Model Composition for Biological Mathematical Systems

Mandeep Gill[1], Steve McKeever[2] and David Gavaghan[1]
[1]*Department of Computer Science, University of Oxford, Parks Road, Oxford, UK*
[2]*Department of Informatics and Media Studies, Uppsala University, Uppsala, Sweden*
Mandeep.Gill@cs.ox.ac.uk

Keywords: Modules, Collaborative Modelling, Biological Mathematical Systems, Heart Models, Generics.

Abstract: Mathematical models are frequently used to model biological process, such as cardiac electrophysiological systems. In order to separate the models from the implementations, and to facilitate curation, domain specific languages (DSLs) have become a popular and effective means of specifying models (Lloyd et al., 2004; Hucka et al., 2004). In previous papers (Gill et al., 2012a; Gill et al., 2012b; McKeever et al., 2013) we have argued for including parameterised modules as part of such DSLs. We presented our *Ode* language and showed how models could be created in a generic fashion. In this paper we extend our work with concrete examples and simulation results. We show how complex heart models can be constructed by aggregation, encapsulation and subtyping. Our use-case retraces the steps taken by (Niederer et al., 2009), which investigated the common history between cardiac models, and shows how they can be cast in our language to be reused and extended. Our DSL enables 'physiological model engineering' through the development of generic modules exploiting high cohesion and low coupling.

1 INTRODUCTION

Modelling is collaborative. With the advent of domain specific languages for describing models we are looking at incorporating features from programming language theory to improve collaborative design of physiological models. Modular programming involves describing a system in terms of encapsulated modules and the interactions between them through well-defined interfaces. Such modular program design is commonly used in software engineering when designing large-scale reusable systems, with the key tenets including encapsulation, abstraction, reuse and extensibility (Sommerville, 2007; Booch, 2006).

We have been developing our own experimental language *Ode*, inspired by CellML (Lloyd et al., 2004), to investigate module features that facilitate collaborative design at the domain specific language level. *Ode* is influenced by biological DSLs and functional programming languages rooted in the lambda calculus (Barendregt, 1985). The syntax and semantics are designed to exploit modeller familiarity with high-level languages such as Matlab and Python. Figure 1 presents the *Ode* grammar.

The semantics of this core language have been described in a previous paper (Gill et al., 2012a). Constant *values* may be defined as the result of mathematical expressions; the syntax for such expressions is similar to most programming languages. *Components* are a means to group, abstract and parameterise repeated computations. However they are inlined at compile-time in a manner similar to C++ templates. *Piecewise* terms enable conditional control-flow similar to a switch statement. A deliberate limitation is the lack of looping control-flow constructs that, when coupled with a recursive restriction on components, ensures termination during model simulation.

$$
\begin{aligned}
\mathit{odeStmt} &\leftarrow \mathit{componentDef} \mid \mathit{valueDef} \mid \mathit{odeDef} \\
\mathit{componentDef} &\leftarrow \texttt{component}\, f\,(id_1,\ldots,id_n) \\
&\qquad \{\mathit{exprStmt}_1,\ldots,\mathit{exprStmt}_j, \texttt{return}\,(E_1,\ldots,E_n)\} \\
\mathit{valueDef} &\leftarrow \texttt{val}\, id_1,\ldots,id_n = E \\
E &\leftarrow t_1 * t_2 \mid t_1 / t_2 \mid t_1 > t_2 \mid \ldots \mid t_1 \,\&\&\, t_2 \mid t \\
t &\leftarrow (E) \mid \mathit{number} \mid \mathit{boolean} \mid \mathit{piecewise} \\
&\quad \mid f(E_1,\ldots,E_n) \mid \texttt{time} \mid \mathit{id} \\
\mathit{piecewise} &\leftarrow \texttt{piecewise}\,\{E_{C1} : E_{T1},\ldots,E_{Cn} : E_{Tn}, \\
&\qquad \texttt{default} : E_{def}\} \\
\mathit{odeDef} &\leftarrow \texttt{ode}\,\{\texttt{init} : id_v, [\,\texttt{deltaVal} : id_{delta}\,]\,\} = E
\end{aligned}
$$

Figure 1: Abbreviated syntax of the core *Ode* language using a simplified variant of EBNF that assumes common rules.

