Application of software engineering methodologies to the development of mathematical biological models

Mandeep Gill

Keble College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2013

This thesis is dedicated to my parents
for their ceaseless help and encouragement along the way.

Acknowledgements

I would like to thank Steve McKeever and David Gavaghan for their encouragement and guidance as my supervisors. With Steve's knowledge of software engineering and Dave's insight into application to biological modelling, I never felt lost during the research process. They were both particularly helpful during some of the more difficult parts of my research.

I would like to thank the Oxford Doctoral Training Centre, and Dave for accepting me onto the programme. I learnt a great deal about computational biology during the year, whilst being introducing to a wide variety of fields and different ways of thinking. I picked up many research and professional skills, and made several good friends along the way.

My research was funded by Engineering and Physical Sciences Research Council and I would like to thank them for their support.

Thank you to my collaborators from both the modelling and software engineering fields, including the Computational Biological research group, their input greatly aided my research. Further acknowledgements are necessary for: Ciara Dangerfield in particular for her advice and input on stochastic modelling and patiently answering any mathematical questions I had; John Walmsley, Tamas Szekely, and Phil Gemmell for their feedback on mathematical modelling; and Jonathan Cooper and Anthony Connor for their advice on several software development and computational modelling issues.

I would also like to thank the researchers in Room 366A, Eoin and Tom especially, for providing a sounding board for ideas and several sanity preserving distractions.

Finally I would like to thank my family for all their support, faith and interest.

Application of software engineering methodologies to the development of mathematical biological models

Mandeep Gill

Keble College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Trinity 2013

Mathematical models have been used to capture the behaviour of biological systems, from low-level biochemical reactions to multi-scale whole-organ models. Models are typically based on experimentally-derived data, attempting to reproduce the observed behaviour through mathematical constructs, e.g. using Ordinary Differential Equations (ODEs) for spatially-homogeneous systems. These models are developed and published as mathematical equations, yet are of such complexity that they necessitate computational simulation. This computational model development is often performed in an *ad hoc* fashion by modellers who lack extensive software engineering experience, resulting in brittle, inefficient model code that is hard to extend and reuse.

Several Domain Specific Languages (DSLs) exist to aid capturing such biological models, including CellML and SBML; however these DSLs are designed to facilitate model curation rather than simplify model development. We present research into the application of techniques from software engineering to this domain; starting with the design, development and implementation of a DSL, termed *Ode*, to aid the creation of ODE-based biological models. This introduces features beneficial to model development, such as model verification and reproducible results.

We compare and contrast model development to large-scale software development, focussing on extensibility and reuse. This work results in a module system that enables the independent construction and combination of model components. We further investigate the use of software engineering processes and patterns to develop complex modular cardiac models.

Model simulation is increasingly computationally demanding, thus models are often created in complex low-level languages such as C/C++. We introduce a highly-efficient, optimising native-code compiler for *Ode* that generates custom, model-specific simulation code and allows use of our structured modelling features without degrading performance.

Finally, in certain contexts the stochastic nature of biological systems becomes relevant. We introduce stochastic constructs to the *Ode* DSL that enable models to use Stochastic Differential Equations (SDEs), the Stochastic Simulation Algorithm (SSA), and hybrid methods. These use our native-code implementation and demonstrate highly-efficient stochastic simulation, beneficial as stochastic simulation is highly computationally intensive. We introduce a further DSL to model ion channels declaratively, demonstrating the benefits of DSLs in the biological domain.

This thesis demonstrates the application of software engineering methodologies, and in particular DSLs, to facilitate the development of both deterministic and stochastic biological models. We demonstrate their benefits with several features that enable the construction of large-scale, reusable and extensible models. This is accomplished whilst providing efficient simulation, creating new opportunities for biological model development, investigation and experimentation.

Contents

Chapter 1

Introduction

This thesis details our research into the application of common software engineering practices to the domain of biological mathematical modelling. This includes the design and implementation of a domain-specific language (DSL), termed *Ode*, for modelling and simulating biological systems described by both continuous and discrete mathematical models. We focus on cardiac electrophysiological models as they are well-studied with a detailed history and present several modelling challenges that we are keen to address with our DSL.

Creation of the *Ode* DSL acts as an overriding, unifying research goal that touches upon and encompasses several aspects of software engineering that we believe are vital to the development and reuse of models in a systematic and correct fashion. This includes research into modular programming and development, model verification including type- and units-checking, and high-performance simulation through an optimising compilation pipeline that makes use of custom, model-specific code-generation. We believe that this research will benefit domain experts and modellers performing model development, allowing them to reason and understand a model's behaviour and obtain repeatable results with confidence that may be related back to experimental data. We believe these goals can be reached by applying software engineering practices used to successfully develop large-scale software to the domain of computational model development.

From a biological modelling perspective our primary focus is capturing the behaviour of ion channels in cardiac cells. Through linking their behaviour into chemical kinetics, we provide multiple modelling mechanisms for ion channels, including deterministic, stochastic, continuous, discrete and hybrid combinations that may all be simulated using computationally efficient

methods. These methods may be applied to model many other biological systems. We envisage this work enabling modellers to perform more detailed investigations into ion channel function from both deterministic and stochastic perspectives and investigate their effect on the cardiac action potential. This may facilitate investigations into model properties and several cardiac cycle disorders [20, 26], with the end goal being clinical use in a predictive medical setting.

In this chapter we present the aims of our research and the motivating factors for this work within our particular domain of cardiac modelling and how our work will aid research within this area. We discuss the areas of research undertaken and summarise the thesis with an overview of each chapter.

1.1 Aim

The aim of this thesis is to investigate aspects and techniques from software engineering that we can apply to the computational simulation of mathematical models of biological systems. We focus in particular on single-cell cardiac electrophysiological systems as this is perhaps one of the most mature area of systems biology modelling. A large body of experimental data is available and many models have been created over the last half-century [107]. These have helped to elucidate cardiac function and provided insight into certain disorders and to drug behaviour [19, 20, 26, 75]. Hence the domain provides an array of validated models that can benefit hugely from applied software engineering techniques.

For instance, cardiac models can be difficult to read and understand when developed in general-purpose languages. This approach is typically far removed from the original mathematical definition and introduces implementation language semantics into the model code, potentially obfuscating the implementation and hindering model verification. They may also be hard to maintain and extend, as the model is often created in a non-modular manner that does not reflect the underlying biological structure. This unstructured code tends to be tightly coupled to the simulation system, further hindering reuse and was a primary motivation for the initial development of modelling DSLs such as SBML and CellML [52, 73, 93].

We hope that techniques from software engineering, used to construct large-scale software, may be applied to these models to demonstrate the benefits of code structure, reuse and extensibility during the model development process. Our work may also be applied to many other biological modelling domains that exhibit complex interactions and demonstrate potential for model reuse and extensibility, such as metabolic pathways, gene regulatory networks or cell cycles [103, 135].

Our research starts by investigating DSLs for modelling biological system behaviour from a mathematical perspective that may be computationally simulated. This overarching theme prompts our research into modelling constructs, model verification through type- and units-checking, high-performance simulation and language features to enable modularity and model reuse. Our concern is that existing biological modelling DSLs are far removed from the model development process. It is currently difficult to capture knowledge within these DSLs — they are hard to extend

and reuse, and are complicated for biologists and modellers to use. For instance, CellML cellular modelling DSL was designed primarily to curate existing, validated, models [93]. As a result it has limited computational power with no support for the abstraction of model computations, decreasing the reusability of model components. Furthermore, the model structure is defined through the manual connection of disparate components contained elsewhere in the model file, increasing fragility and decreasing model cohesion. Modularity within the DSL is equivalently limited, requiring the static importing of model components that must be manually linked whilst taking care to match the correct names of values in the process. We do not consider it an effective, operational DSL; rather a markup language more suited to the curation of existing models than model development and *in silico* experimentation.

Our conjecture is that creating a DSL to capture the manner in which modellers develop equations and simulate models traditionally, typically through MATLAB or low-level C code, yet based on a sound computational foundation, is a stronger approach. This allows us to utilise aspects of modern programming language theory and design to advance the model development process to the benefit of modellers. Development of the DSL presents several research opportunities, within the domains of language design, efficient implementation, and software engineering including collaborative model development and model reuse. We detail several of these research areas undertaken within our work in the following section.

Software engineering itself is an engineering discipline concerned with all aspects of software production [129], as opposed to general 'coding' often seen in computational modelling. The discipline evolved as a response to several issues — large software systems were typically delivered late and lacking required functionality, cost more than expected, and were unreliable. Good software should deliver the desired functionality and performance to the user whilst remaining maintainable, dependable, and usable. Fundamental software engineering activities involve software specification, development, validation, and evolution.

Computational scientific methods focus on theory and fundamentals, however software engineering is also concerned with the practicalities of developing and delivering useful software. The study in [68] suggests that only by combining these approaches, both the 'science of programming' and 'engineering of dependability', we obtain domain models that distil the knowledge and techniques required to construct software correctly and practically within a field. As such we

aim to investigate the application of relevant aspects of software engineering, including testing, reusability, reliability, adaptability, and deployment, to the construction of computational models. This could lead to a set of design patterns that can be applied to generate reusable models that exhibit qualities similar to those seen in well-engineered software [4, 16, 50, 51].

Software is often tested to ensure it exhibits the correct behaviour, similarly in modelling we may utilise the concept of *functional curation* to combine a model with experimental data for testing, verification, and validation [32]. Functional curation provides a framework whereby models can, preferably automatically, be compared against previous models and experimentally observed behaviour during development, increasing confidence in a model. We aim to investigate methods for constructing models and simulations programmatically that allow functional curation to take place and facilitate the generation of reproducible results, in a manner influenced by agile, test-driven software development [74]. Such continuous checking, alongside further aspects from software engineering, has the potential to greatly impact systems biology modelling. We hope it leads to the formation of documented software engineering practices that may benefit the physiological modelling domain.

We believe that the application of software engineering techniques, and in particular the use of DSLs, will enable modellers to build robust extensible models using many advanced programming features in a simple manner. In turn this will allow the efficient construction of deterministic and stochastic models within several biological domains that exhibit high-performance simulation. In particular we feel that this work will enable the collaborative construction of reusable ion channel models for use in cardiac electrophysiological models. This thesis demonstrates the novel research undertaken and results obtained during investigation of these assumptions and hypotheses.

1.2 Areas of Research

Our research spans three major themes, covering computational model development, simulation performance, and applications to cardiac and biological modelling in general. We investigate the need for a more systematic, engineering based approach to model development, and present solutions through DSLs, design patterns and the application of software engineering concepts including modularity, extensibility, collaboration, and testing. We are interested in computationally-efficient simulation. This would enable modellers to create more complex models that accurately depict biological systems, investigate alternate modelling formulations, and reduce the time taken when conducting multiple simulations (as may be required for sensitivity analysis and stochastic methods).

Our final theme is the application of this research to the cardiac modelling domain, focussing on modelling ion channels from deterministic and stochastic perspectives. We look at how our research may be used to collaboratively build modular cardiac models that may be distributed through model repositories. This may help the understanding of cardiac behaviour in relation to experimental data, or be used within studies investigating novel pathologies or pharmacological agents, thus aiding the greater goal of predictive medicine [71]. The following list presents several individual pieces of novel research and results along these themes:

- Investigation into the process of model development, resulting in a domain modelling language for physiological systems. This DSL supports the construction of mathematical biological models comprised of deterministic and stochastic elements, and is designed for modeller ease-of-use and efficient simulation. It forms the base for our investigations into biological modelling design patterns. Furthermore we have used alternate DSLs to capture particular modelling sub-domains of interest, e.g. declaratively describing ion channel states, declaring units-of-measure in a model, creating modules and scripting simulations;

- Introduction of a type system and programmable units-of-measure system to the biological modelling domain that supports automatic inference and safe conversion of expressions to facilitate the creation of reliable models;

- Introduction of a programmable, typed module system to the biological modelling domain. This provides to the modeller flexibility, facilitates collaborative model reuse and enables

the creation of a range of model components that can be statically checked for correct composition and combined without increasing the simulation overhead. Modules may be parameterised by use of generic modules, enabling specialised reuse and modification within larger, more complex models;

- The use of the module system to investigate the application of software engineering concepts such as abstraction, modularity, and encapsulation to create an initial modular framework for constructing cardiac electrophysiological models based on the modular composition of reusable components. Modules can be created to represent independent ionic currents within a cardiac cell that can be replaced and interchanged within alternate models, mirroring the natural development of cardiac models and enabling phylogenetic studies into model development and reuse [105]. This framework may be extended and used by modellers to replicate and build new models by integrating and modifying existing components within new models, perhaps in response to newly obtained data [20]. It will enable the creation of bespoke models composed from a library of known components of differing complexities for specific modelling and simulation use-cases. In creating this framework we define structures and techniques useful for reusable model development, creating an initial pattern language [4, 16] for cardiac models. These patterns can be recognised and applied by future modellers to ease model construction;

- Use of an optimising compilation pipeline, late-binding and run-time specialised code-generation to enable model-specific efficient simulation of mathematical models. This includes an extensible optimisation stage that contains a novel auto-vectorisation mechanism that drastically increases the performance of computationally expensive mathematical functions. Efficient simulation may ease investigations that require multiple simulation runs, such as parameter estimation and sensitivity analysis, and benefit generation of multiple stochastic simulation trajectories, whilst potentially leading to the development of more complex and detailed models;

- Extension of the DSL to provide stochastic modelling constructs, based on the biochemical reactions, within a unified modelling language that provides high-performance discrete and continuous stochastic simulation. This may be used to create cardiac models that model

the stochastic effects on ion channel behaviour, e.g. due to drug block, in a hybrid-manner that can be simulated extremely efficiently. Such constructs may be used in conjunction with the modular framework to provide alternate stochastic implementations of existing ion currents within models.

