Abstraction in Physiological Modelling Languages

Steve McKeever[1] Mandeep Gill[2] Anthony Connor[2] David Johnson[2]
[1]Department of Informatics and Media, Uppsala University, Sweden
[2] Department of Computer Science, University of Oxford, UK
Email: steve.mckeever@im.uu.se, {mandeep.gill, anthony.connor, david.johnson}@cs.ox.ac.uk

Keywords: Domain Specific Languages, Biological Modelling, Mathematical Modelling, Generics, Inheritance

Abstract

In this paper we discuss two projects looking at applying advanced abstraction mechanisms from software engineering to the field of physiological modelling. We focus on two abstraction mechanisms commonly found in modern object-oriented programming languages: generics and inheritance. Generics allows classes to take other classes as parameters, allowing common behaviour to be described with particularities abstracted away. We demonstrate this technique on an example from heart modelling. Inheritance allows one to reuse code and to establish a subtype of an existing object. We focus on the benefits reaped from inheritance where this property enables run-time substitutability. This technique is demonstrated within the context of multi-scale tumour modelling. Finally, we look at how combining both techniques enables greater modularity and the construction of a model driven framework for the rapid creation and extension of families of biological models.

1. INTRODUCTION

Extensibility is a characteristic of systems design that is a measure of the degree and effort to which a system can introduce new functionality, with minimal disruption to its existing behaviour. Software designed to be extensible tends to exhibit *low coupling* and *high cohesion*. Coupling is a measure of the interdependency between program modules, while cohesion is the degree to which functionality within a module is related. Low coupling is typically achieved by designing modules to interact through well-defined interfaces independently from their internal representation, making them easier to reuse and extend. High cohesion occurs when software is designed to encapsulate functionality that is closely related, making module code easier to understand and maintain. Designing software that exhibits both of these properties is technically demanding, but ultimately leads to more extensible software being developed, a desirable quality that is invaluable to encouraging the longevity of software.

Within the domain of physiological modelling, there are different approaches to describing models beyond documenting the raw mathematics or formal specification. A wide range of software tools for developing models and running simulations of physiology have been developed and many model developers rely on general purpose modelling tools (e.g. Mathematica, MATLAB etc.) and software programming languages (e.g. R, C/C++, Perl etc.). Machine readable description languages are needed to develop computer-based simulations, however, the use of general purpose tools and environments creates challenges for model reuse and composition, since different model implementations may be based on completely different technological frameworks. Without a Common Reference Model [1] to enable interoperability of software tools, and careful consideration in the design of such a domain-specific language (DSL), model extensibility will be difficult to achieve.

Over the last decade, two markup languages for computational biology have emerged as standards for model description, with the Systems Biology Markup Language (SBML) [2] and CellML [3] research programmes. SBML is a domain-specific XML-based markup language that addresses biochemical processes at the molecular scale. Developed out of the cardiac modelling community, CellML is a modelling markup language that aims to cover a range of biological phenomenon, primarily cell-function. Both SBML and CellML encapsulate models and internal components to a certain degree, but their approaches are relatively basic to ensure backwards compatibility with existing models. What neither language takes into account is that by simply allowing direct connectivity of data between modules, any notion of cohesion is not accounted for. Models typically simulate multiple processes where biological concepts may be spread over different parts of the code or multiple concepts represented in one portion of code. The grouping of concerns to achieve better modularity and encapsulation is left to the developer.

This rest of this paper presents our experiences in two case studies that approach biological model development using methods taken from software engineering that promote low coupling and high cohesion, and aim towards extensibility. We first present a syntax of an ideal object-oriented (OO) language for mathematical modelling. Throughout this paper the language is used alongside UML diagrams in order to convey the key concepts at a high level. Next we present the first case study that discusses generic modules and demonstrate their efficacy for enabling modularity in cardiac modelling. We then go on to introduce an OO modelling framework for implementing hybrid multi-scale models of vascular tumour

growth. We use examples from this framework to exhibit how class inheritance could be used within an OO biological modelling language to enable effective code reuse. Finally, we bring the two approaches together and discuss the benefits of combining inheritance with generics.