The basic expression language is extended with

several constructs to support the mathematical modelling of (biological) systems over time, including ordinary differential equations (ODEs). ODEs require an initial value $y(0)$, and an expression describing the derivative, y'.

The focus of this research has been to apply software engineering techniques that allow models developed in *Ode* to be highly reusable and extensible, facilitating rapid composition to investigate certain parameters of interest. To the best of our knowledge, these characteristics have eluded the majority of biological modelling software frameworks to date.

Our paper in outlined as follows. Section 2 motivates modular model design. In Section 3 we provide a precise description of our module system and motivating example. In Section 4 we present results from simulating composite models derived from experimentally-related model modules. Finally in Section 5 we conclude our work and discuss current research. We demonstrate the utility of the module system using examples from a modular form of the Hodgkin-Huxley (HH52) electrophysiological model of the squid giant-axon (Hodgkin and Huxley, 1952).

2 MODULAR MODEL DESIGN AND DEVELOPMENT

Modelling biological systems is an integrative, interdisciplinary process, drawing on work from researchers in a global environment from a variety of fields, including physiologists, mathematicians, and computer scientists. Models themselves are comprised from data collected through various studies and are developed iteratively, with newer, generally more realistic, models derived from existing ones (Niederer et al., 2009).

As such a need exists to efficiently facilitate the collaborative development and reuse of models. This would allow for the safe construction of models, guaranteeing the validity of models developed collaboratively and derived from multiple sources, and ensuring that they may be composed successfully. We believe our research provides features that facilitate such collaborative modelling in a structured and safe manner.

We present a summation of the methodologies used to structure and develop large-scale reusable biological mathematical models using *Ode*, inspired by many techniques utilised in software engineering (Sommerville, 2007; Booch, 2006).

Encapsulation is the wrapping up of operations and attributes into a module, so that those attributes may only be manipulated through or accessed via the operations provided by the module. We can view modules as similar to objects, representing a convenient data-centric manner to decompose systems into understandable and manageable units/building blocks. Good encapsulation hides the details of a module's internal attributes and operations from its users. These techniques are known as information hiding and implementation hiding, and their use is essential for promoting the understandability of code within a reusable domain-specific modelling framework. It allows abstracting away the low-level details and providing a high-level interface to the module functionality. Explicit export definitions within module declarations facilitate implementation hiding and help to achieve this goal. Figure 2 provides an example of how we may encapsulate several cell-level parameters from the HH52 model into a module.

HH52DefaultParameters
+C_m : float mV = 1
+E_R : float mV = -73
+i_Stim : float mA cm-2

Figure 2: Cell and stimulus parameters for the HH52 model grouped and encapsulated into a single *parameters* module.

Generics, provided via parameterised modules, enable modellers to operate on encapsulated objects in an abstracted manner, enabling polymorphic reuse and substitutability, similar to C++ templates. It is a style of programming in which algorithms are written in terms of to-be-specified-later constructs that are then instantiated when needed by replacing the generic variables with appropriate concrete parameters. In the context of heart modelling, we show that this technique enables the reuse of modules, the creation of alternative implementations, and the mixed usage of ion channel representations from a variety of models.

Structural subtyping is implemented at the module level, where we say that module *B* is a subtype of module *A*, the supertype. This occurs when module *B* has the same (or a superset of the) interface as *A*, yet may contain a different implementation. In this case modules of type *A* can be *substituted* by modules of type *B*. This occurs because methods written to operate on elements of the supertype can also operate on elements of the subtype; in a manner similar to inheritance in object-orientated programming (OOP). However it is implemented at compile-time without any performance penalty and does not require a direct, nominative relation between the supertype and subtype. Any modules that have a similar 'shape' can be considered a subtype, providing low-coupling between modules and allowing rapid development and substitution of alternate implementations. If used properly, subtyping, coupled with generics, can improve the understandability of model code by reducing the conceptual distance between code and the real-world