1.3 Thesis Summary

This chapter has provided a high-level introduction to our research and outlined our domain of interest — mathematical biological modelling with a focus on cardiac electrophysiology. The overall aims of the research were discussed and the novel aspects of the work presented, from both computational and modelling perspectives. In the chapters that comprise this thesis we describe in detail the research areas, challenges, and results. They are discussed with respect to the aims and motivation of our research provided in Section 1.1.

Chapter 2 presents an overview of several important areas within our domain. This includes an introduction to biological modelling and a review of cardiac electrophysiological modelling, covering use of chemical kinetics to model ion channel function and presenting a family of related models used within our studies. We cover relevant aspects of software engineering, including the use of DSLs to model biological systems, and methodologies commonly used to structure and develop reliable, extensible software in a systematic fashion. Aspects of high-performance computing are examined, covering the software and hardware considerations required for efficient simulation.

The primary features of the *Ode* DSL are detailed in Chapter 3, presenting the syntax and semantics alongside several implementation details. We present a type system, that includes a programmable units-of-measure system, to check expression validity and enable reliable construction and composition of models. Finally we discuss the scriptable user-interface created to construct and configure simulations programmatically. It provides a separation between a model and its simulation protocol, encouraging repeatable results and facilitating functional curation.

Chapter 4 describes a module system that facilitates model abstraction and reuse through static and parameterised modules. The module system encourages collaborative model reuse and development, and provides mechanisms for abstraction and specialisation of model components. These may be used to create and build larger models from structured module interfaces and implementations, aiding the application of software engineering to biological models. We investigate design patterns useful for constructing reusable, extensible models; this is used to guide a case-study creating a modular cardiac modelling framework that demonstrates the module system and allows us to conduct several bespoke cardiac simulations.

Chapter 5 discusses the DSL implementation backend, including the construction of a high-performance native-code compiler. We discuss the simulation strategy, the numerical algorithms implemented, and present several high- and low-level optimisations including a novel auto-vectorisation scheme. The backend generates highly-efficient model-specific simulation code, as we demonstrate through the simulation of several cardiac models that exhibit up to twice the performance of equivalent C-based models.

In Chapter 6 we extend the DSL to support the stochastic modelling of biological systems, from continuous and discrete perspectives. We develop a higher-level DSL for declaratively modelling ion channels used within cardiac electrophysiological models that compiles into an *Ode* model. We perform several benchmarks of the implementation that demonstrate highly-efficient stochastic simulation, including a doubling of performance of the direct-SSA over an equivalent C-based solver, and perform a study into the modelling and simulation of the Hodgkin-Huxley model with stochastic subcomponents using our tools and construction patterns.

We conclude our thesis in Chapter 7 with a discussion of the research and results with respect to our initial aims. We present areas for future research that cover both computational and modelling aspects and list open areas and questions yet to be answered during our work. Examples of the language and models are given throughout the thesis, and several complete examples follow in Appendix D; many examples are based on the well-known Hodgkin-Huxley squid giant-axon model [69] for ease of comparison. To demonstrate the structure of *Ode* models within the thesis we utilise a diagrammatic notation from software design termed Unified Modelling Language (UML). Appendix D.2 contains a brief introduction to UML and describes our use of the notation to visually depict *Ode* model code. Fig. A.1 in the appendix presents a general overview of the *Ode* DSL described in this thesis, illustrating the flow of a model through several major implementation stages to create efficient, native, simulation code.

1.3.1 Supplementary Materials

Sections of this thesis have been published in several papers. [56] presented our initial work in crafting the *Ode* DSL and modelling features as discussed in Chapter 3. In [57] we presented the construction of the module system as described in Sections 4.2 and 4.3. In Section 4.1 we use the module system to examine modular patterns for model construction, this work was published as

a joint paper in [98]. Finally, the application of our module system to the modular construction and simulation of cardiac models, described in Section 4.4, was presented in [55].

We intend to produce several papers based on the research conducted in later chapters. We believe our novel work on an optimising model/simulation framework described in Chapter 5 is suited to publication within the high-performance and supercomputing domains. Furthermore, the stochastic modelling features and *Ion* DSL described in Chapter 6 are relevant to the computational physical chemistry and cardiac modelling fields.

Source code for the *Ode* and *Ion* DSLs and related packages may be downloaded from `http://bitbucket.org/mands/ode`.

Chapter 2

Literature Review

This chapter provides the background for our investigation into the use of software engineering methodologies for the mathematical modelling of cardiac electrophysiological systems and presents the need for a high-performance approach.

Biological systems may be modelled as discrete stochastic systems interacting and reacting at small spatial and temporal scales; however this may be abstracted for specific modelling use-cases or for computational efficiency to continuous stochastic and continuous deterministic approximations. These modelling approaches are used to simulate biochemical reactions, metabolic pathways, and the electrophysiological properties of cells. We wish to investigate the application of techniques used in software engineering to enable modellers to design such systems efficiently and in a reliable, reusable and verified manner.

The following sections describe the biological background and motivation for our work, detailing the issues with existing modelling and simulation methods. Cardiac cell biology is explained in tandem with various modelling approaches ranging from whole-organ models, to single cell models, to ion channel dynamics within the cell. The issues and limitations of these approaches from modelling and computational perspectives are discussed. We present an overview of cardiac electrophysiological function, and describe the construction and simulation of such models from a mathematical perspective, including an overview of ion channel function within the cell. Chemical kinetics are presented as a means to model and simulate ion channel dynamics, enabling deterministic, stochastic and hybrid simulation methods.

We consider the computational development of such models, and present an overview of software engineering practices commonly used to develop large-scale reusable and reliable

systems. This includes type systems, modularity, testing, software architecture and design patterns and, importantly, domain-specific languages (DSLs). DSLs may be used to ease the creation and representation of models and as a means to control and optimise the simulation of computationally complex systems. The general design and structure of modelling DSLs is discussed and a review of existing biological DSLs is given. The chapter concludes with an overview of the high-performance implementation and simulation of modelling DSLs, covering current systems and a discussion of software and hardware concerns.

2.1 Cardiac Modelling

We introduce cardiac cell modelling from a biological perspective, studying the cardiac action potential and its role in cardiac muscle contraction. This process is modelled at various spatial levels from the whole organ, to homogeneous cells, down to the modelling of cellular process such as ionic transfer. We look at a range of cardiac cell models, describing their function through ionic transfer using the Hodgkin and Huxley squid giant-axon model [69] as a reference. We present an overview of ion channel modelling, discussing the effect of an ion channel's behaviour on the action potential and its role in cardiac cycle irregularities.

2.1.1 Cardiac Electrophysiology

Cardiac muscle cells, illustrated in Fig. 2.1, are complex biological systems where various processes interact to generate electrical excitation, the action potential (AP) and contraction. The AP is a short-lasting event where the transmembrane voltage rapidly rises and falls with a consistent trajectory. During AP-generation, passive and facilitated flow of ions through membrane ion channels result in dynamically changing ionic concentrations and a varying transmembrane voltage [106, 124].

Fig. 2.2 depicts a typical AP as seen in a nerve cell, demonstrating the depolarisation, repolarisation and refractory periods. The AP is created by the depolarisation of the excitable membrane from its resting potential to above the activation threshold due to an influx of sodium ions, initiating the *all-or-nothing* AP. From this depolarisation, chloride and calcium channels are activated and an efflux of potassium gradually repolarises the membrane towards the resting potential.

This electrical stimulation propagates to adjacent cells and across the heart, causing regular, sequenced muscle contraction through what is termed *Excitation Contraction* (EC) coupling. The entry of calcium ions (Ca^{2+}) during the excitation phase of the AP triggers a sequence of events that leads to the contraction of cardiac muscle. Fluctuations within ion channel activations and timings are considered to be the cause of several cardiac disorders and arrhythmias [26, 44]. An example is Long-QT syndrome, a rare heart condition that leads to irregular heartbeats and is related to potassium channel inconsistencies during membrane repolarisation [26].

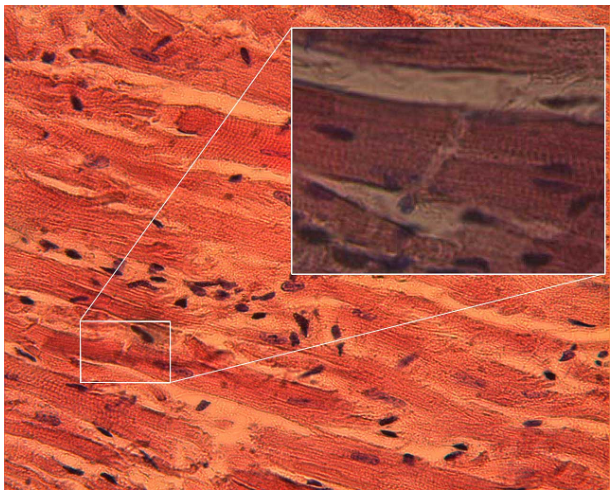


Figure 2.1: Image depicting a section of cardiac muscle; electrical signals propagate across the cells and trigger muscle contraction.

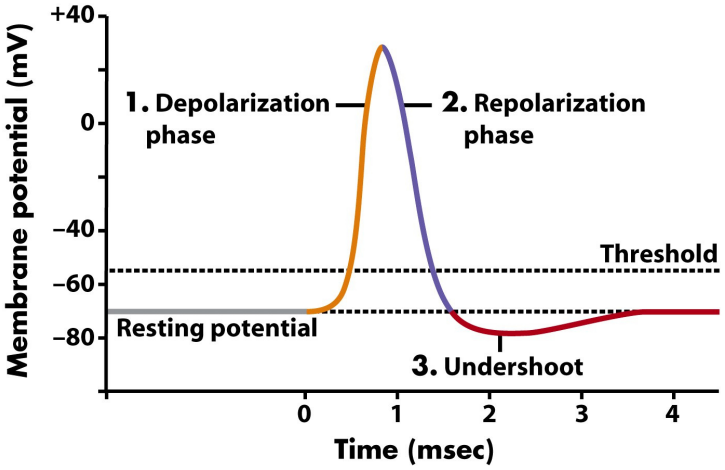


Figure 2.2: Annotated view of an idealised neuronal action potential, demonstrating the depolarising, repolarising and refractory periods.

2.1.2 Mathematical Cardiac Modelling

Cardiac cellular systems have been modelled mathematically using a variety of approaches, including discrete state models, ordinary differential equations (ODEs) and partial differential equations (PDEs). These models may have differing temporal and spatial scales, from individual molecular reactions, through cellular behaviour, to tissue/whole organ systemic behaviour [103]. The innate complexity of these models renders analytical solution impractical, and solutions must be obtained numerically via computational approaches. Numerical simulation is computationally intensive, and finding efficient means of simulation to do so is pressing concern.

Whole-organ electrophysiological models of the heart are usually modelled by PDEs in both spatial and temporal domains, typically using the bidomain equation [123]. Meshes obtained

from experimental data, usually MRI scans, are used to model the electrical activity and AP propagation through the organ. This requires simulating models of a single cell at each point on the mesh, often using ODEs.

The first mathematical model of a cardiac cell was created by Noble in 1962 [106], an ODE system that modelled the movement of ions across the excitable cell membranes of cardiac Purkinje fibres to generate and propagate action and pacemaker potentials from a continuous, deterministic viewpoint. This was based on the pioneering work by Hodgkin and Huxley in 1952 in modelling the behaviour of ion channels within the squid giant-axon during the AP [69]. Such mathematical models of cardiac ion channels are used to link the molecular processes that underlie ion channel function to the electrical activity of the whole cell. They may model ion channel function in several ways depending on the behaviour under observation, continuously and deterministically at a macro-level using ODEs, as the Noble and Hodgkin-Huxley models do; or stochastically, either continuously using stochastic differential equations (SDEs) or discretely using the Stochastic Simulation Algorithm (SSA), as discussed further in Section 2.1.4.

2.1.3 Cardiac Electrophysiological Models

Starting with the Hodgkin-Huxley (HH52) neuronal model of the squid giant-axon, we provide an overview and timeline of several related cardiac electrophysiological models that helps describe their evolution. The original HH52 model serves as the starting point for many examples in this thesis to demonstrate the electrophysiological modelling domain. Although a neuronal model it served as the basis for electrophysiological modelling within the cardiac domain. Table 2.1 lists these models with their most pertinent information relating to our usage.