2. A CORE PHYSIOLOGICAL MODELLING LANGUAGE

In this section, we define the syntax of a simple OO language that describes several important properties that we feel a physiological modelling language should incorporate. Such a core language would have similar syntax and semantics to the restricted subsets of Java often used for research purposes with extensions for modelling continuous, deterministic mathematical models.

OO programming (OOP) involves describing a system in terms of encapsulated objects and the interactions between them through well-defined interfaces. OO design is often used in software engineering when designing large-scale reusable systems, with some of the key tenets being encapsulation, abstraction, modularity and reuse. Inheritance is a central feature of OOP which allows classes to reuse code defined within another class. It is commonly used to enable code reuse within an OO framework and to allow inherited objects to be substituted for one another. Generics enable modellers to operate on encapsulated objects in an abstracted manner, enabling reuse and substitutability, as commonly seen in C++ templates. Both inheritance and generics, applied appropriately, can be used to enable greater code reuse and extensibility when structuring complex biological models.

2.1. Core Language

Our OO modelling language supports value declarations, functions, conditionals and looping constructs for logical and control-flow. To support mathematical modelling we include explicit support for ODEs and facilitate linking to PDEs solvers for certain classes of problems, thus enabling the creation of multi-scale continuous, deterministic models.

We utilise the typed object system of the core language to structure our models into classes consisting of collections of related value and component definitions that form some logical (potentially biological) grouping. Classes may be imported through an import declaration and their definitions accessed using the dot-notation as found in Java. Objects have type signatures used to specify the module requirements and enable safe composition and reuse of its encapsulated components. Finally the language defines templates that may be used to parameterise the classes used within an object and the interfaces they implement in a generic manner. A simplified overview of language requirements of our DSL are shown in Figure 1.

```
fileDef     ::= ( importStmt | interfaceDef | classDef )*
importStmt  ::= import id [ as id ]

interfaceDef ::= interface id {interfaceVar*}
interfaceVar ::= id :: typeDef* -> typeDef*
typeDef     ::= double | bool | [typeDef]

classDef    ::= class id [ < (id :: id)* > ]
                [ extends id | implements id] {classStmt*}
classStmt   ::= funcDef | valDef | odeDef | arrayDef
funcDef     ::= function f (id*)
                { funcStmt*; return expr}
funcStmt    ::= valDef | ifStmt | arrayDef | odeDef

valDef      ::= valid = expr
ifStmt      ::= if expr then expr* [ else expr*]
arrayDef    ::= val id [number]=[expr*]
odeDef      ::= ode id {init:number} = expr

expr        ::= e * e | e / e | e > e | ···
e           ::= (expr) | number | boolean | time
                | ifStmt | id . e′ | e′
e′          ::= f (expr*) | id [number] | id
```

Figure 1. Simplified EBNF for Physiological Modelling DSL (assumes presence of basic operators such as *id*).

Section 3. examines the use of generics, interfaces and modular programming, utilising our core language to structure complex models. Section 4. utilises the OO support, inheritance and polymorphism within the language to reuse models within a structured model framework. The examples in the remainder of this paper are given using the syntax displayed in Figure 1 which we explore through case studies.

3. GENERICS

Our first project has explored the use of generics. Generic programming 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.

The work in this section is based on an independent DSL we have created and implemented for high-performance simulation of single-scale biological models [4]. However, the examples and general modelling concepts are given in terms of our core OO modelling language described in the previous section.

```
class Parameters {
  val Cm = 1
  val E_R = -75
  val i_Stim = if (time > 10 and time < 10.5)
    then 1 else 0 }

import Parameters as Pa

class Model {
  function membrane() {
    ode dV {init : 0} = (-Pa.i_Stim
      + i_L) / Pa.Cm
    val i_L = 0.3 * (V - E_L)
    val E_L = Pa.E_R + 10.613 } }
```

Figure 2. Including Classes.

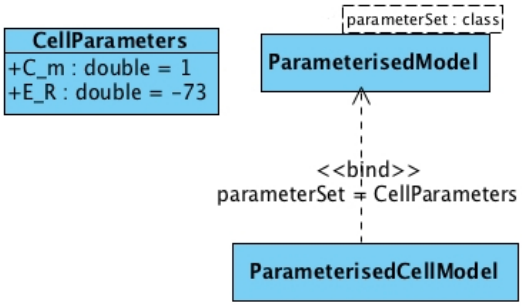

Figure 3. UML diagram which indicates how Model takes a class parameter, this binding relationship indicates the assignment of a class to a template parameter, which in turn generates a new concrete model class from the template.