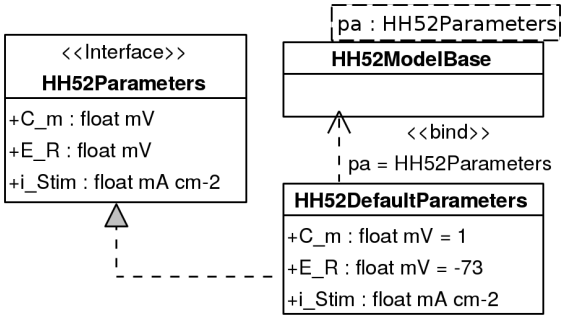

Figure 3: Application of a generic module parameter, the binding relationship indicates the assignment of a module as an argument, generating a new module. The generic module must implement the *parameters* interface.

system which the code models. For instance, Figure 3 presents a use of both generics and subtyping to produce a parameterised module of a given supertype.

In software development, code reuse provides many benefits — the code has already been developed, tested and potentially deployed. A similar effect is observed with model code, where equations and definitions are often similar and should be reused if possible. However in both domains, code often requires modifications specific to each particular instance of reuse. Containment-based relationships, such as composition and aggregation, can be implemented by modules to enable lightweight and flexible code reuse.

Modules may be imported and wrapped within containing modules to re-implement and augment existing functionality, delegating to existing code where required. This provides code-reuse through composition and aggregation rather than OO inheritance, achieving reuse statically rather than at run-time.

This technique therefore works hand-in-hand with the substitutability property of modules and structural subtyping mechanism to allow modellers to create model implementations which are both malleable and extensible. It enables type-safe substitution whilst avoiding deep inheritance-chains and hierarchies often seen in systems implemented using traditional OO-languages (Booch, 2006).

When parameterised modules are utilised with aggregation they allow generic code reuse, abstracting the particular instances until simulation time. This provides compile-time substitutability and a powerful mechanism for rapid model construction and modification. Figure 4 demonstrates aggregation to structure and reuse code with the HH52 parameters module.

3 MODULE SYSTEM – SYNTAX AND SEMANTICS

$$
\begin{aligned}
moduleStmt &\leftarrow moduleDef \mid importStmt \mid applyModule \\
moduleDef &\leftarrow \texttt{module}\ id\ moduleBody \mid \\
&\qquad \texttt{module}\ id\ (id_1,\ldots,id_n)\ moduleBody \\
moduleBody &\leftarrow \{\, odeStmt_1,\ldots,odeStmt_n \,\} \\
importStmt &\leftarrow \texttt{import}\ id_1\ [\texttt{as}\ id_2] \\
exportStmt &\leftarrow \texttt{export}\ id_1,\ldots,\ id_n] \\
appModule &\leftarrow id = f(id_1,\ldots,id_n) \mid id
\end{aligned}
$$

Figure 5: Module extensions to *Ode* DSL.

In this section we describe the *Ode* module system used to structure models into modules consisting of collections of related value and component definitions that form some logical (and potentially biological) grouping. We describe the syntax and semantics, the grammar for the module language is provided in Figure 5. We discuss parameterised modules, enabling compile-time substitution of components and generic, interface-driven development that provides further opportunities for abstraction and model reuse.

The module system was inspired by OCaml, where it is effectively a higher-level language that enables the programmable creation of independent modules (Leroy, 2000). The *Ode* module system operates similarly; providing variable definitions, references, function abstraction and application (i.e. parametrised modules), and limited computation, all at the typed module level. This provides extensive power to reconfigure models for particular tasks.

3.1 Module Definition

```
module Parameters {
  val E_R = -75 { unit : mV }
  val Cm = 1 { unit : mF.cm^-2 }
  val period = 60 { unit : ms }
  val i_Stim = piecewise {
      time >= 10 and time <= 10.5:
          20 { unit : mA.cm^-2 },
      default: 0 { unit : mA.cm^-2 }
  } }
```

A standard module in *Ode* consists of a fixed collection of related *Ode* expressions, i.e. value and component definitions, that form some grouping. This may be a logical or biological grouping, e.g. an ion channel within a cardiac model. For instance, the code fragment above introduces the syntax for creating a module that encapsulates several cell-level parameters within