Hodgkin-Huxley Model (HH52) (1952) [69]

The Hodgkin-Huxley model was the first major electrophysiological model; the result of a series of experiments investigating electric current flow through the surface membrane of the giant axon nerve fibre of a squid [69]. The authors developed a mathematical description of the membrane behaviour based on these experiments that accounts for the electrical excitation and conduction within the fibre. Within the model, transmembrane ionic currents combine to generate a cellular AP. Simulation of this model is depicted in Fig. 2.3, demonstrating the changes in membrane voltage, V_m, and current from the voltage-dependent K channel during an AP. As seen in the

Model	**Year**	**Species**	**Ion Channels / Pumps**	**State Variables**
Hodgkin-Huxley (HH52) [69]	1952	Squid	2	4
Noble (N62) [106]	1962	Mammalian	2	4
Beeler-Reuter (BR77) [9]	1977	Mammalian	4	8
DiFrancesco-Noble (DN85) [41]	1985	Mammalian	10	16
Luo-Rudy (LR1) [96]	1991	Guinea Pig	6	8
Luo-Rudy Dyanmic (LRd94) [94, 95]	1994	Guinea Pig	11	12
ten Tusscher *et al.* (TNNP04) [133]	2004	Human	12	16
Iyer *et al.* (IMW04) [75]	2004	Human	11	67
ten Tusscher-Panfilov (TP06) [134]	2006	Human	13	20
Grandi *et al.* (GPB10) [63]	2010	Human	13	38
O'Hara *et al.* (ORd11) [109]	2011	Human	15	41

Table 2.1: Electrophysiological models used within thesis, aside from the HH52 model they are all cardiac ventricular models. As experimental data has revealed new ion channel function, the models have become larger and more complex.

figure, and mentioned in Section 2.1.1, the presence of I_K results in the repolarisation of the membrane to its resting voltage.

Fig. 2.4 presents a schematic view of the HH52 model, alongside an electrical circuit interpretation that models ionic and 'leakage' currents as well as the capacitive effect of the membrane. This capacitance provides charge separation such that changes in V_m are due to charge displacement caused by the ionic currents. This effect is described by the following equation, where C_m is the membrane capacitance and I_{ion} the total transmembrane ionic current,

$$\frac{dV_m}{dt} = -\frac{1}{C_m} \cdot I_{ion}. \tag{2.1}$$

As seen in Fig. 2.4b, I_{ion} is the sum of three currents in the HH52 model: I_{Na} represents the depolarising sodium current, I_K the repolarising potassium current, and I_L the 'leakage' current. The driving force for I_{Na} and I_K is generated by transmembrane Na^+ and K^+ concentration gradients, whose magnitude is the difference between V_m and the equilibrium potential for the ion, E_{ion}, computed using the Nernst equation [69]. The ionic current may then be calculated via Ohm's Law. For example I_{Na}, where g_{Na} is the Na^+ conductance (i.e. the reciprocal of R_{Na} in Fig. 2.4b), is modelled by,

$$I_{Na} = g_{Na} \cdot (V_m - E_{Na}). \tag{2.2}$$

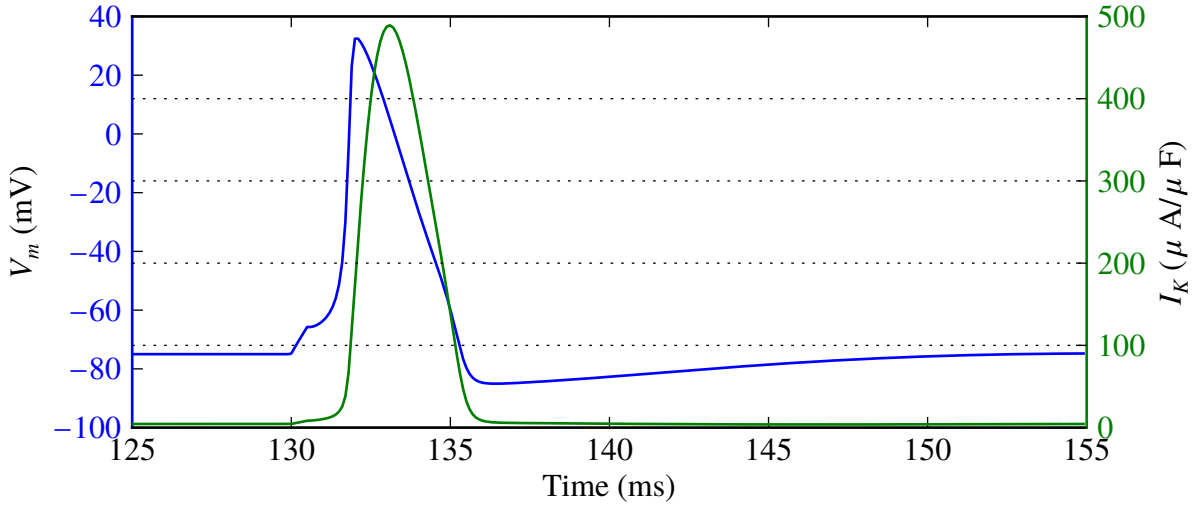
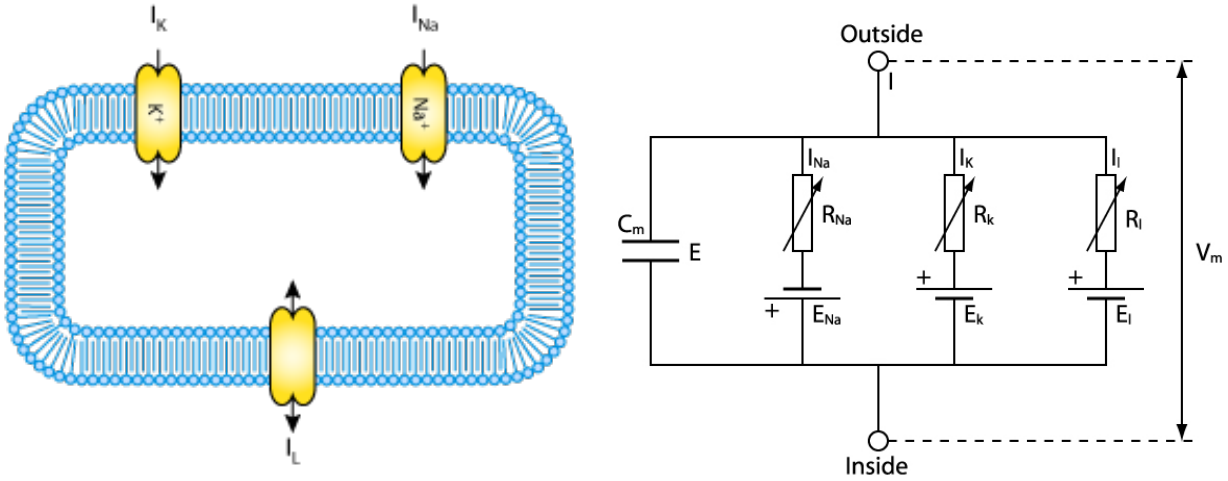


Figure 2.3: AP of the neuronal HH52 model, clearly indicating the depolarising, repolarising and gradual refractory periods. The I_K plot demonstrates the repolarising effect of K^+ on the AP.

The conductance g_{Na} is calculated as a function of the open probability of a series of hypothetical gates that represent the channel activation and the maximum conductance of the membrane for each ion type, in this case Na^+. The model describes the hypothetical activation gates of each ion channel using ODEs, these are termed HH-type channel equations and are discussed further in Section 2.1.4. This continuous, deterministic model derived by Hodgkin and Huxley to describe the ionic currents has formed the basis for almost all electrophysiological models of excitable tissues since. The HH52 model consists of the equations in Fig. 2.5 and the parameters in Fig. 2.6 to describe the transmembrane potential and ionic currents. The equations are shown grouped by their ion channel, we use these groupings in Chapter 4 to encapsulate ion channel function into independent modules. We now briefly summarise the cardiac models listed in Table 2.1 and used within our thesis, they all build upon the electrophysiological modelling concepts introduced by the HH52 model.

Noble (N62) (1962) [106]

The N62 model represents the first application of HH-type channel equations to a cardiac model, where the authors model the sodium channel within the cell. The model contains a single gated inwards current, I_{Na}, this effectively combines the Na^+ and Ca^{2+} channels.



(a) Cell based schematic of the model, indicating sodium and potassium channels, and leakage current.

(b) Circuit representation of the model illustrating channel currents, voltage-dependent conductance, and the capacitive cell membrane to model transmembrane potential.

Figure 2.4: Hodgkin-Huxley (HH52) model structure (Reproduced from CellML repository).

Beeler-Reuter (BR77) (1977) [9]

The BR model was the first to formulate the cardiac ventricular AP from transmembrane ionic currents entirely using HH-type equations. It provided the framework used in development of future, more comprehensive models of the cardiac ventricular AP, thus is useful for the development of our modular framework in Section 4.4. The model has been extensively used in simulations of AP propagation within multicellular models of cardiac tissue.

DiFrancesco-Noble (DN85) (1985) [41]

A Purkinje fibre model that incorporates not only HH-type ion channels but also ion exchangers, including the *Na–K* exchange (the sodium pump), and *Na–Ca* exchange. The *Na–K* exchange is used in subsequent models, including the Luo-Rudy models.

Luo-Rudy (LR91 and LRd94) (1991-94) [94–96]

The LR91 model of the mammalian (guinea pig) ventricular AP was developed in two major stages, the LR phase 1 model (LR91) and the dynamic LR model (LRd94). The goal of the LR91 model development was to reformulate the most important depolarising and repolarising currents based on current data from single-cell and single-channel recordings, including I_{Na}, I_K, I_{K1}. Additional development continued based on experimental findings at the single ion channel level.

Membrane voltage

$$\frac{dV}{dt} = \frac{-(-i_{Stim} + i_{Na} + i_K + i_L)}{C_m}$$

Leakage current

$$i_L = g_L(V - E_L)$$

Depolarising *Na* channel

$$i_{Na} = g_{Na} m^3 h (V - E_{Na})$$

***Na* channel m gate**

$$\frac{dm}{dt} = \alpha_m(1 - m) - \beta_m m \qquad \alpha_m = \frac{-0.1(V + 50)}{e^{\frac{-(V+50)}{10}} - 1} \qquad \beta_m = 4e^{\frac{-(V+75)}{18}}$$

***Na* channel h gate**

$$\frac{dh}{dt} = \alpha_h(1 - h) - \beta_h h \qquad \alpha_h = 0.07e^{\frac{-(V+75)}{20}} \qquad \beta_h = \frac{1}{e^{\frac{-(V+45)}{10}} + 1}$$

Repolarising *K* channel

$$i_K = g_K n^4 (V - E_K)$$

***K* channel n gate**

$$\frac{dn}{dt} = \alpha_n(1 - n) - \beta_n n \qquad \alpha_n = \frac{-0.01(V + 65)}{e^{\frac{-(V+65)}{10}} - 1} \qquad \beta_n = 0.125e^{\frac{V+75}{80}}$$

Figure 2.5: HH52 Model Equations

The LRd94 model introduced processes that control and model the dynamically changing concentrations of intracellular ions (Ca^{2+}, Na^+, K^+) and their effect on the transmembrane currents. Fig. 2.7 illustrates the model structure, depicting the membrane-based and intracellular ion channels and pumps that comprise the model. The LRd94 model was one of the first to model ion channel function using a Markov-chain based approach rather than the traditional HH-type formulation, as is discussed in Section 2.1.4. This approach captures the conformational state changes of individual channels at the cost of increased model and computational complexity. It was used to investigate the effect of Na^+ channel mutation on long-QT syndrome [26].

ten Tusscher *et al.* (TNNP04) (2004) [133]

A commonly used human ventricular model based on the LRd94 guinea pig model, containing several directly derived components. It was based on experimental human data and intended to be computationally efficient for use within multi-cellular simulation.

Iyer *et al.* (IMW04) (2004) [75]

A human ventricular model again based on the LRd94 models, with several components directly derived from it. The model provides a detailed description of whole-cell calcium homoeostasis

Parameter	Definition	Value
$V(0)$	Initial membrane voltage	-75 mV
E_R	Membrane equilibrium potential	-75 mV
C_m	Membrane capacitance	1 μF/cm^2
i_{Stim}	Stimulus current	$\begin{cases} 20, & 10 < t < 10.5 \\ 0 \end{cases}$ μA/cm^2
g_L	I_L conductance	0.3 mS/cm^2
E_L	I_L equilibrium potential	−64.387 mV
g_{Na}	I_{Na} conductance	120 mS/cm^2
E_{Na}	I_{Na} equilibrium potential	45 mV
$m(0)$	Initial I_{Na} m gate	0.05
$h(0)$	Initial I_{Na} h gate	0.6
g_K	I_K conductance	36 mS/cm^2
E_K	I_K equilibrium potential	-87 mV
$n(0)$	Initial I_K n gate	0.325

Figure 2.6: HH52 Model Parameters

and accurately reproduces diverse aspects of excitation-contraction coupling. The main ionic currents are described using Markov-chains, hence the model is very complex (with over 60 state variables) and computationally demanding. The model is commonly used and an extensive phylogenetic study has been performed using this model and the previous TNNP04 model as a base [105].

ten Tusscher & Panfilov (TP06) (2006) [134]

An improved version of the TNNP04 model from 2004, where the calcium dynamics, slow delayed rectifier current I_{Ks} and L-type calcium current I_{CaL} were reformulated. The advantage of this model is that it accurately reproduces the behaviour of the AP-duration restitution (APDR) curve. The model is currently widely used.