3.1. Generic Classes

Utilising our core language described in Section 2., we demonstrate in Figure 2 a simple example that defines a model used to calculate the voltage across a cell membrane, as found in biological electrophysiological models [5]. Constant values, such as `Cm` and `E_L`, may be defined from the results of mathematical expressions. Functions, such as `membrane`, are provided as a means to group, abstract and parameterise repeated computations. Finally we demonstrate how biological models may be structured into reusable components that may be shared and reused between models utilising the class import system. Within our work in cardiac modelling we utilise objects to represent an ion channel component of a cell model, or the model parameters.

Utilising objects to group related definitions enables modularity and code reuse, however the imported objects and abstractions 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. Figures 3 and 4 indicate how generics may be used both at a high level and syntactically within our DSL to achieve this goal.

Here, we have introduced a form of interface-based model

```
class Parameters {
  val Cm = 1
  val E_R = -75
  val i_Stim = if (time > 10 and time < 10.5)
    then 1 else 0 }

class Model <Pa :: Parameters> {
  function membrane() {
    ode dV {init : 0} = (-Pa.i_Stim
      + i_L) / Pa.Cm
    val i_L = 0.3 * (V - E_L)
    val E_L = Pa.E_R + 10.613 } }
```

Figure 4. Cardiac model example utilising generic classes to describe model parameters.

construction that enables the specialisation of reusable model components and facilitates several modelling use cases through the modification of parameters and equations used within a model. Generics enables the substitutability of compatible components, yet may be specialised by providing a type signature/interface that a generic component must implement. Objects may be parameterised by other objects leading to the creation of complex, specialised objects via a form of aggregation. They may be implemented as a compile-time construct in a similar manner to templates in C++, or at runtime as in C#, allowing type-safe substitutability of components with differing performance trade-offs. Generics were implemented in our previous DSL via parameterised modules [4], using an implementation influenced by ML functors [6] and C++ templates [7].

3.2. Application of Generics

Our core language may be used to model mathematically many classes of biological systems at differing scales, such as cell-cycle models and signalling pathways. In [8] we focus on cardiac electrophysiological models as they are well-known and highly-developed [5]. We utilised generics to develop a range of models to investigate the rapid, collaborative development, modification and reuse of models in this DSL.

At a high-level the excitation process of a cardiac cell is governed by the flux of charged ions. The results from a sample simulation are given in Figure 5. Within our cardiac modelling framework the models were split into classes to describe the flow of charged ions across the cell membrane caused by ion channels. We separate our cell models into a collection of reusable and substitutable ion channel objects as most cardiac research data and experiments occur at the ion channel level.

This abstraction can be expressed utilising generics within our object system, multiple objects representing the same ion channel may be easily created, coexist and be substituted within ever more complex and existing models. A type-signature suitable for describing ion channel objects within cell models was created that all ion channel objects must im-

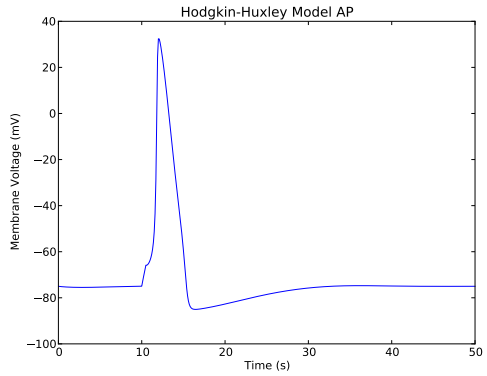

Figure 5. Action potential of Squid Giant-Axon from Hodgkin-Huxley Model [9].

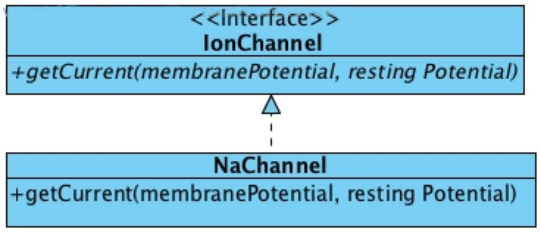

Figure 6. UML diagram indicating how a class implements an interface.

plement. When applying generic objects, we specify the interface a generic object requires, ensuring that only valid ion channel objects may be utilised as generic parameters. This could be checked at compile-time by the type system.

```
interface IonChannel {
  getCurrent :: (membraneV, restingV) -> current }

class NaChannel implements IonChannel {
   getCurrent (membraneV, restingV) { ... } }

class Model <NaChannel :: IonChannel> { ... }
```

This relationship is demonstrated by the UML diagram in Figure 6.