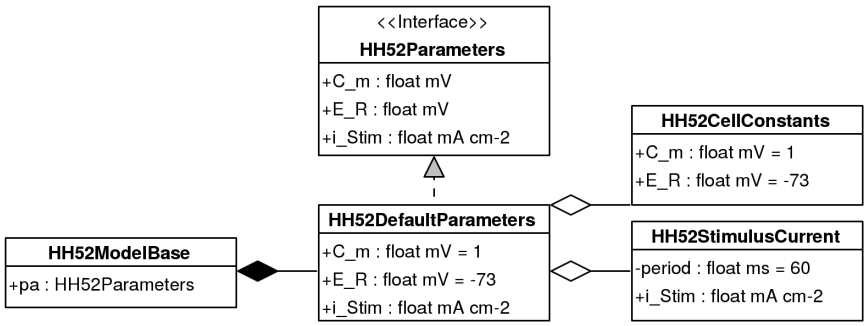

Figure 4: Diagram indicates use of containment and aggregation to provide code structure and reuse. We compose the parameters module with the main model, whilst specifying that it is a subtype of `HH52Parameters`. The `HH52DefaultParameters` module implements this interface via the aggregation and reuse of two sub-modules containing the required code.

the HH52 model, as was illustrated in Figure 2. A module declaration uses the `module` keyword followed by the module name and body.

Several modules may be defined within a single file, where the directory path in which the files are placed creates a module hierarchy. Modules within an *Ode* file may be referenced using a dot-notation that corresponds to lookup within the module hierarchy. This structure enables the creation of multiple repositories of modelling components, each with their own hierarchy, that can be shared, reused and uniquely identified in the system.

3.2 Importing & Using Modules

```
module HH52Model {
  import Cardiac.HH52.Parameters as Pa
  // leakage current
  val E_L = Pa.E_R + 10.613 { unit : mV }
  val i_L = 0.3 { unit : mS.cm^-2 } * (V - E_L)
  // ... channels i_Na & i_K ...
  init V = -75 { unit : mV }
  ode { init : V } =
     -(-Pa.i_Stim + i_Na + i_K + i_L) / Pa.Cm
}
```

A module may references another module's expressions through an import declaration. This is demonstrated in the above code sample, illustrating the composition of the HH52 parameters module with the HH52 base model, as in Figure 2. We use a dot-notation in the import declaration to uniquely reference to a filepath within the module hierarchy and a module with the specified file. Thus, in the above example the HH52 base model imports a module `Parameters` from a file called `HH52` within the `Cardiac` directory of an available and loaded module repository. When processing an import declaration, the module is retrieved according to the available repositories and loaded into the global module environment.

The visible definitions of an imported module are again accessed using a dot-notation on the qualified module name, demonstrated by references to `Pa` above.

3.3 Module Interface

Modules expose an interface, or signature, comprised from the collection of identifiers and their types visible from outside the module. The uses of a module within a model also leads to the implicit creation of a required interface that any imported modules must implement. When importing a module, interfaces are matched and checked by the module type-system to ensure successful composition.

A modeller may configure the visibility of certain definitions to modify the interface. This may be achieved by structuring the model code such that only exportable definitions are visible, or by explicitly declaring the definitions to export (similar to the *public/private* access modifiers in OO languages) through the use of an `export` command.

In this manner only the vital information is exported in the interface, mimicking biological mechanisms of compartmentalisation and containment.

3.4 Parametrised Modules

Grouping related definitions into modules enables modularity and code reuse, however the modules and the abstractions within them are fixed rather than generic. When creating reusable components it is desirable to configure them according to a specific use-case, for instance altering the parameters of a reusable ion channel or creating protocols to compare simulations against experimental data (Cooper et al., 2011).

In several languages parameterised modules present a flexible and powerful means for accomplishing this. They are implemented as compile-time construct used to create generic components in a similar manner to ML functors (Leroy, 2000) and C++ tem-

plates, allowing type-safe substitutability of components with differing implementation details.

Modules may be parameterised by other modules leading to the creation of complex, specialised modules via a form of aggregation. This flexibility and abstraction enables rapid investigations into parameter variations, including parameter sweeps and sensitivity analysis from a single code-base (O'Hara et al., 2011).