Grandi *et al.* (GPB10) (2010) [63]

The GPB10 model was developed from a rabbit ventricular model and includes new definitions of ionic current densities and kinetics according to recent experimental data on human myocytes. It improves the response to frequency changes, has a better performance against current block with respect to the TP06 model and simulates basic excitation-contraction coupling.

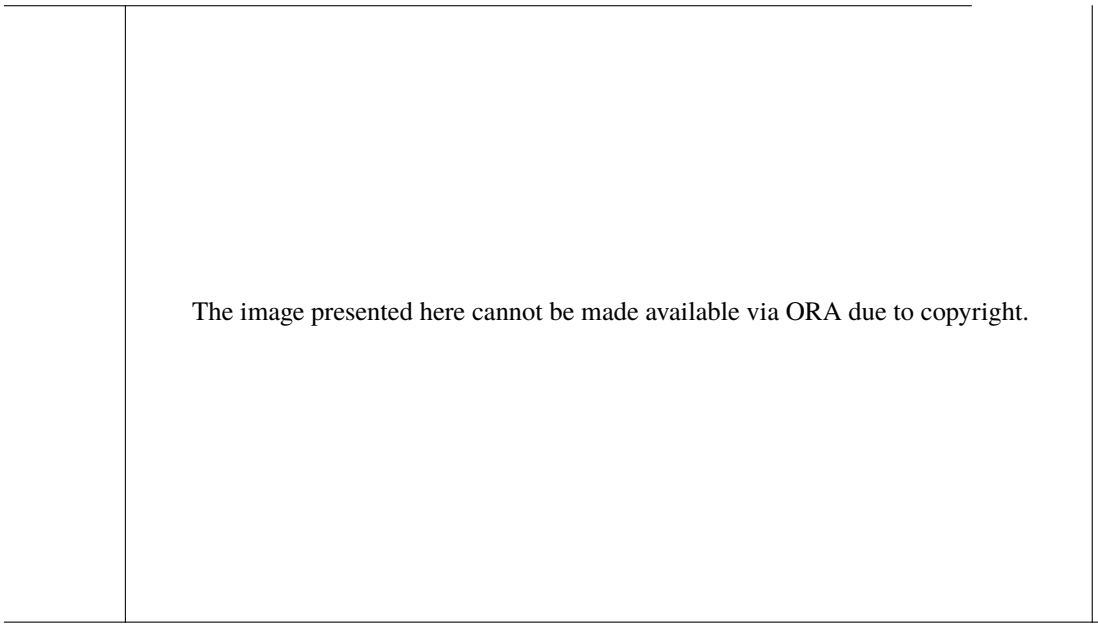


Figure 2.7: Structural overview of the Luo-Rudy ventricular cell model (LRd94) that describes the dynamic changes in intracellular Na^+, K^+, and Ca^{2+}[95]. This model uses Markov models to depict the behaviour of several ion channels and HH-type equations for the remaining channels and pumps (Reproduced from Fig. 2 in [124]).

O'Hara *et al.* (ORd11) (2011) [109]

A human ventricular AP model that uses newly obtained undiseased human ventricular data rather than those from differing cell types or species, including new measurements for the L-type Ca^{2+} current, K^+ current, and Na^+–Ca^{2+} exchange current. The ORd11 model is used to describe cellular electrophysiological mechanisms specific to human ventricular myocytes and successfully reproduces experimental AP morphology, AP duration (APD) rate dependence, and restitution. Interestingly the model deliberately utilises HH-type equations for currents to remain computationally efficient.

2.1.4 Ion Channel Modelling

Cardiac cellular models have traditionally modelled the transmembrane ionic currents that cause an AP through HH-type gating equations that represent a large ensemble of ion channels at a macroscopic level [69]. However such a scheme does not consider the relationship between ion channel structure and function at the single-channel level. More recently ion channel states have been modelled using a Markov-based approach that captures the conformational state changes of individual channels at the cost of increased model and computational complexity [75, 134].

2.1.4.1 HH-type gating

HH-type gating was demonstrated in Fig. 2.5 to calculate the HH52 K channel current. Referencing this with the general HH-type current equation in Eq. (2.2), we see that the maximum channel conductance, g_K, is multiplied, and effectively 'gated', by the n^4 parameter. Within the HH52 model, this can be seen to represent 4 identical, independent, membrane-bound 'particles' that control the opening of the K channel, where the probability of each particle being in the correct, permissive, position is n. The kinetic behaviour of the particles transitioning between permissive and non-permissive positions can be described by a first-order reaction,

$$n' \underset{\beta_n}{\overset{\alpha_n}{\rightleftharpoons}} n, \tag{2.3}$$

where n' is the probability of not being in the permissive position n, and α_n and β_n are voltage-dependent rate constants previously given in Fig. 2.6 [67]. Deriving the reaction-rate equations for this reaction provides the ODEs used within the HH52 model as seen in Fig. 2.5.

2.1.4.2 Markov-formulation Gating

The HH-type gating scheme considers the activation gates to be fully independent and kinetically identical, modelling the ionic current transmitted through large ensembles of ion channels at a macroscopic level. However, simulating ion channel mutations and molecular interactions requires the formulation of single-channel models that represent specific channel states (e.g. open, closed, or inactivated), their interdependencies, and their incorporation into whole cell models [124]. This single-channel approach constitutes a major departure from the HH-type gating scheme adopted in many cardiac cell models.

Models that express the individual ion channel states are termed Markov-formulation models, as transitions between states depend only on the present channel conformation [67, 103]. Markov-formulation models compute the occupancy of a channel in its various kinetic states as a function of voltage and time (and possibly other factors such as ligand binding), conducting ions when it occupies an open state(s) [128].

Returning to the HH52 K channel from the previous section, we may model a single K channel as containing 4 independent and identical subunits that each have a resting and activated state, with the same transition rates α_n and β_n as before. It is only when all subunits are in the activated

state that the channel is open, giving rise to the follow kinetic state diagram [67],

$$C1 \underset{\beta_n}{\overset{4\alpha_n}{\rightleftharpoons}} C2 \underset{2\beta_n}{\overset{3\alpha_n}{\rightleftharpoons}} C3 \underset{3\beta_n}{\overset{2\alpha_n}{\rightleftharpoons}} C4 \underset{4\beta_n}{\overset{\alpha_n}{\rightleftharpoons}} O, \tag{2.4}$$

where $C1$-$C4$ represent closed states, and O an open state when all channel subunits are active[1]. As shown in [124], we may then modify the HH-type current equation in Eq. (2.2) to provide the following equation that describes the macroscopic current density through an ensemble of such channels residing in the open state as defined by O and changing over time,

$$I_X = \overline{g_{sc,x}} \cdot n \cdot O \cdot (V_m - E_X), \tag{2.5}$$

where for an arbitrary channel X, $\overline{g_{sc,x}}$ is the single channel conductance, n is the number of channels (per unit membrane area), O is the open state probability and $(V_m - E_X)$ is the driving force. O may obtained by modelling the first-order reactions in the system shown in Eq. (2.4) according to kinetic theory [67]. This may be through the use of reaction-rate ODEs, or via continuous and discrete stochastic methods that may be more appropriate in certain scenarios, as discussed in the next section. Using ODEs we obtain the following set of equations, the method by which this system is derived is discussed in Section 2.2.5,

$$\begin{aligned}
\frac{dC_1}{dt} &= -4\alpha_n C_1 + \beta_n C_2 \\
\frac{dC_2}{dt} &= -(-3\alpha_n + \beta_n)C_2 + 4\alpha_n C_1 + 2\beta_n C_3 \\
\frac{dC_3}{dt} &= -(-2\alpha_n + 2\beta_n)C_3 + 3\alpha_n C_2 + 3\beta_n C_4 \\
\frac{dC_4}{dt} &= -(-\alpha_n + 3\beta_n)C_4 + 2\alpha_n C_3 + 4\beta_n O \\
\frac{dO}{dt} &= -\beta_n O + \alpha_n C_4.
\end{aligned} \tag{2.6}$$

This single-channel formulation of the current density can be incorporated into a cellular AP model, providing a mechanistic link between the whole-cell AP and the ion channel structure/function. The Luo-Rudy ventricular model (LRd94) [95, 96] depicted in Fig. 2.7 provides an example, using Markov-models to describe several ion channels.

[1] In fact, this state diagram is kinetically identical to its HH-type equivalent, however many ion channel state diagrams include dependent transitions that can not be modelled from a HH-type macroscopic viewpoint [124].

2.1.5 Summary

This section introduced the basics of cardiac cell electrophysiology from a biological and modelling perspective. We outlined a selection of related cardiac electrophysiological cell models that we reference throughout this thesis, describing ion channel modelling with an in-depth look at the Hodgkin-Huxley squid giant-axon model (HH52). HH-type ion channels models and their limitations were discussed and Markov-style models that explicitly model individual channel state were introduced. In the following section we discuss chemical kinetic theory and its application to modelling and simulating Markov-formulation ion channels.

2.2 Biochemical Reaction Kinetics

Biological systems can be described as systems of biochemical reactions, encompassing processes such as metabolism, signal transduction, and gene expression; or in our case, modelling the kinetic behaviour of Markov-formulation ion channels [67]. The reactions may involve bond-breaking, as in the classic definition of a chemical reaction, or may only involve strong non-covalent interactions between molecules, such as hydrogen bonding [103]. Either way, one may measure the thermodynamics and kinetics of these reactions and quantitatively describe their rates.

When modelling biological systems we tend to consider such high reaction propensities that the observed behaviour appears continuous, and at such a scale that the underlying stochastic properties become negligible. This aids model simplicity and computational efficiency. However such an approach abstracts over the physical laws of chemical kinetics. The time evolution of a chemically reacting system is neither continuous nor deterministic and is instead reliant upon discrete, molecular Brownian motion. Given the same initial condition, different trajectories of the state will occur and trajectories may cross paths in time. For instance, inherent stochastic fluctuations in particle numbers can change considerably the dynamic behaviour of biochemical systems both quantitatively and qualitatively [140].

Biochemical reactions may be modelled by a variety of processes deriving from a definition known as the Chemical Master Equation (CME). The CME is extremely hard to solve analytically or numerically. We present several simulation algorithms and approximations, each increasing the generality of simulating a trajectory of the CME in exchange for increased algorithmic simplicity and computational efficiency [111]. These approximations are used to model systems in a computationally efficient way at a large scale and comprise many biological models in use today, including those of Markov-formulation ion channel models. Finally, we discuss hybrid systems that combine multiple approximations in parallel during simulation, increasing computational efficiency and enabling a modeller to determine the accuracy required when simulating particular subsystems of interest.

2.2.1 Stochastic Biological and Cardiac Modelling

The stochastic effects of biological systems at micro-scales can have significant effects upon their observed behaviour and should be considered when developing computational mathematical models. Stochastic behaviour affects higher level biochemical and biological processes, such as biochemical reaction networks, gene transcription and ion channel fluctuations and should be accounted for within exact models. Cells, for instance, are well known to exhibit properties such as bistability and also may have different properties despite having the same parent cell — investigation of stochastic dynamics can explain these phenomena. The stochastic properties of a system may be under investigation also, to determine the behaviour of systems in response to molecular noise and robustness to random perturbations.

Within the cardiac modelling domain, both HH-type and Markov-formulation ion channel models are usually simulated continuously and deterministically through ODE-based approximations of the reaction kinetics. This approximation is appropriate in the general case where channel populations are so large that stochastic behaviour is negligible, and aids computational efficiency [64]. However it is possible to simulate the state occupancy of a Markov-formulation ion channel in a stochastic manner to capture more accurately the underlying biological processes of molecular activation and channel conformational changes. For instance when modelling ion channel mutations [26] or the effect of drugs blocking specific channels, channel populations may decrease such that stochastic behaviour becomes significant. Other cases may include modelling stochastic events such as ectopic beats that generate arrhythmias, or simply investigating ion channel behaviour through more detailed simulations. These cases may be simulated discretely through the stochastic simulation algorithm (SSA) [59] or continuously through stochastic differential equations (SDEs) [61] — as are introduced in this section.

2.2.2 Chemical Master Equation (CME)

Consider a unimolecular, reversible, reaction between chemical species with one intermediate state, where a_1 and a_2 are the forward reaction rate constants, and b_1 and b_2 the reverse rate constants,

$$X1 \underset{b_1}{\overset{a_1}{\rightleftharpoons}} X2 \underset{b_2}{\overset{a_2}{\rightleftharpoons}} X3. \tag{2.7}$$

The evolution of this system over time in a well-mixed solution is dependent on several factors. This includes the reactions between the species, the number of molecules of each species, temperature, and pressure. Let $\mathbf{N}$ be the number of chemical species $\{S_1, \ldots, S_N\}$ in such a system, which interact through $\mathbf{M}$ chemical reactions $\{R_1, \ldots, R_M\}$, and $X_i(t)$ denote the number of molecules of a species S_i at time t. The evolution of the system will allow for determining the state vector $\mathbf{X}(t) \equiv (X_1(t), \ldots, X_N(t))$ at time t, given it was in state $\mathbf{X}(t_0) = \mathbf{x}_0$ at some initial time t_0.