3.3. Results

Utilising generic objects to model ion channels within cardiac systems enables more advanced or detailed representations of ion channels, perhaps representing newer experimental data, to be placed into existing models. Detailed representations of ion channels may be utilised to investigate the functionality of a particular ion channel or more simple representations may be used to reduce the computational complexity of a specific model [10]. Performing such abstracted model reuse through the application of compile-time generics enables specialisation without incurring a run-time performance penalty.

Generic objects may also be used to alter models for particular simulations, e.g. to perform sensitivity analysis of the model to parameter fluctuations or to alter equations, e.g. to model the ion channel changes caused by mutation or drug block [11]. As an example, type-correct generic adapter objects may be created that simply reduce the current of an generic ion channel parameter by 50%, enabling investigations into drug block in an abstracted manner.

Objects created in the way 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 encourage high cohesion, grouping biological function together where need be.

4. CLASS INHERITANCE IN OBJECT ORIENTED MODELLING

Our second project involves the development of an OO modelling framework for implementing hybrid and multiscale models of vascular tumour growth. The focus of development has been to apply software engineering techniques that allow models developed in our framework to be highly reusable and extensible. The development strategy adopted has been to reverse-engineer a family of related models published by Tomás Alarcón and collaborators [12, 13, 14, 15, 16, 17] in order to extract and abstract the common methodologies and data structures involved in the development of vascular tumour growth models. The OO framework has been developed based on these abstractions, and a functioning implementation of the framework is being developed in C++ [18, 19]. In this section we use examples taken from the implementation of this framework to outline the benefits of our second abstraction mechanism, inheritance. We also demonstrate how inheritance can be used to structure the dynamic behaviour of an entity such as a collection of cells.

Inheritance in OOP is, in part, a mechanism by which to prescribe appropriate relationships between classes of objects in a model system. More specifically, inheritance describes an *is-a* type relationship between classes of objects. In this way, if used properly, inheritance can improve the understandability of model code by helping to minimise the conceptual distance between code and the real-world system which the code models. For instance, in the models which motivate our work we consider several different sub-cellular models. We consider one model which instructs cells to divide after a fixed duration (Figure 7b: `FixedDurationSubCellularModel`), a model which considers the explicit time evolution of several cell-cycle proteins and is dependent on the local concentration of oxygen (Figure 7b: `Alarcon2005SubCellularModel`) and a more phenomenological model which consid-

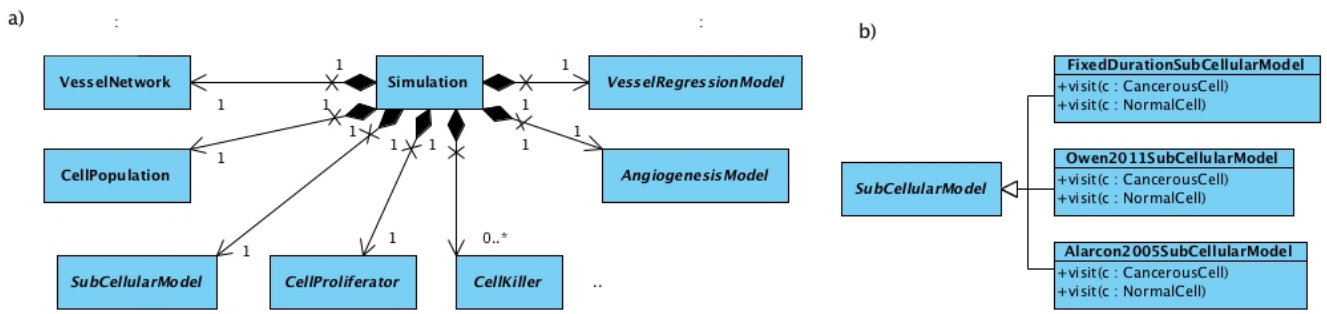

Figure 7. This class diagram (a) shows some of the major classes involved in a simulation of vascular tumour growth in our modelling framework. The abstract classes in (a) are replaced by concrete classes at runtime. For example, objects which are of type `FixedDurationSubCellularModel`, `Alarcon2005SubCellularModel` or `Owen2011SubCellularModel` in (b) may be provided by a modeller where a `SubCellularModel` object is expected by a `Simulation` object. The `visit` methods, shown in (b) are necessary in order to implement the visitor pattern adopted in our framework.