3.5 Parameterised Module Definition

```
module HH52ModelBase(Pa) {
  // leakage channel
  val E_L = Pa.E_R + 10.613 { unit : mV }
  val i_L = 0.3 { unit : mS.cm^-2 } * (V - E_L)
  // ... channels i_Na & i_K ...
  init V = -75 { unit : mV }
  ode { init : V } =
     -(-Pa.i_Stim + i_Na + i_K + i_L) / Pa.Cm
}
```

The above example demonstrates parameterised modules within *Ode*, where the specific parameters module utilised by the HH52 base model is parameterised and made generic. The syntax extends module definitions to include a list of module parameters, these are generic module arguments that are determined during application. Parameterised modules can be considered analogous to functions over modules, they take a set of input module arguments and return a new module.

Parameterised modules enable a form of interface-based model construction that allows for the specialisation of reusable module components. They facilitate several component based modelling techniques through the modification of parameters and equations used within a module, for instance the creation of multiple model subcomponents ranging in complexity and accuracy to minimise computational demands.

3.6 Parameterised Module Application

```
module HH52Model = HH52ModelBase(Parameters)
```

The above code fragment demonstrates applying a concrete version of the HH52 parameters module to the base module, resulting in a complete version of the HH52 model, as was illustrated in Figure 3.

Applying a functor is the process of replacing the generic module parameters with real modules, the effect of which is to instantiate a new module derived from applying the parameters to the functor body. This new, concrete module can then be used elsewhere.

We can dynamically apply functors, enabling the run-time configuration of models that exhibit certain behaviours and investigation of effects using particular parameters. This provides a flexible means for rapidly constructing families of specialised models that can be controlled by an external process at run-time.

4 MODULAR CARDIAC SIMULATIONS

In this section we utilise the module system and DSL abstractions previously described to design a component-based modular modelling framework. We hope this will eventually enable the agile development, reuse and investigation/experimentation of cardiac electrophysiological models in a collaborative fashion (Noble and Rudy, 2001).

Our use case is influenced by research in (Niederer et al., 2009) that investigated the common history between cardiac and ion channel models, but we conduct similar simulations through programmable modular composition rather than manual model construction. Our sample framework is used to conduct several simulations, generating action potential (AP) membrane voltage curves for cardiac models created from the dynamic substitution of ion channel modules. This is intended to show the ease at which a well-designed framework lends itself to reuse, extensibility and flexibility for modellers to investigate particular phenomena.Furthermore it demonstrates the use of the type system to ensure subsequent model validity.

4.1 Methodology

We investigate and construct a modelling architecture suitable for creating models in a flexible and extensible manner comprised from individual modules. This framework is used to develop a repository of cardiac models to help qualitatively determine the DSL's effectiveness in the large-scale structure and rapid, collaborative development, modification and reuse of models. Several cardiac models are safely composed in a scripted manner from modules in the repository.

4.1.1 Cardiac Models

The cardiac models are listed in Table 1, and are chosen to demonstrate the iterative development process and reuse, of parameters and equations.

4.1.2 Model Structure & Development

We separate the cell models into reusable and substitutable ion channel objects that each model the flow of a particular charged ion across the cell membrane, resulting in the generation of an AP. We abstract and

Model	Year	Species	Ion Channels	State Variables
BR77 * (Beeler and Reuter, 1977)	1977	Mammalian	4	8
LR91 (Luo and Rudy, 1991)	1991	Guinea Pig	6	8
LRd94 (Luo and Rudy, 1994)	1994	Guinea Pig	11	12

Table 1: Cardiac ventricular electrophysiological models used within study, with base model denoted with a *

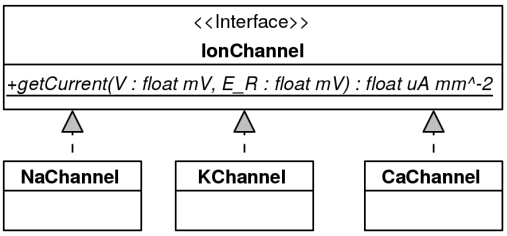

Figure 6: Interface for reusable ion channel models, consists of component, `getCurrent`, that computes I_{Ion} for calculating the membrane voltage. All ion channel models implement this interface, as `NaChannel` and `KChannel` do.

modularise at this point as most cardiac cell experiments, and subsequent research data, occurs at the ion channel level. As more information is derived regarding their function we hope that their differing representations within cell models may be used interchangeably and gradually unify into single canonical form. This abstraction can be expressed neatly within our module system; multiple modules representing the same ion channel may be created, coexist and be substituted within existing and ever more complex newer models. In this way, appropriate relationships, which mimic real-world relationships, are defined between modules in our modelling domain.