We are interested in the species population changes caused by chemical reactions, triggered by molecular collisions within the solution due to Brownian motion, a stochastic and discrete process. Each reaction R_j has two key properties. The first is a state-change vector $\mathbf{v}_j \equiv (v_{1j}, \ldots, v_{Nj})$, where v_{ij} is the change in population S_i caused by one R_j reaction, causing a state change from $\mathbf{x}$ to $\mathbf{x} + \mathbf{v}_j$. The other is the reaction propensity function a_j, defined as,

$a_j(\mathbf{x})\, dt \triangleq$ the probability, given $\mathbf{X}(t) = \mathbf{x}$, that one R_j reaction will occur somewhere within the solution in the next infinitesimal time interval $[t, t + dt)$.

The propensity function for a particular reaction R_j can be determined by the reaction type (unimolecular or bimolecular), the number of molecules of the species concerned (e.g. x_i, x_k) and a reaction constant c_j.

The time-evolution equation for the system state probability distribution, $P(\mathbf{x}, t \mid \mathbf{x}_0, t_0)$, may be derived and is known as the Chemical Master Equation (CME) [111]. The CME is a set of coupled ODEs, with one equation for every possible combination of reactant molecules. This may be solved analytically for simple cases, however for larger/more complex systems even numerical solutions become impractical due to the resultant explosion in system state space [60]. An equivalent solution may be formed from the trajectories of the system state over time; by computationally simulating an ensemble of trajectories, one may generate the distribution.

Stochastic simulation is therefore a way to generate trajectories of a Markov process to compute the distribution of all possible trajectories, effectively sidestepping the problems of solving the CME. However, assembling an accurate distribution requires a large number of trajectories, usually upwards of 5000 [111]. Therefore a number of different stochastic algorithms, approximations and hybrid schemes exist to simulate and optimise the process [66, 91].

2.2.3 Stochastic Simulation Algorithm (SSA)

The Stochastic Simulation Algorithm (SSA) is an exact discrete algorithm that generates a simulation trajectory of a chemical reaction system [59, 60], representing a single possible evolution of the system as described by the CME. The key to generating simulated trajectories of $\mathbf{X}(t)$ is a new probability function, termed the reaction probability density function,

$P(\tau, j \mid \mathbf{x}, t) \triangleq$ the probability, given $\mathbf{X}(t) = \mathbf{x}$, that the next reaction in the system will occur in the infinitesimal time interval $[t + \tau, t + \tau + dt)$ and will be an R_j reaction.

This is a joint probability density function of two random variables, the time to the next reaction τ and the index of the next reaction j, given the system is in state $\mathbf{x}$. Several exact Monte Carlo procedures exist for generating samples of τ and j according to this distribution [60, 111]. The original simulation algorithm is termed the direct method and is perhaps the simplest. Under this method, we set,

$$\tau = \frac{1}{a_0(\mathbf{x})} \ln\left(\frac{1}{r_1}\right),$$
$$j = \text{the smallest integer satisfying } \sum_{j'=1}^{j} a_{j'}(\mathbf{x}) > r_2 a_0(\mathbf{x}),$$

where r_1 and r_2 are random numbers drawn from the uniform distribution in the unit interval, and $a_0(\mathbf{x})$ is the sum of all reaction propensities when the system is in state $\mathbf{x}$. We then trigger the chosen reaction R_j, this involves updating the system state from $\mathbf{x}$ to $(\mathbf{x} + \mathbf{v}_j)$, and incrementing the time from t to $(t + \tau)$. As such, by iteratively drawing random numbers according to the density function and updating the system state in response, simulation may progress, one reaction after the other. The direct-SSA is described further in Appendix B.3.

During simulation the algorithm effectively simulates discretely the path of each molecule within the mixture using random numbers that determine which reaction to trigger and when. As a result, the direct-SSA is a computationally expensive operation whose most complex step, determining the next reaction, increases linearly with the number of reactions, i.e. $O(\|\mathbf{M}\|)$. As each simulation is expensive and many simulation runs are required, the direct-SSA is rarely used for larger systems or within multi-scale models.

Improvements to the exact, direct-SSA, such as the optimised direct method and next-reaction

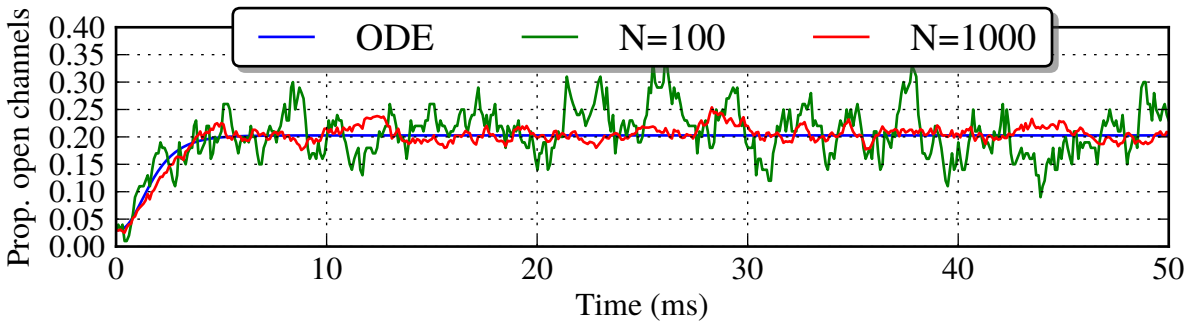


Figure 2.8: Simulation of HH K channel at V = 0 using the SSA with the number of channels, N, at 100 and 1000. As N increases behaviour tends towards the deterministic representation.

method [60], optimise computing the next system state and determining the next reaction. Approximate SSA algorithms, such as the τ-leap method [58], attempt to balance computational efficiency with increasing error depending on, for instance, the speed of the reaction (stiffness) or the number of molecules of each species in the reaction.

2.2.3.1 SSA ion channel simulation

We may consider the reaction system in Eq. (2.7) as representing a simple Markov-formulation ion channel, where the 3 chemical species represent the individual channel states; the reaction rates represent the transition rates; and the number of molecules of each species represents the discrete number of channels in that particular state [67]. The SSA can be used to simulate this ion channel system from a discrete, stochastic perspective. Such simulation takes into account the stochastic behaviour that occurs at low channel numbers, potentially resulting in biologically-relevant differentiating behaviour, as discussed in Section 2.2.1.

Fig. 2.8 demonstrates this using the HH52 K channel derived in Eq. (2.4). This figure plots the proportion of open K channels given a clamped membrane voltage (V_m) of 0 mV, when using a deterministic ODE representation (see Section 2.2.5) and two SSA simulation trajectories where the total number of channels, N, is set at 100 and 1000. All plots indicate that at this voltage ~20% of the channels are open and generating an ionic current. However, the SSA plots demonstrate stochastic behaviour in the number of open channels, as discussed in Section 2.2.1, that becomes less relevant as N increases.

2.2.4 Chemical Langevin Equation (CLE)

As the number of molecules in a species grows larger, the discrete effects become less apparent and the reactions in the system may be modelled continuously. At this point we can model the jump-Markov process of the CME as a continuous-Markov process whose distribution is governed by Chemical Langevin Equation (CLE) [61, 111]. The CLE is a multi-variable Itō stochastic differential equation (SDE) [83] that approximates the time evolution of the molecular counts of reacting chemical species, and is as follows,

$$d\mathbf{X}(t) = \sum_{j=1}^{M} \mathbf{v}_j a_j(\mathbf{X}(t))\, dt + \sum_{j=1}^{M} \mathbf{v}_j \sqrt{a_j(\mathbf{X}(t))}\, dW_j(t), \tag{2.8}$$

where $W_j(t)$ are normally-distributed Wiener increments representing Brownian motion. We can consider the first half the equation as the deterministic evolution of the system due to the reaction rates, described further in Section 2.2.5, and the second half as the random fluctuations caused by Brownian motion. As with the CME, it is impractical to analytically solve the CLE, numerical simulation is instead used to obtain trajectories of the process that may be used to generate the probability distribution of the system states. Simulation may be performed using numerical solvers such as the Euler-Maruyama (EM) [83] and variants that utilise reflection or projection to keep species levels positive and bounded [37, 110]. Alternate formulations of the CLE exist that optimise simulation by reusing Wiener increments, reducing expensive random number generator (RNG) calls during simulation [99]. SDEs and the EM solver are discussed further in Appendix B.2.

2.2.4.1 SDE-based ion channel simulation

We may use the CLE in Eq. (2.8) to model the reaction system in Eq. (2.7) as if it were a 3-state ion channel, as in Section 2.2.3.1. We present the following discretised form for modelling ion channels based on Eq. (2.8) [37, 99, 110],

$$\Delta\mathbf{X} = \mathbf{A}\mathbf{X}\Delta t + \frac{1}{\sqrt{N}}\mathbf{E}\mathbf{F}(\mathbf{X})\Delta\mathbf{W}, \tag{2.9}$$

where Δt is the scalar time increment; $\mathbf{X}$ a vector of current state values/proportions; N is the total population (i.e. number of channels in the system), $\Delta\mathbf{W}$ is the vector of Wiener increments

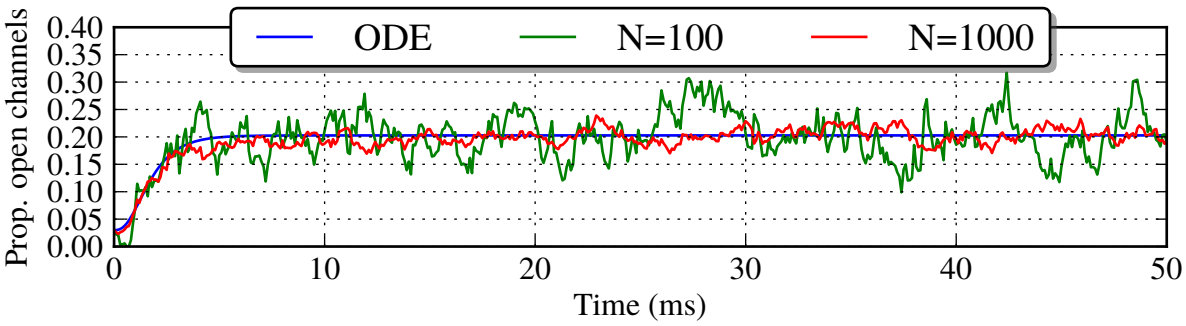


Figure 2.9: Simulation of HH K channel at V = 0 using SDEs with the number of channels, N, at 100 and 1000. As N increases behaviour tends towards the deterministic representation.

for each reversible reaction; $\mathbf{E}$ is a binary matrix of states against reversible reactions; and $\mathbf{F}(\mathbf{X})$ is a diagonal matrix whose elements are the root of the sum of the propensities for each reaction. Finally, $\mathbf{A}$ represents the deterministic reaction rates derived from the sum of reaction propensities and is given in Section 2.2.5.1 on Page 34. Specific values for the reaction system described in Eq. (2.7) using the CLE form shown in Eq. (2.9) are as follows,

$$\mathbf{X} = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} \qquad \Delta\mathbf{W} = \begin{bmatrix} \Delta W_1 \\ \Delta W_2 \end{bmatrix} \tag{2.10}$$

$$\mathbf{E} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \qquad \mathbf{F}(\mathbf{X}) = \begin{bmatrix} \sqrt{a_1 X_1 + b_1 X_2} & 0 \\ 0 & \sqrt{a_2 X_2 + b_2 X_3} \end{bmatrix}.$$

The Euler-Maruyama (EM) scheme can be used to simulate a trajectory of the evolution of this ion channel system. This is demonstrated in Fig. 2.9, presenting a trajectories of the HH52 K channel derived in Eq. (2.4). This figure plots the proportion of open K channels given a clamped membrane voltage (V_m) of 0 mV, when using a deterministic ODE representation, and two EM simulation runs of the SDE channel representation where N is set at 100 and 1000. The results are similar to discrete trajectories of Fig. 2.8, with ~20% of the channels in a open state. As with the SSA results, the SDE-based trajectories demonstrate stochastic behaviour in the number of open channels that becomes less relevant as N increases.

2.2.5 Reaction Rate Equations (RREs)

For even larger numbers of molecules of a species, the noise term of the CLE (see Eq. (2.8)) becomes negligibly small and, as was alluded to earlier, we obtain,

$$d\mathbf{X}(t) = \sum_{j=1}^{M} \mathbf{v}_j a_j(\mathbf{X}(t))\, dt, \tag{2.11}$$

In fact, this ODE is a manifestation of the reaction rate equations (RRE) from conventional deterministic chemical kinetics used to represent the system, where the propensity constant c_j can be converted into the conventional rate equation constant k_j [61]. The RREs may also be derived directly from the initial reaction equations through the rate laws, for instance many reactions defined in biochemical kinetics consist of second order reactions at most, and the conversion to RREs is known.