ers the time evolution of a simple cell-cycle phase (Figure 7b: `Owen2011SubCellularModel`). These models are described in full in [12], [13] and [20], respectively. In our modelling framework the classes `FixedDurationSubCellularModel`, `Alarcon2005SubCellularModel` and `Owen2011SubCellularModel` all inherit from the class `SubCellularModel` since they *are a type of* sub cellular model. Similarly, we define relationships between various biological entities. For example, in our framework we define the classes `CancerousCell` and `NormalCell`, which inherit from another `Cell` class, which in turn inherits from a `BiologicalEntity` class. In this way, appropriate relationships, which mimic real-world relationships, are defined between some classes of objects in our modelling domain.

As alluded to above, class inheritance also provides a powerful mechanism by which code may be reused. By inheriting from an existing class, such as our `Cell` class, the subclasses (`CancerousCell` and `NormalCell`) automatically obtain all of the functionality (operations) and attributes defined in the `Cell` class. Furthermore, new attributes and operations may be defined in the subclasses. Thus inheritance allows new classes to be defined in terms of incremental variations of more basic and abstract classes.

Inheritance also enables run-time *substitutability*. In our example this means that objects of type `FixedDurationSubCellularModel`, `Alarcon2005SubCellularModel` or `Owen2011SubCellularModel`, if defined appropriately, may be provided at run-time in any context where an object of type `SubCellularModel` is expected without affecting the correctness of the model implementation. This property of an OO system imposes strict rules on class inheritance, namely that any class inheritance hierarchy must follow the principle of type conformance [21].

Inheritance exploits what is known as *subclass coupling*, where the subclass is connected to the superclass, but the superclass is not connected to the subclass. This means that other objects only need to know about the parent as through substitutability the outside client can treat them all the same. Moreover, through subclass coupling, changes in the superclass inherently causes changes in the subclass. This helps to eliminate redundancy in subclasses.

Dynamic polymorphism, or overriding, is another OO technique which permits programmers to redefine the implementation of inherited operations in a subclass. Calls to an operation which have been overridden in a subclass generally provide some sort of modified functionality. This technique therefore works hand-in-hand with the substitutability property of class inheritance to allow modellers to create model implementations which are both malleable and extensible.

4.1. Application of Inheritance

Vascularisation of a tumour marks the transition of that tumour from being essentially harmless to increasingly invasive and eventually fatal [22]. During their avascular growth phase, tumours are typically manageable and relatively harmless. They are limited in size due to the lack of nutrients available to fuel their proliferation. However, once a tumour grows too large to be supported by the oxygen supplied from existing vasculature, the tumour cells secrete various angiogenic growth factors, known collectively as tumour angiogenesis factors (TAFs). TAFs diffuse through the healthy tissue surrounding the tumour and upon reaching a blood vessel will stimulate vascular growth towards the tumour. This process is know as sprouting angiogenesis. Over time the vascular network evolves to a state in which the level of nutrients and oxygen in the tissue surrounding the tumour have increased sufficiently to allow for further growth of the tumour.

In order to gain comprehensible insight into such a complex and multiscale phenomenon and, in particular, to expose cross-scale coupling mechanisms, multiscale mathematical models are commonly constructed. Modelling tumour growth in this way involves the integration of a number of biological models concerning different processes which may occur on different time and length scales. Each model may also integrate different mathematical methodologies to adequately model these processes [23]. Figure 7a illustrates how the simulations of such multiscale models are constructed in our modelling framework. An instance of a `Simulation` class contains references to various model components. These components may be partitioned into physiological-based classes (e.g. `CellPopulation`) and a number of physics- or rule-based classes which dictate the behaviour of those physiological classes.

