió, P. Computational Modeling, Formal Analysis, and Tools for Systems Biology. *PLOS Computational Biology* **12,** e1004591 (2016).

19. Walpole, J., Papin, J. A. & Peirce, S. M. Multiscale Computational Models of Complex Biological Systems. *Annual Review of Biomedical Engineering* **15,** 137–154 (2013).

20. Hucka, M. *et al.* Promoting Coordinated Development of Community-Based Information Standards for Modeling in Biology: The COMBINE Initiative. *Frontiers in Bioengineering and Biotechnology* **3** (2015).

21. Wolstencroft, K. *et al.* SEEK: A Systems Biology Data and Model Management Platform. *BMC Systems Biology* **9** (2015).

22. Novère, N. L. *et al.* Minimum Information Requested in the Annotation of Biochemical Models (MIRIAM). *Nature Biotechnology* **23,** 1509–1515 (2005).

23. Lopez, C. F., Muhlich, J. L., Bachman, J. A. & Sorger, P. K. Programming Biological Models in Python Using PySB. *Molecular Systems Biology* **9,** 646 (2013).

24. Olivier, B. G., Rohwer, J. M. & Hofmeyr, J.-H. S. Modelling Cellular Systems with PySCeS. *Bioinformatics* **21,** 560–561 (2005).

25. Choi, K. *et al.* Tellurium: An Extensible Python-Based Modeling Environment for Systems and Synthetic Biology. *Biosystems* **171,** 74–79 (2018).

26. Smith, L. P., Bergmann, F. T., Chandran, D. & Sauro, H. M. Antimony: A Modular Model Definition Language. *Bioinformatics* **25,** 2452–2454 (2009).

27. Bezanson, J., Edelman, A., Karpinski, S. & Shah, V. B. Julia: A Fresh Approach to Numerical Computing. *SIAM Review* **59,** 65–98 (2017).

28. 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* **30** (Gent, Belgium, 1997), 151–155.

29. Mateják, M. *et al. Physiolibrary - Modelica Library for Physiology* in *Proceedings of the 10th International Modelica Conference* **96** (Lund, Sweden, 2014), 499–505.

30. Maggioli, F., Mancini, T. & Tronci, E. SBML2Modelica: Integrating Biochemical Models within Open-Standard Simulation Ecosystems. *Bioinformatics.* Epub ahead of print, btz860 (2019).

31. Hellerstein, J. L., Gu, S., Choi, K. & Sauro, H. M. Recent Advances in Biomedical Simulations: A Manifesto for Model Engineering. *F1000Research* **8,** 261 (2019).

32. Mulugeta, L. *et al.* Credibility, Replicability, and Reproducibility in Simulation for Biomedicine and Clinical Applications in Neuroscience. *Frontiers in Neuroinformatics* **12** (2018).

33. Cooling, M. T. *et al.* Standard Virtual Biological Parts: A Repository of Modular Modeling Components for Synthetic Biology. *Bioinformatics* **26,** 925–931 (2010).

34. Guyton, A. C., Coleman, T. G. & Granger, H. J. Circulation: Overall Regulation. *Annual Review of Physiology* **34,** 13–44 (1972).

35. Blochwitz, T. *et al. The Functional Mockup Interface for Tool Independent Exchange of Simulation Models* in *Proceedings of the 8th International Modelica Conference* (Dresden, Germany, 2011), 105–114.

36. 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* (Munich, Germany, 2012), 173–184.

37. Zhu, X.-G. *et al.* Plants *in Silico*: Why, Why Now and What?-An Integrative Platform for Plant Systems Biology Research. *Plant, Cell & Environment* **39,** 1049–1057 (2016).

38. Neal, M. L. *et al.* A Reappraisal of How to Build Modular, Reusable Models of Biological Systems. *PLOS Computational Biology* **10,** e1003849 (2014).

39. Mirschel, S., Steinmetz, K., Rempel, M., Ginkel, M. & Gilles, E. D. ProMoT: Modular Modeling for Systems Biology. *Bioinformatics* **25,** 687–689 (2009).

40. Cooling, M. T., Hunter, P. & Crampin, E. J. Modelling Biological Modularity with CellML. *IET Systems Biology* **2,** 73–79 (2008).

41. Kell, D. The Virtual Human: Towards a Global Systems Biology of Multiscale, Distributed Biochemical Network Models. *IUBMB Life* **59,** 689–695 (2007).

42. Holzhütter, H.-G., Drasdo, D., Preusser, T., Lippert, J. & Henney, A. M. The Virtual Liver: A Multidisciplinary, Multilevel Challenge for Systems Biology. *Wiley Interdisciplinary Reviews: Systems Biology and Medicine* **4,** 221–235 (2012).

43. Loew, L. M. & Schaff, J. C. The Virtual Cell: A Software Environment for Computational Cell Biology. *Trends in Biotechnology* **19,** 401–406 (2001).

44. Zhu, H., Huang, S. & Dhar, P. The next Step in Systems Biology: Simulating the Temporospatial Dynamics of Molecular Network. *BioEssays* **26,** 68–72 (2004).

45. Le Novère, N. *et al.* The Systems Biology Graphical Notation. *Nature Biotechnology* **27,** 735–741 (2009).

46. Alves, R., Antunes, F. & Salvador, A. Tools for Kinetic Modeling of Biochemical Networks. *Nature Biotechnology* **24,** 667–672 (2006).

47. 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).

48. 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).

49. Åkesson, J. R., Gäfvert, M. & Tummescheit, H. *JModelica—An Open Source Platform for Optimization of Modelica Models* in *Proceedings of the 6th Vienna International Conference on Mathematical Modelling* **34** (Vienna, Austria, 2009).

50. Hastings, J. *et al.* ChEBI in 2016: Improved Services and an Expanding Collection of Metabolites. *Nucleic Acids Research* **44,** D1214–D1219 (2016).

51. 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).

52. Neal, M. L. *et al.* Semantics-Based Composition of Integrated Cardiomyocyte Models Motivated by Real-World Use Cases. *PLOS ONE* **10,** e0145621 (2015).

53. 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).

54. 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, 2017), 815–822.

55. Schölzel, C., Blesius, V., Ernst, G. & Dominik, A. *Required Characteristics for Modeling Languages in Systems Biology: A Software Engineering Perspective* preprint 10.1101/2019.12.16.875260v1 (bioRxiv, 2019).