2.2.5.1 ODE-based ion channel simulation

We may use the ODE in Eq. (2.11) to model the reaction system in Eq. (2.7) as if it were a 3-state ion channel, as in Sections 2.2.3.1 and 2.2.4.1. As with the CLE, we present a discretised form,

$$\Delta\mathbf{X} = \mathbf{A}\mathbf{X}\Delta t, \tag{2.12}$$

where, as before, Δt is the scalar time increment; $\mathbf{X}$ a vector of current state values/proportions; and $\mathbf{A}$ represents the deterministic reaction rates derived from the sum of reaction propensities for each state in the system. Specific values for the reaction system described in Eq. (2.7) using the ODE form shown in Eq. (2.12) are as follows,

$$\mathbf{X} = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} \qquad A = \begin{bmatrix} -a_1 & b_1 & 0 \\ a_1 & -(b_1 + a_2) & b_2 \\ 0 & a_2 & -b_2 \end{bmatrix}. \tag{2.13}$$

We can solve this equation numerically to determine the continuous behaviour of the system, using ODE solvers such as the forward-Euler, or 4^{th}-Order Runge-Kutta, these are discussed further in Appendix B.1. Figs. 2.8 and 2.9 included the deterministic solution obtained from simulating the HH52 K channel derived in Eq. (2.4) as a reference. This representation was used

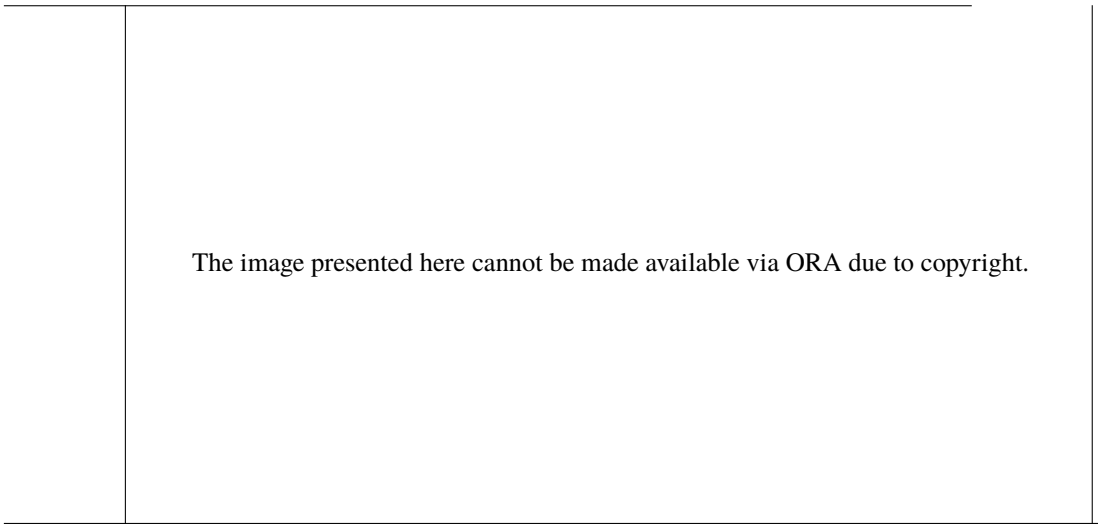


Figure 2.10: Hybrid simulation process that partitions a reaction system into subsections for simulation using differing approaches. Partitioning is dependent on several properties, including the reaction propensity and species population. (Reproduced from Fig. 2 in [111])

to deterministically model the repolarising current, I_K, in Fig. 2.3.

2.2.6 Hybrid Modelling and Simulation

It may be useful to combine the methods available to simulate biochemical processes within a single simulation. This may be to model more closely the observed behaviour or to increase the computational efficiency in exchange for numerical approximations during the simulation process. We may apply hybrid modelling to the cardiac electrophysiological domain, where a Markov-formulation ion channel may be modelled and simulated using one of the reaction schemes previously described, whilst the membrane voltage, V_m is calculated by an ODE utilising the ionic currents generated by each channel (see Eq. (2.1)); enabling an holistic view of the impact of stochastic ion channel behaviour on the AP.

Hybrid simulation aims to combine the advantages of these alternative, complementary simulation methods [111]. Fig. 2.10 illustrates this, the entire system is partitioned into subsections that may be simulated using the appropriate methods simultaneously. For instance continuous and discrete schemes may be mixed to compensate for computational inefficiencies, or stochastic and deterministic schemes used when investigating the stochastic behaviour of a model subsystem.

The conversion of a model into separate partitions may be manual and specified by the modeller or automated depending on various criteria. Similarly partitioning may be performed statically ahead of simulation or dynamically with re-partitioning occurring during simulation.

Partitioning strategies may take into account reaction rates, species population levels or rate of fluctuations, and other factors depending on the properties under investigation for the model.

Simulation proceeds both independently and simultaneously for partitions, with synchronisation occurring as required. For instance in one hybrid configuration, between two discrete events the fast/continuous partitions may evolve solely due to their own actions and be simulated by ODEs, SDEs or approximate discrete simulations independently.

However this approach ignores any effects that the independent evolution of each partition within the system may have on other partitions. For instance, reaction propensities a_j within a slow partition may be altered by the independent evolution of fast partitions and ideally should be a function of time, that is $a_j(t)$. Hybrid approaches allow for significant computational efficiency gains, however usage may result in numerical error caused by approximations during simulation.

2.2.7 Summary

We have provided an overview of the mathematical modelling of biochemical reactions using the CME. We illustrated how a series of approximations may be used to simulate chemical kinetic systems described by the CME, ranging from exact, stochastic, yet computationally demanding, algorithms to the continuous, deterministic reaction rate equations. Hybrid simulation of systems was discussed, allowing for models to be divided into partitions that are simulated independently. A hybrid-approach permits modellers to capture the stochastic behaviour of subsystems with varying degrees of accuracy whilst enabling efficient computational simulation.

The methods described for modelling and simulating biochemical reactions may be applied to Markov-formulation ion channels within cardiac cell models. This was demonstrated through several sample plots and allows for a closer link between the channel states, potential stochastic behaviours and their effect on the action potential (AP). However such models may be difficult to construct, and computationally expensive to simulate, prohibiting usage within multi-scale models. In the following sections we detail how DSLs may aid the creation of ion channel models that are backed by optimised implementations to enable efficient chemical kinetic (hybrid) simulation. Providing support for stochastic modelling and high-performance simulation within our DSL will facilitate the creation of more complex and detailed models that may provide insight into specific areas of ion channel function.

2.3 Software Engineering

Software engineering is an engineering discipline concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use [68, 129]. Programs, i.e. models in our case, must be correct and meet requirements, thus the discipline depends on scientific approaches to programming that can be observed through the use of program analysis, type systems, code generation, and more to develop complex software. It involves the application of theories, methods, and tools where appropriate, yet as an engineering discipline may involve discovering solutions when the theories are not applicable. This is a practical aspect derived from experimentation, resulting in best practices and domain knowledge. The combination of these approaches is termed *domain modelling*, a set of processes specific and ideal to the domain expressed through tools such as domain specific languages (DSLs) and development/modelling environments [68].

The discipline is larger than just the technical aspects of software development, encompassing activities such as software project management, deployment, and documentation, to form a set of processes that support software production. Several of the fundamental activities common to all software development processes include [129],

Specification the definition of the software to be produced,

Development the process of designing and programming the software,

Validation checking the software to ensure it is correct and meets the specification,

Evolution modifications to reflect changing specification, new features and general refactoring.

Well-designed software has several attributes: it is maintainable, i.e. may evolve to meet changing needs; dependable, covering aspects such as reliability, correctness, security, and safety; efficient with regards to hardware resources; and acceptable to its intended users for its designed purpose. This can be achieved through the application of both scientific and practical engineering approaches to the software development lifecycle [68].

In the following sections we discuss several aspects of software engineering that we believe will aid the development of large-scale, reusable, reliable, and extensible models. We describe the benefits domain-specific languages (DSLs) bring to domain experts unfamiliar with software

engineering, particularly when performing complex computational tasks such as biological modelling and simulation. We further describe the use of type systems in developing reliable software, and the advantages of modularity, abstraction and component-based architectures when developing reusable software. Finally we discuss testing methodologies in relation to experimental data as a means to ensure model correctness.

2.3.1 Domain-Specific Languages

In contrast to general-purpose languages utilised for any computational task, DSLs are designed to perform a specific task in an intuitive manner [49]. A well designed DSL is capable of simply expressing a domain task, with fewer lines of code. Specific domain knowledge enables syntactic and semantic checks and a wide range of optimisations to be applied at the domain and implementation levels. DSLs may be external or internal [70]; external DSLs are stand-alone languages with their own implementation, whereas internal DSLs are embedded within a host language and rely on the host implementation to function [36, 127].

DSLs enable modellers to describe their model and associated domain-specific data in a significantly simpler form than would be required in a general-purpose language. An example biological DSL may allow input of both parameters and valid parameter ranges for a model instance. These may be used to perform automated sweeps over the parameter ranges during simulation to determine the sensitivity of the model to variation. Useful domain-specific data for biological models may include values for *clamping* parameters and units-of-measure annotations.

Additionally, from a domain perspective the creation of models in existing programming languages can be difficult for modellers lacking a programming background and is far removed from the exploratory modelling process. This can result in the presence of low-level computational details and language specific idiosyncrasies within the model definition and simulation code. Such code hinders model development, model reuse and *in silico* experimentation into model behaviour. As a result many DSLs have been developed to ease the creation of models by modellers.

2.3.1.1 Existing Biological Modelling DSLs

A range of computational modelling methodologies and DSLs have been researched and developed, covering many biological domains [46, 122]. Such computational models may focus on the design of executable computer algorithms that mimic biological phenomena. For instance, the

(stochastic) pi-calculus has been applied to the biological domain [118, 121], and used to model and check several low-level biochemical processes including signal receptors, signal transduction pathways and the Na^{2+} ion pump [24, 25]. However further research is needed regarding the computational efficiency of such discrete, low-level models and their integration within higher-level or multi-scale models. As discussed in Section 2.1.2, cardiac electrophysiological modelling employs predominantly ODE-based methods, extending the mathematical modelling foundation derived by Hodgkin and Huxley and applied to cardiac cells by Noble (see Section 2.1.2). Our research focusses upon such models, in particular the mathematical modelling of ion channel function within the cell using multiple representations, from continuous and deterministic, to discrete and stochastic.

Within the domain of physiological modelling, different approaches exist for describing models beyond documenting the raw mathematics or formal specification. A wide range of computational tools for developing models and running physiological simulations have been developed; with many modellers utilising more mathematically oriented languages, such as Mathematica, MATLAB and R, or general-purpose programming languages, such as C/C++, Java, and Perl, to perform their modelling tasks. Machine-readable description languages are needed to develop computational simulations, however the use of general-purpose development tools and languages for this task creates challenges.

Models tend to be created as *ad hoc* operational code rather then utilising standardised specifications, resulting in an overlap between model definition and simulation code. This may increase the difficulty in reproducing results and hinders model reuse, collaboration, and composition; since different model implementations may be based on completely different technological frameworks. Without a *common reference model* [42] (or *lingua franca*) 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, through the Systems Biology Markup Language (SBML) [73] and CellML [93] research programmes. SBML is a domain-specific XML markup language that addresses biochemical processes and reactions at the molecular scale. It is primarily used as a common interchange format for such models between multiple development and simulation

environments. In contrast to CellML it does not support model composition or the higher-level conceptual modelling of spatially homogeneous cells.

2.3.1.2 CellML

Initially developed within the cardiac modelling community, CellML is a modelling markup language that aims to cover a range of biological phenomena with a focus on cellular function. A CellML model is composed from several individual components that may each contain ODEs. This enables the description of continuous deterministic cardiac models, however it does not enable explicit ion channel modelling or stochastic simulation. Components may be connected via containment or encapsulation, mimicking biological systems' structure, to create larger or more complex models [141]. An extensive online repository of CellML models has been created[2], these models can be simulated directly or manually adapted for further research.

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 unaccounted 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 [141]. The grouping of concerns to achieve better modularity and encapsulation is left to the developer.

The fundamental issue with CellML is, as with SBML, it was developed primarily to enable interoperation between simulation environments for existing, published and validated models. The format may be used with several modelling and simulation environments, such as OpenCell and Chaste, to simulate (cardiac cell) models. It was not designed to supplant the use of general-purpose languages such as MATLAB for developing models or to enable model experimentation — it is purely a curation language for describing existing models that have been implemented and validated elsewhere. As a result CellML has a rich set of features to aid the classification of published models, for instance, author and institution tags, model description sections, and extensible RDF-based metadata facilities. Conversely CellML lacks features we believe may aid the model development process, particularly those that aid the use of software engineering.

CellML is based on XML and as a result is verbose and hard to edit directly. This may

[2] `http://models.cellml.org`

be seen in Appendix D.3, containing (edited) CellML representations of the HH52 and BR77 electrophysiological models (see Section 2.1.3). It necessitates the use of custom curation environments, such as COR [53] for CellML, rather than direct development and presents an unfamiliar and somewhat steep learning curve for using the DSL. A shorthand/non-XML syntax for CellML has been developed that greatly reduces the model size, an example is provided in Appendix D.3. However this syntactic change does not address the following semantic issues that make CellML impractical for model development and experimentation.

Firstly, CellML does not support the operational development of models, providing no means for abstracting computations. Components with input and output parameters are provided to collate logical related expressions, however these parameters must be manually linked to a single fixed value within the main model. The component can not be reused with other values to provide an parameterised abstraction over the expressions similar to how a function does. These components are manually connected to each other within a separate `connection` section, in contrast to the value assignment and function calling conventions common to general-purpose languages. Furthermore, although CellML supports structural notions of encapsulation and containment, these are not expressed when actually defining the components. Components instead exist in a flat namespace and model structure is explicitly defined within a separate `group` section. This inhibits model cohesion and again is in contrast to general-purpose languages, where such structural notions are provided in a cohesive manner, e.g. scoping rules and accessibility modifiers.