Using these model components the `Simulation` class co-ordinates the events which occur during a model simulation, whilst delegating the responsibility of carrying out specific tasks to other classes. For example, a `Simulation` contains a reference to a `SubCellularModel` object. Different types of subcellular models are implemented in the concrete subclasses of the `SubCellularModel` class, as described above. Due to substitutability, an instance of one such subclass may be assigned to the `SubCellularModel` reference at run-time, providing the implementation as desired by a modeller for a particular realisation of a model. Similar substitutions may be made for the other aspects of the system behaviour shown in Figure 7a, e.g. cell movement, angiogenesis, etc. By exploiting class inheritance and substitutability in this way, our framework allows the construction and implementation of a wide array of different models of vascular tumour growth. The structure of the framework in Figure 7 is actually an example of extensive use of the strategy pattern. Further details of this and other design patterns implemented in our framework may be found in [18].

Figure 8 illustrates how a new subcellular model may be defined within our modelling framework. After compilation, this new sub cellular model may again be substituted at run-time where the `SubCellularModel` class is defined within the `Simulation` containment hierarchy (Figure 7a). Thus, by exploiting the substitutability property of inheritance, our framework allows scientists and modellers to rapidly develop and implement novel models within the constraints of our framework.

4.2. Results

Our use-case driven design process has yielded an OO framework which allows a particular family of vascular tumour growth models to be extended both intuitively and with relative ease. In large part, this extensibility has been achieved by applying OO design patterns which exploit the substitutability property of class inheritance hierarchies.

```
class NewOxygenDependentModel extends SubCellularModel {
  ...
  // When p53 concentration exceeds threshold, kill cell
  if (Cell.getChemicalConcentration(Chemical.p53) >
      Parameters.normalCellDeathThresholdConcentration)
                Cell.kill();
  // When cycle time exceeds divide time,
  // flag cell to divide.
  if (Cell.cellCycleTime >=
     Parameters.cellDivideTime) {
       // Flag cell to divide.
       Cell.divide();
       // Reset the cell cycle time of this cell.
       Cell.setCellCycleTime(Cell.cellCycleTime -
                  Parameters.cellDivideTime);
                          }
}
```

Figure 8. Pseudocode fragment illustrating how our OO framework extends the abstract `SubCellularModel` class, allowing model developers to provide their own concrete implementation.

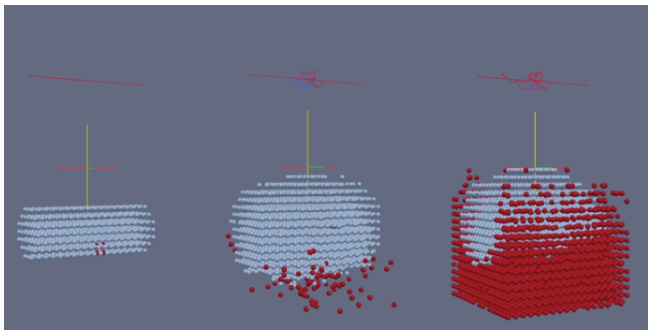

Figure 9. Progressive time points from a 3-D simulation of vascular tumour growth. Cell distributions are shown separately to the local evolving vasculature. Cancerous cells are shown in red and normal cells in blue.

Figure 9 shows three time points of a simulation implemented within our framework, which coupled several of the submodels originally described by Alarcón and co-workers. We are currently working on extending our framework to also enable the implementation of continuum models of vascular tumour growth. Results of this work will be presented in future publications.

5. GENERICS AND INHERITANCE

In this section we demonstrate how inheritance may be used to abstract out common functionality, enable code reuse and further constrain the types of arguments that may be passed to objects as generic parameters. Meanwhile, generics are used to provide substitutability without incurring a performance penalty. Both techniques orthogonally help model development by increasing cohesion and decreasing coupling.

```
interface IonChannel {
    getCurrent :: (membraneV, restingV) -> current }

class BaseNaChannel implements IonChannel {
     getCurrent(membraneV, restingV) { ... } }

class NaChannel_N62 extends BaseNaChannel { ... }

class NaChannel_BR77 extends BaseNaChannel { ... }
```

Figure 10. Extending Cardiac electrophysiology modelling framework to inheritance.

```
class Model <NaChannel :: BaseNaChannel> { ... }
```

Figure 11. Generic Cell Model.