A standardised interface for ion channels models was determined that enables their use and replacement within cell models. This interface is illustrated in Figure 6, and consists of a single component, `getCurrent`. The input parameters to this component represent the transmembrane potential and equilibrium potential respectively, and the output is the current generated due to ionic flow.

A cardiac model structurally depends upon and contains the ion channel modules, requiring that they expose the `IonChannel` signature to be compatible. This signature is checked at compile-time by the type system to ensure that only valid ion channel objects are used. The structure used to associate a cardiac model and its related ion channels is illustrated in Figure 7.

We have used encapsulation to separate definitions for ion channels into modules that expose a specified interface. We now demonstrate how aggregation, encapsulation and subtyping may be used to abstract out common functionality and enable code reuse. From a cardiac modelling perspective, we can define common functionality for ion channels. These are then encapsulated and extended in later, more complex channel model subtypes through aggregation (mirroring the real life development of such models), as was demonstrated in Figure 2. We can thus delegate to existing code, and override and specialise as needed within the newer encapsulated module, whilst ensuring it exposes the same interface to enable substitutability.

We use generics to specify the input ion channels to each model via parameterised modules, allowing simulation-time configuration and instantiation of concrete modules. Generics provide a standardised manner to alter models and parameters, providing substitutability without incurring a performance penalty. For example, in Figure 8 the model, `BR77Model`, contains the generic module reference `NaChannel` that exposes the `IonChannel` interface. We do not need to specify during model development which specific `NaChannel` implementation we are referring to.

We utilise encapsulation, aggregation, subtyping and generics to abstract behaviour and provide substitutability of reusable model components. This benefits modellers by enabling model reuse, component-driven development, and type-checked model composition. These techniques orthogonally help model development by increasing cohesion and decreasing coupling.

4.2 Results

```
module BR77NaChannel {
  export (getCurrent)
  component getCurrent(V, E_R) {
    // ... channel constants & computations ...
    // calculate current
    val i_Na = (g_Na*pow(m, 3.0)*h*j+g_Nac)
               * (V - E_Na)
    return (i_Na)
  } }
```

The above listing contains a segment from the BR77 I_{Na} model (Beeler and Reuter, 1977), this segment includes the top-level component that comprises the module interface and implements the type-signature required for reusable channels within our framework.

```
module BR77Model(NaChannel) {
  // membrane voltage
  init V = -84.624 { unit : mV }
  //  ... model constants and computations ...
  val i_Na = NaChannel.getCurrent(V, E_R)
  // Membrane Voltage ODE
  ode { initVal : V, deltaVal : dV } =
              (i_Stim - (i_Na+i_s+i_x1+i_K1))/C
```

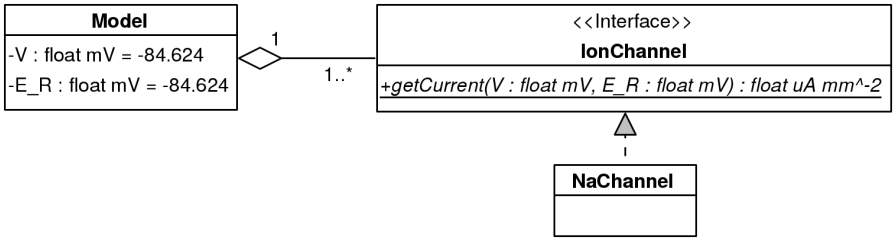

Figure 7: Modular structure of a cardiac model in the repository. The base model represents the cell membrane and calculates the membrane voltage through references to many ionic channels that implement the `IonChannel` interface.