Modularity was introduced with CellML version 1.1, in that encapsulated models can be linked together through public and private interfaces. This allows multiple models whose variables might refer to the same entity to be logically linked. This component-based approach allows reuse of whole models or parts of models described with CellML markup. This modular coupling has been demonstrated in a number of published models, however it is not without its problems. The model developer needs to ensure consistency of variable names, as well as manually normalise units used across modules. Variables can also be made directly accessible by declaring them so through public interfaces, again leading to content coupling. As such, importing can require great discipline on the modellers part and the majority of models and tools are CellML 1.0 only; best practices for modular modelling in CellML may be found in [141]. Furthermore, modules

are statically imported and configured during development; development against interfaces that may be later bound to alternate implementations is not supported, reducing the (collaborative) modelling benefits that modularity provides.

These issues arise due to the overriding notion of CellML first and foremost as a curation language for validated models. In contrast we wish to research and use DSLs to aid the model development process, translating software engineering techniques used to construct large-scale systems to the domain of computational mathematical model development. This includes supporting the abstraction and parameterisation of expressions, collaborative modular model development, and model verification techniques such as type and units-of-measure checking.

2.3.2 Type and Units Systems

Types

We introduce types by referencing the following definition from [119],

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

Types can be used to compute the validity of expressions in a program, with the type of an expression partially specifying what it does. The types of expressions may be checked dynamically at run-time, such as in Python or MATLAB, or statically at compile-time, as in Java or C#.

Whilst general static type systems are weak compared to dependently-typed systems and/or automated proof environments, they have several benefits: they are comparatively lightweight such that programmers use them; they are machine-checked, often with minimal programmer assistance; and they are both mandatory and ubiquitous such that programmers cannot avoid them [81]. As a result we can consider the static type-checker to be the most widely used verification technology in use today, providing a good cost/benefit ratio. Additionally, from a software engineering perspective: static types provide feedback regarding the well-formedness of a program during development; they provide information and enable optimisations that improve code-generation and runtime performance; and they do not require expensive run-time checks.

Types may be manually created and specified in a program, or inferred automatically via typing rules that ensure the correct construction of a fully-typed program. Type inference is

commonly used by functional programming languages, such as OCaml [90] and Haskell [116], providing the benefits of static typing without requiring explicit annotations and thus easing developer usage.

A strong, static type system supports program verification by restricting the set of types that are allowed or admitted for a certain value. This may also be used to verify computational models, ensuring that 'well-typed' models will be compiled successfully and simulation will complete. Such verification provides a barrier through which only well-formed and correctly constructed models may pass. Tangentially related to types are the concept of ontologies, used to semantically identify and describe the behaviour of model elements [33, 52, 54, 93], however they are not considered within our research due to time constraints.

Units

We can think of units-of-measure within mathematical equations as analogous to a type-system, depicting the kind of value an expression may represent. They can be manually applied and checked within calculations involving physical systems to ensure dimensional consistency in the same way as a type-system checks computational expressions. Usage of correct units is vitally important when modelling and simulating physical systems, computationally or otherwise. The incorrect computation of units has led to serious failures in systems software — for instance the Mars Climate Orbiter was lost in 1999 due to software mixing values in pound-seconds and newton-seconds[3].

Units-of-measure annotations and checking has been implemented in several general-purpose languages, including Ada [2], and F# [78, 79, 131]. Within the biological modelling domain, CellML supports unit annotations for values that can be checked by several CellML systems, including COR and PyCML [30]. However the unit system is basic and cumbersome to use. It requires the upfront definition of every unit used in the model and the manual annotation of a unit to every variable within the model. Furthermore the units assigned to a value are static and cannot be converted or cast between one another [93].

We believe units systems could play a greater role within the biological modelling domain. Firstly, unit inference, in conjunction with unit-aware mathematical operations, would reduce the number of manual unit annotations required in a model. Secondly, unit casting and conversion

[3] `http://mars.jpl.nasa.gov/msp98/orbiter/`

would increase interoperability between independently developed models. For instance, several cardiac models discussed in Section 2.1.3 describe model behaviour over time using *s* while others use *ms*. System supported unit-conversion between compatible units would allow the use of elements from one model in another.

Finally we note how types and units can be used to document and specify model components and interfaces, particularly with respect to generating reusable and replaceable model modules, a key aim of our research as discussed below.

2.3.3 Modularity and Reuse

The evolution of cardiac models (described in Section 2.1.3) is defined by the continuing development and reuse of ionic current models within the cellular membrane to comprehensively describe an AP, motivated by newer experimental research and data [107]. We aim to mirror this process within the DSL, enabling modellers to reuse components in an easy, safe and extensible manner through modules. This will enable the construction of complex models from reusable components that may be replaced and refined as necessary. For instance, the O'Hara model [109] utilises the Hodgkin-Huxley formulation for current equations that require computational efficiency, yet suggests that currents or fluxes may be replaced by more complex Markov-based formulations to investigate mutation or drug effects as needed.

Such submodel component reuse is explored in [105] where components representing ionic currents from two models, with a common ancestor, are switched on a piecewise basis and the composite model simulated. Within our DSL we intend to provide the capabilities to reproduce the methodology from this paper in an automated rather than manual manner, using interchangeable modules within a cardiac modelling framework — this research forms the basis of our model development and simulation study in Section 4.4. This will demonstrate the benefits of modular programming, allowing abstract and disparate model elements to be integrated within larger models to perform *in silico* experimentation. It will also provide modelling benefits, allowing creating of model repositories that can be extended and reused, and potentially improve biological understanding by facilitating investigations into common ancestry of models, their relation to experimental data, and guide future research and experiments.

Biological model modularity has been investigated at the systems biology level in [97] using

a Lisp-based macro system and in [141] using the restricted CellML import mechanism; we intend to build upon this in a comprehensive, holistic manner with direct DSL support for typed modular repositories. Furthermore, we wish to parameterise submodels, such that they may be replaced, specialised and modified for use within larger models; providing a greater potential for model reuse. We envisage several forms of modelling specialisation, (1) altering the parameters or equations of the component, (2) altering the interface by which the components communicate, or (3) creating new components from composite behaviour.

Models often exhibit patterns of containment and encapsulation; this can be captured by encapsulating logic related to a particular model component within software structures that express the logical model design/hierarchy. Abstraction is the development against opaque interfaces rather than directly manipulating the underlying model code. This enables loosely coupled components to be created, facilitating the development of an interchangeable system. Combined encapsulation and abstraction may provide an interface-driven, modular style of model development that enables the construction of complex models predicated upon abstracted components. Such structures are commonly employed when constructing large-scale software systems, and are inspired by architectural patterns found in the real-world [4]. However an analogy can also be drawn to the observed behaviour of biological models and systems, e.g. the 'interface' presented by an ion channel on a cardiac cell membrane to the extra-cellular space.

The creation of complex systems from the composition of reusable components is common in software engineering [50]. An entire family of design patterns has been created and applied to large-scale construction of object-based systems [51], and we believe many of these may be applicable to the development of component-based biological models. We will investigate their application using our DSL and examine the best practices that emerge in the cardiac and physiological modelling domains. This includes object-oriented (OO) mechanisms such as interfaces, object-hierarchies and subtyping; component-based aggregation and specialisation; and late-bound flexible architectures that utilise dependency injection, 'mocking' and proxying structures. As an example, we envisage the use of singletons to hold model parameters, abstract factories to construct and replace model subcomponents, and the observer pattern to pass messages between biological components [51].

We hope to form our own initial pattern language [16] that may be best used and applied to

the domain of physiological model development. This may be referenced by modellers to provide identifiable structures and techniques when constructing models, reusing domain knowledge.

2.3.4 Model Verification and Validation

Computational model verification and validation is conducted to ensure that the implementation of the model, and the conceptual model itself, is correct with respect to the system at large. Verification is defined as 'the process of determining that a computational model accurately represents the underlying mathematical model and its solution' whereas validation is defined as 'the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model' [65]. Succinctly we may say that verification is 'building the model right' and validation is 'building the right model'. By definition verification must occur prior to validation in order to separate errors due to model implementation from uncertainty due to model formulation. The verification and validation process is an iterative process continuously refined during development in response to comparisons with the system.

The object of model verification is to ensure that the computational model implementation is correct, and is used to compare this representation to the conceptual model. The process allows users to test the model to find and fix errors within the implementation, e.g. are the parameters and expressions of the model correctly represented. Techniques used to verify model implementations include logical flow diagrams and examining the model output under various input parameters. Methods from software verification are often also applicable, e.g. formal verification methods such as model checking, proof assistants, and type-systems, along with engineering-based techniques such as code conventions and verification through testing [74, 129].

Model validation is used to determine that a model is an accurate representation of a system. It ensures that the model is built for a specific purpose and determines its validity for that purpose. Validation is usually achieved through model calibration, an iterative process of comparing the model to system behaviour and using the discrepancies, and the insights gained, to improve the model until the accuracy becomes acceptable. Thus, for computational mathematical biological models, validation involves comparing the model simulation to experimental data to determine if the results are acceptable. This may involve sensitivity analysis, comparing input-output transformations on the model and system itself.

We intend to research several features that facilitate model verification and aid the process of model validation, investigating techniques used to verify software within the domain of computational mathematical modelling. We do not consider highly-formal verification methods such as model-checking and automated proof assistants, and instead are concerned with lightweight, unobtrusive, verification and validation methods. This includes type checking (see Section 2.3.2), units-of-measure systems, modular development, and facilities for scripting simulations to enable repeatable results and functional curation.

2.3.4.1 Model Testing & Functional Curation

Software must be tested, verified, and validated to ensure it is correct with respect to its specification; we may apply the same testing methodologies to models to ensure their functional correctness. Integrating models with experimental data is a key issue in computational mathematical modelling. We would like to investigate how a model responds to the presence of pathologies or pharmacological agents; this would require a testing framework and a method for comparing expected results with those obtained. Similarly we require confidence in our model simulations and wish to be able to verify and validate them; results should be reproducible under the same protocol and easily compared to models of the same biological system. Achieving this would allow us to compare differing modelling methodologies against experimental data and perform *in silico* experiments in response.

This goal is termed *functional model curation* [32], the central aim being that a model is continuously tested and calibrated against experimental data during development and curation. This requires abstracting and applying the same experimental protocols to the model during development whereby simulation results and experimental data can then be compared. This increases confidence in a model's behaviour, and allows extension and reuse to occur in a verified, validated, and tested fashion.

The process is similar to test-driven development and continuous deployment processes from agile software engineering [74]. When using these methods, tests to verify expected behaviour are created first. As development progresses the system is repeatedly constructed and verified against the tests in an independent environment, providing correctness assurances and guarding against regressions. When applied to functional curation, we can consider the generation of valid simulation results as the expected behaviour, with the test being if the results

and experimental/reference data can be successfully compared under the environment defined by the experimental protocol. The use of functional curation demonstrates how software engineering techniques may aid the computational modelling discipline.

2.3.5 Summary

We introduced software engineering as a discipline that applies both scientific and pragmatic engineering methods to the software development process. From this we introduced relevant practices from software engineering used to construct reliable and extensible software systems.

We discussed how DSLs facilitate the high-level development of systems and models. This included a brief overview of existing biological modelling DSLs and a critique of their abilities to facilitate reusable, large-scale model development. The benefits of strong typing, modularity and component-based architectures were described, these may be used for developing large-scale models comprised from reusable, abstracted subcomponents in a manner similar to OO design. Finally we looked at model testing through the use of functional curation to validate models against each other and experimental data, facilitating reproducible results. Such attributes will be utilised during the design of our DSLs to enable large-scale model development in a modular, reusable manner including an investigation of potential design patterns and best practices.

2.4 Computational Simulation

Within this section we detail our motivation for requiring high-performance simulation of complex cardiac models. We define high-performance simulation as optimised simulation on a single workstation CPU core that makes efficient use of CPU resources. We focus on such desktop hardware as it is commonly used during model development and many simulation systems operate on such machines. Such desktop CPU architectures are often used in supercomputers[4], and thus the techniques will be directly transferable to individual cores on such machines for parallel use within multi-scale simulations. An overview is provided of current simulation packages and the methods they use to achieve efficient simulation. The software implementation of DSLs is examined with a focus on optimisation, high-performance simulation and efficient use of modern hardware resources.

2.4.1 Motivation

Computational efficiency is always a concern and within our chosen domain is important for several reasons. Firstly the time required for a single simulation is significant during (interactive) model development or when performing experimentations into model behaviour. Additionally, performing sensitivity analysis, parameter estimation and/or fitting to experimental data requires a significant number of simulations due to the iterative nature of the methods. Improving computational efficiency will enable modellers to create more detailed models containing fewer approximations, that may then be integrated within larger, multi-scale models.

The need for efficient simulation becomes more apparent with the increasing complexity of models [53], e.g. the IMW04 ODE-based cardiac model [75] requires over 10 minutes to simulate to a 'steady-state'[5] on a modern desktop PC. To reduce computational requirements some systems purposefully model ionic currents using HH-type channel formulations, or alternatively provide approximations to Markov-formulation derived ionic current equations through quasi-steady-state assumptions [134].