From a cardiac modelling perspective, we can use inheritance to define common functionality for ion channels that are then extended in later, more complex channel models (mirroring the real life development of such models), as is shown in Figure 10.

Generics provide the ability to alter the models and parameters in a standardised manner. By exploiting the substitutability property of inheritance in combination with generics, we are able to further restrict a modellers intentions. For example, in Figure 11 the generic variable `NaChannel` is of type `BaseNaChannel`. Through substitutability, at compile time `NaChannel_N62` [24] or `NaChannel_BR77` [25] may then be provided as a generic parameter to the `Model` class in a type-safe way. The compiler ensures that the signatures of any class provided as a parameter are consistent with the `Model` class specification, and should a modeller attempt to provide an inappropriate type as a parameter a compile time error is generated. This therefore ensures that, at runtime, models are correctly defined and implemented.

We utilise both inheritance and generics to abstract behaviour and provide substitutability of reusable model components, as shown in Figures 12 and 13. This benefits the

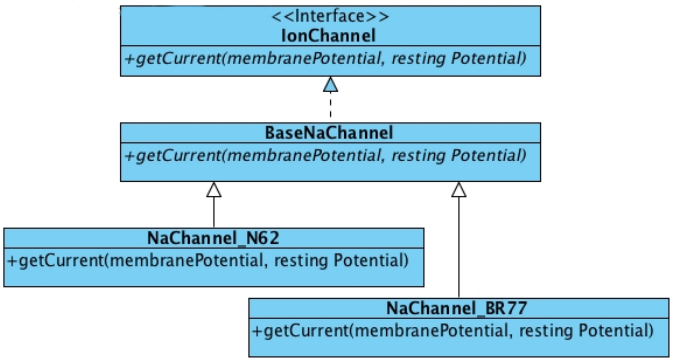

Figure 12. Use of inheritance within Cardiac electrophysiology model framework.

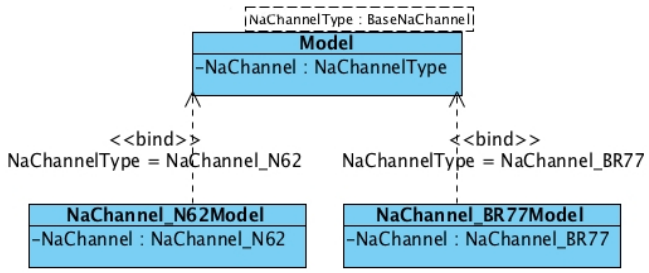

Figure 13. Use of generics within Cardiac electrophysiology model framework, as in Figure 3 we indicate the application of a template using a binding relationship.

modellers by allowing for model reuse, type-safe component-driven development, and type-checking of model composition. Further work could employ type wildcards, as implemented in Java generics, further restricting the implementations allowed by the compiler and allowing type-safe model composition.

6. DISCUSSION

In this paper we have looked at key features that a DSL for physiological modelling would ideally provide for enabling modularity, encapsulation, reuse and extensibility. We have built upon an existing DSL in order to demonstrate how generics may be used to facilitate the creation of a reusable repository of cardiac model components parameterised at the ion channel level. A range of published models, parameterised by their ion channels, have been created and validated utilising this technique. We introduced an OO framework for implementing multi-scale models of vascular tumour growth and used examples from models implemented in this framework to exhibit how class inheritance may be used to enable run-time substitutability of various model components. This enables modellers to easily customise and extend existing models in an intuitive way. Finally we showed that, when combined, these techniques allow model designers to pick and choose suitable abstractions to ensure that their codes may be maintained and extended in a well-structured and type-checked manner.

Both of these abstraction techniques independently increase cohesion and decrease the coupling of modules. However, there is a cost involved with utilising these techniques; modellers have to spend time and effort designing their code with such abstract architectures in mind in order to reap the benefits. We feel that it is an endeavour that allows many implicit understandings of the modeller to be captured, and one that is well suited to teams working together. Even though the initial investment of structuring one's designs in a modular OO fashion might not be immediately academically relevant for one's particular field, the ability to rapidly modify code enables a more adaptive approach to model creation.