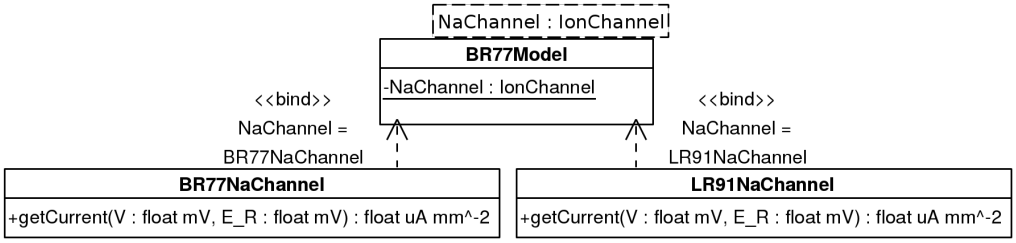

Figure 8: Using generics within the model framework — either `BR77NaChannel` or `LR91NaChannel` can be applied as module argument to the base `BR77Model` cardiac model to represent the generic `NaChannel`.

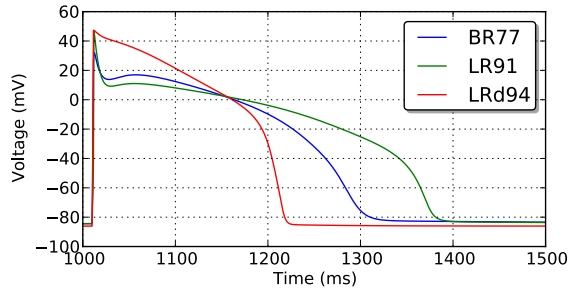

Figure 9: AP curves from applying original I_{Na} channels into models using modules.

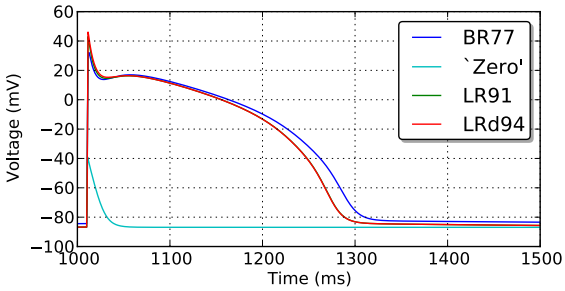

Figure 10: AP curves from applying alternate I_{Na} channels into BR77 cardiac cell model using modules.

}

Similarly, the listing above contains a segment from a cardiac model within the framework. It shows a BR77 cell model parametrised by a module representing I_{Na}, this may be provided by any ion channel model that implements the correct module signature.

We compiled and ran a series of simulations that demonstrate the substitution and impact of the *NaChannel* within the models in a fully scripted manner. This was inspired by the work in (Niederer et al., 2009) and the modular framework was structured to enable such rapid-modifications and experimentation safely. We initially composed each cardiac base model with its corresponding sodium channel and simulated the resulting complete model to generate the reference AP curves depicted in Figure 9. The sodium channels from the remaining models in the set were then applied to the BR77 cell model to generate a new family of

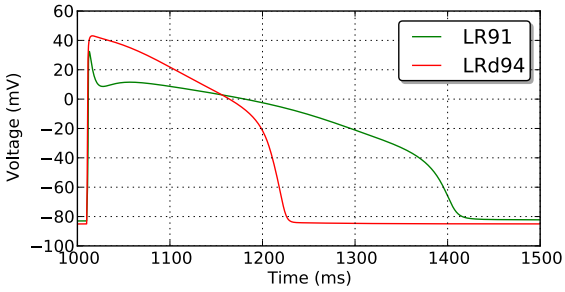

Figure 11: AP curves from applying BR77 I_{Na} channel into alternate models using modules.

models whose AP curves are presented in Figure 10. This plot shows that the LR91 and LRd94 I_{Na} models only slightly shorten the AP duration of the BR77 cellular model. I_{Zero}, implementing the same module signature but returning a constant zero current, fails to

trigger an AP as expected as sodium ions are responsible for AP initialisation.

We finally performed the inverse operations, applying the I_{Na} BR77 model into the remaining cellular models to investigate its influence. The AP curves from simulating these models are presented in Figure 11, where replacing the I_{Na} from the LR91 and LRd94 models increases the AP duration and alters the peak AP. We do not consider these results particularly important in terms of their biological meaning, instead they serve to demonstrate the application and benefits of modular programming and architecture to developing reusable, extensible and dynamic models.