We have demonstrated interoperability with a weak seman-

tic alignment at the code level; through interfaces, subtype inheritance and generic instantiation. Future work will consider the use of ontologies to facilitate a Model Driven Approach using UML's meta-object facility and corresponding tool support.

REFERENCES

[1] S.Y. Diallo *et al.* Understanding interoperability. In *SpringSim (EAIA)*, pages 84–91. SCS/ACM, 2011.

[2] M. Hucka *et al.* Evolving a lingua franca and associated software infrastructure for computational systems biology: The systems biology markup language (SBML) project. *IEE Systems Biology*, 1(1), June 2004.

[3] C.M. Lloyd, M.D. Halstead, and P.F. Nielsen. CellML: its future, present and past. *Progress in Biophysics and Molecular Biology*, 85(2-3):433 – 450, 2004. Modelling Cellular and Tissue Function.

[4] M. Gill, S. McKeever, and D. Gavaghan. Modular mathematical modelling of biological systems. In *SpringSim (TMS-DEVS)*. SCS/ACM, 2012.

[5] D. Noble and Y. Rudy. 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, June 2001.

[6] F.Z. Nardelli. Objective Caml module system. Technical report, 2005.

[7] G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *ACM SIGPLAN Notices*, volume 41, pages 295–308. ACM, 2006.

[8] M. Gill, S. McKeever, and D. Gavaghan. Modules for reusable and collaborative modelling of biological mathematical systems. In *21ST IEEE International WETICE Conference (WETICE-2012)*, 2012.

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

[10] T. O'Hara *et al.* Simulation of the undiseased human cardiac ventricular action potential: model formulation and experimental validation. *PLoS Computational Biology*, 7(5):e1002061, May 2011.

[11] V. Iyer, R. Mazhari, and R.L. Winslow. A computational model of the human left-ventricular epicardial myocyte. *Biophysical journal*, 87(3):1507–25, September 2004.

[12] T. Alarcón, H.M. Byrne, and P.K. Maini. A cellular automaton model for tumour growth in inhomogeneous environment. *Journal of Theoretical Biology*, 225(2):257–274, 2003.

[13] T. Alarcón, H.M. Byrne, and P.K. Maini. A multiple scale model for tumor growth. *Multiscale Model Simulation*, 3(2):440–475, 2005.

[14] T. Alarcón *et al.* Multiscale modelling of tumour growth and therapy: The influence of vessel normalisation on chemotherapy. *Computational and Mathematical Methods in Medicine*, 7(2-3):85–119, 2006.

[15] R. Betteridge *et al.* The impact of cell crowding and active cell movement on vascular tumour growth. *Networks and heterogeneous media*, 1(4):515–535, 2006.

[16] M.R. Owen *et al.* Angiogenesis and vascular remodelling in normal and cancerous tissues. *Journal of Mathematical Biology*, 58(4-5):689–721, April 2009.

[17] H. Perfahl *et al.* Multiscale modelling of vascular tumour growth in 3D: the roles of domain size and boundary conditions. *PLoS ONE*, 6(4):(17 pages), 2011.

[18] A.J. Connor *et al.* Object-oriented paradigms for modelling vascular tumour growth: a case study. *IARIA SIMUL 2012*, pages 74–83, November 2012.

[19] D. Johnson, A.J. Connor, and S. McKeever. Modular markup for simulating vascular tumour growth. In *5th IARWISOCI - The TUMOR Workshop*, pages 53–56, 2012.

[20] M.R. Owen *et al.* Mathematical modelling predicts synergistic antitumor effects of combining a macrophage based, hypoxia-targeted gene therapy with chemotherapy. *Cancer Research*, 71(8):2826–2837, April 2011.

[21] Meilir Page-Jones. *Fundamentals of object-oriented design in UML*. Addison-Wesley, 2002.

[22] J. Folkman *et al.* Tumor angiogenesis: Therapeutic implications. *New England Journal of Medicine*, 285(21):1182–1186, 1971.

[23] T.S. Deisboeck *et al.* Multiscale cancer modeling. *Annual Review of Biomedical Engineering*, 13(1):127–155, August 2011.

[24] D. Noble. A modification of the Hodgkin-Huxley equations applicable to Purkinje fibre action and pacemaker potentials. *The Journal of Physiology*, 160(2):317, 1962.

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