5 CONCLUSIONS

Our use-case driven design process has yielded a modular framework which allows a family of cardiac electrophysiological models to be extended both intuitively and with relative ease. This extensibility has been achieved by utilising features such as generics and subtyping which exploit the substitutability property of module hierarchies along with encapsulation for code structure and aggregation for code reuse. This is a common approach used to structure large-scale software projects. New cardiac models may be created that utilise existing ion channel models, enabling the DSL to naturally capture the reuse of models and experimental data. Thereby increasing robustness which is especially important if these models are ever used in a predictive pharmaceutical or clinical setting.

We can create hybrid models that include ion channel representations derived from a variety of models. This can be undertaken automatically by the module system in a collaborative manner simply by utilising ion channel modules from different models when instantiating a generic cell model. Generic modules may be used to perform sensitivity analysis of the model to parameter fluctuations. They can also be used to alter equations, for example to model ion channel changes and resulting effect on the AP caused by mutation or drug block (O'Hara et al., 2011). As an example, type-correct generic adapter modules may be created that simply reduce the current by 50%, enabling investigation into drug block in an abstracted manner.

Models created in the fashion that we have illustrated do not depend on each other explicitly. They do not communicate with one another either. Parameterisation only requires the type signature of the parameter object to be known. Consequently they demonstrate low coupling and high cohesion, an aspect of model design that we feel is important for reusability and extensibility (McKeever et al., 2013).

We are currently developing a repository to reproduce existing models. However other use-cases are possible with such a modular repository for collaborative and iterative model development. This includes enabling the reuse of modules and the creation of alternative implementations.

REFERENCES

Barendregt, H. (1985). *The lambda calculus: Its syntax and semantics*, volume 103. Access Online via Elsevier.

Beeler, G. and Reuter, H. (1977). Reconstruction of the action potential of ventricular myocardial fibres. *The Journal of physiology*, 268(1):177.

Booch, G. (2006). *Object Oriented Analysis and Design with Applications*. Pearson.

Cooper, J., Mirams, G., and Niederer, S. (2011). High-throughput functional curation of cellular electrophysiology models. *Progress in biophysics and molecular biology*, 107(1):11–20.

Gill, M., McKeever, S., and Gavaghan, D. (2012a). Modular Mathematical Modelling of Biological Systems. In *Symposium on Theory of Modeling and Simulation (TMS'12)*.

Gill, M., McKeever, S., and Gavaghan, D. (2012b). Modules for Reusable and Collaborative Modelling of Biological Mathematical Systems. In *21ST IEEE International WETICE Conference (WETICE-2012)*.

Hodgkin, A. and Huxley, A. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500.

Hucka, M., Finney, A., Bornstein, B., Keating, S., and Shapiro, B. (2004). Evolving a lingua franca and associated software infrastructure for computational systems biology: the Systems Biology Markup Language (SBML) project. *Systems Biology*.

Leroy, X. (2000). A modular module system. *Journal of Functional Programming*, 10(3):269–303.

Lloyd, C., Halstead, M., and Nielsen, P. (2004). CellML: its future, present and past. *Progress in Biophysics and Molecular Biology*, 85:433–450.

Luo, C. and Rudy, Y. (1991). A model of the ventricular cardiac action potential. Depolarization, repolarization, and their interaction. *Circulation research*, 68(6):1501.

Luo, C. and Rudy, Y. (1994). A dynamic model of the cardiac ventricular action potential. II. Afterde-

polarizations, triggered activity, and potentiation. *Circulation research*, 74(6):1097–113.

McKeever, S., Gill, M., Connor, A., and Johnson, D. (2013). Abstraction in physiological modelling languages. In *Symposium on Theory of Modeling & Simulation (TMS'13)*.

Niederer, S., Fink, M., Noble, D., and Smith, N. (2009). A meta-analysis of cardiac electrophysiology computational models. *Experimental physiology*, 94(5):486–95.

Noble, D. and Rudy, Y. (2001). Models of cardiac ventricular action potentials: iterative interaction between experiment and simulation. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 359(1783):1127–1142.

O'Hara, T., Virág, L., Varró, A., and Rudy, Y. (2011). Simulation of the undiseased human cardiac ventricular action potential: model formulation and experimental validation. *PLoS computational biology*, 7(5):e1002061.

Sommerville, I. (2007). *Software Engineering*. Addison Wesley.