Computational efficiency is also highly important when simulating stochastic models. For instance, simulation of a complex Markov-formulation ionic current using the direct-SSA requires

[4] As seen on the Top 500 list at `http://www.top500.org` and with Intel's Many Integrated Cores initiative.

[5] A state reached where the initial transients of a non-linear ODE system have decayed away.

many simulation trajectories to approximate the probability distribution, incurring a significant computational expense. We believe custom code-generation and model specific optimisations may help provide greater simulation efficiency.

2.4.2 Current Simulation Systems

Often models are initially developed as *ad hoc* systems within mathematical environments such as MATLAB. The model description and simulation code is mixed and is simulated using the environment's built-in functions (for instance MATLAB includes several of the ODE solvers listed in Appendix B.1.1). When the simulation of a model is deemed sufficiently slow the system is completely rewritten in a natively compiled language, usually C/C++/Fortran; increasing the computational performance is often achieved at the cost of model simplicity and reuse.

As an alternative to *ad hoc* code, modellers may use a variety of existing simulation packages that support current modelling formats and/or DSLs (see Section 2.3.1.1). Two simulation approaches exist in such environments. Mathematical environments built within MATLAB and R[6] generally load data files/structures encoding the model at run-time and construct internal structures that are interpreted by the simulation algorithm. Alternatively, Chaste, CVODE (see below) and many other high-performance simulation systems provide a native library and API that a model must implement. This model representation is then compiled and executed alongside the simulation library.

2.4.2.1 Chaste

Chaste[7] is a C++ framework capable of simulating cellular and organ models comprised of DEs [120]. It includes high-performance ODE and PDE solvers and has extensive support for cardiac simulation and analysis. It enables the mathematical modelling of the electrophysiological activity of single cells and whole organs from a continuous, deterministic view. As a native-code environment Chaste requires models to be developed as C++ classes that are then compiled and linked with the Chaste framework. Source-based converters such as PyCML[8] can translate CellML models to this API whilst performing verification, unit-checking and optimisations [31].

[6] `http://www.r-project.org/`
[7] `http://chaste.comlab.ox.ac.uk`
[8] `https://chaste.comab.ox.ac.uk/pycml`

2.4.2.2 CellML environments

OpenCell[9] is the 'official' CellML modelling environment and also implements high-performance simulation via native code compilation and linking [52]. Included within the OpenCell package is a self-contained C compiler (`gcc`) and runtime environment. Prior to simulation OpenCell translates the CellML model into C using the provided CellML API, compiles it to native code and links to the well-optimised CVODE ODE library (see Appendix B.1) to perform adaptive numerical simulation [27]. This has now been succeeded by the OpenCOR environment[10] that performs a similar task, using the same CellML API to convert to C and includes an integrated version of the Clang C compiler[11] instead.

Finally, COR[12] is another well-known CellML modelling and simulation environment that implements high-performance simulation via native code compilation of the model data [53]. Native code is obtained through compilation-via-C or through a simple inbuilt native-code generator. This model representation is again linked to CVODE for efficient simulation.

2.4.3 Native Code Generation and Implementation

2.4.3.1 Compilation-via-C

Obtaining efficient simulation in most modelling environments and implementations is achieved by translating models to a C representation and compiling, as demonstrated by the above CellML environments. There are advantages to this approach; C is a simple low-level language with well-performing compilers, it removes the need for an explicit language implementation, and the model gains access to the range of structural and numerical optimisations available during C compilation. However it requires translating a model to C, losing much of the structure of the model, and conformation to the C language semantics, which may not be appropriate for the domain. For instance C assumes a stack-based memory model, making functions calls more expensive at the CPU level. Furthermore C does not provide direct access to several CPU features, such as the state of floating-point exception flags or vector registers that we may wish to utilise to increase performance. The compilation is a static process performed prior to simulation,

[9] `http://www.physiome.org.nz/cellml/tools/opencell`
[10] `http://www.opencor.ws`
[11] `http://clang.llvm.org`
[12] `http://cor.physiol.ox.ac.uk`

forbidding dynamic changes within the model. Finally the range of the domain and model-specific optimisations that may be performed during compilation are now restricted to those provided by the host-language.

2.4.3.2 DSL Implementation

DSLs may be directly interpreted according to the semantics of the language or compiled to a lower level form, perhaps machine code, prior to execution. An interpretive approach enables rapid prototyping and ease of development, with the result that evaluation speed is hindered due to the interpretive overhead of each command.

However, high-performance simulation requires the use of native code, such as compilation-via-C as described above. Yet to obtain the highest possible simulation performance we must be able to perform domain and model-specific optimisations and generate directly model-specific native code rather than compilation-via-C or comparable low-level languages. For instance, the mutable memory model used by C can cause aliasing issues that disable several optimisations; whereas we may wish to compile via representations more suited to program analysis and optimisation, such as Static Single-Assignment (SSA) [6].

During native compilation, DSLs may be iteratively and statically compiled to lower-level forms, potentially arriving at a machine code representation suitable for direct and efficient execution. At each stage of the compilation process, the abstract syntax representation of the program may be converted into more intermediate representations. These compilation stages may occur prior to execution, and/or during execution using a *just-in-time* (JIT) compilation approach. This staged compilation permits a range of optimisations at each stage depending on new static or run-time information that in turn guides the compiled output [3].

2.4.3.3 Low-level Optimisations

Domain-specific, model-specific and low-level optimisations are required to generate efficient simulation code. Potential low-level optimisations during compilation include common sub-expression elimination, constant evaluation, and function inlining [3, 5]. A typed DSL may utilise type-erasure during implementation and replace typed value representations with the low-level 'unboxed' values once the program has been successfully type-checked. For efficient execution the final machine code must contend with a range of issues including register allocation, cache

usage, instruction selection and scheduling, and memory management [21].

High-performance simulation implementations must also utilise modern hardware features and developments to optimally harness all available computational power. These include optimising for larger and faster on-die caches, increased and wider registers, and multiple shared memory cores on custom high speed interconnects [113]. Vector-processing units are common on modern CPUs, also known as Single-Instruction Multiple-Data (SIMD) systems, and facilitate performing the same floating-point (FP) and integer operations simultaneously on several data elements. They may be used to perform atomic numerical operations in parallel on packed vectors, a technique termed Instruction Level Parallelism (ILP) [11]. High-performance computing clusters are often required for large-scale numerical computation, such as when performing multi-scale whole-organ simulations. These provide independent-memory computing over high-latency, relatively low-bandwidth network connections, potentially including the Internet.

2.4.4 Summary

This section described why high-performance simulation is necessary for complex model simulation of deterministic and stochastic models. The techniques used to accomplish this in current environments, generally native-code compilation-via-C, were detailed and the benefits and drawbacks listed. Several low-level details regarding native-code DSL implementations were outlined, including an overview of procedures, optimisations and modern-hardware constructs available to greatly improve run-time efficiency and simulation performance.

The introduction of high-performance techniques to ion channel modelling will enable the creation of more complex and accurate models that further capture ion channel dynamics and their effect on the AP. A high-performance approach may allow use of hybrid-stochastic models within whole-organ simulations or real-time simulation [8].

2.5 Summary

This chapter discussed the background material relevant to this thesis as detailed in Section 1.1. Cardiac cells were discussed from a biological and modelling perspective, focussing on modelling ion channels and their effect on the cellular AP. We presented an overview of several cardiac cell models used within this thesis, including an in-depth look at the HH52 model that introduced the structure of cardiac models. Multiple formulations of ion channels were presented and modelling methodologies discussed.

Chemical kinetics and the CME were introduced to model the behaviour of biochemical reactions. The SSA was introduced as a means to simulate chemical kinetic reactions. However, computational complexity necessitates the use of approximate algorithms, particularly for large models. These ranged from stochastic discrete to continuous deterministic approaches that exhibit differing properties and use-cases, culminating in a hybrid modelling approach that may simulate model subsections using these differing approximations.

A brief overview of software engineering practices commonly used to develop large-scale software systems was provided. This included references to software architecture and design patterns, and the need for encapsulation and abstraction to aid re-usability. We detailed how these may be useful when developing reusable, reliable models in contrast to non-extensible, tightly coupled models developed by mathematical modelling communities at large. The benefits of DSLs over existing modelling methods were outlined, and we discussed the shortcomings of the CellML and SBML biological DSLs with regards to developing well-engineered models.

The chapter concluded with an overview of simulation implementations, presenting the motivation for high-performance simulation within the domain, the methods used in existing simulation packages and the software and hardware aspects — including native compilation, vectorisation, and low-level issues.

Having reviewed and introduced our modelling domain and the benefits that software engineering methodologies could bring to the domain, we now discuss our research efforts to date. In the following chapters we investigate and guide the creation of a DSL, termed *Ode*, to model biological mathematical systems, e.g. ion channels within cardiac cells, from multiple modelling perspectives. The DSL will facilitate usage of modern software engineering methodologies to

create reusable, extensible models. It will provide high-performance simulation via a run-time based, staged, native code generator that creates optimised, model-specific, simulation code. The next chapter introduces the *Ode* modelling language, discussing research into enabling the efficient and verified creation of, initially ODE-based, mathematical biological models, and into facilitating repeatable model simulation.

Chapter 3

Ode Modelling Language

In this chapter we present our initial research into the design of the *Ode* modelling language, and provide details of its implementation and usage in modelling and simulating mathematical biological models. The DSL include features suited to biological modelling with an emphasis on techniques from software engineering such as program verification, reuse and modularity.

The core computational constructs within the language are introduced on a piece-by-piece basis detailing their syntax, high-level semantics, usage and examples. We introduce the modelling constructs present within the DSL that are used to model continuous deterministic systems composed of ODEs. This includes an overview of the modelling process and general simulation scheme, again with several example models. Significantly more detailed and complete models may be found in Appendix D.4, alongside CellML versions for comparison in Appendix D.3.

Ode is used as a lightweight frontend with syntax bearing a superficial similarity to scripting programming languages commonly used for modelling. We introduce the *Core* intermediate representation (IR), based on the lambda-calculus [7], used to describe models internally. The *Core* IR is utilised within many stages of the front- and middle-end of the compiler pipeline, e.g. type-/unit-checking and module resolution. In Chapter 5, we detail the conversion from *Core* into the backend IR, termed *CoreFlat*, an extremely simple and restricted numerical language suitable for direct simulation on a variety of hardware platforms. Appendices A.3 and C.1 describe the conversion semantics for each stage.

We present mechanisms for model verification, including a fully-inferring compile-time type system that ensures models may be simulated without error. This is extended with a powerful, programmable yet lightweight units-of-measure system to ensure all model calculations are

dimensionally consistent, and introduces several novel enhancements of its own.

We finally describe the construction of a command-driven interactive environment to configure and execute simulations within the system. This provides several benefits to modellers arising from the separation of model definition and configuration and may be utilised in interactive, batch and programmable modes. It provides opportunities for rapid model development, simulation and iteration.

3.1 Language Overview — Syntax & Semantics

Ode is a small, numerical DSL influenced by our experience with CellML [93], SBML [73], and functional programming languages [136]. It is designed to enable the creation of mathematical models typically used to represent biological systems in an easy-to-use environment. The overall design principals are to reuse modeller familiarity with existing scientific computing environments and high-level languages such as MATLAB and Python, and to infer information rather than explicitly require it from users. This avoids cluttering the model description and also keeps the system easier to use by non-programming experts. To place the model structure and computations within a formal context, the language bears a resemblance to the simply-typed lambda calculus [119] extended with features necessary for describing mathematical models computationally.

We designed *Ode* as an independent, external DSL, rather than utilising existing functional languages such as OCaml and Haskell for several reasons. Firstly, it allows us to integrate modelling constructs such as ODEs (and later, SDEs and SSA-reactions) as first class constructs within the language, with full support from the type and module systems. Furthermore it allows full control of the simulation process, permitting the introduction of domain-specific optimisations and custom code-generation (see Chapter 5). Finally, we feel implementation on top of an existing functional language would require modellers to posses some knowledge of the underlying language semantics, creating a high-barrier for the intended audience.

As might be expected *Ode* includes support for numbers, functions, expressions, and variables declarations; along with support for ODEs, SDEs and SSA-reaction equations (the latter two being described in Chapter 6). *Ode* has a strict call-by-value evaluation strategy in order to reduce code-generation and run-time complexity whilst increasing performance. The system is designed to be largely immutable, where variables are all constant and may not be modified after their initial declaration. This provides two major benefits: relieving the modeller from explicitly managing model state; and providing greater freedom for the implementation to alter the model definitions. The language features a simple type system (described in Section 3.3.1) implemented at compile-time and used to ensure several aspects of model correctness and as an implementation optimisation. As *Ode* is designed for high-performance simulation, run-time

typing is not permitted and all values are statically-allocated and stored as raw, unboxed, values within memory.

The language aims to present a system for structuring and specifying models composed from the numerical computations that occur during each timestep of a simulation in a flexible manner. To accomplish this, several abstractions are evaluated at compile-time to generate optimised model code for within the simulation loop. In the following sections we describe the basic features of the language used to compute and structure (primarily numerical) expressions. They are demonstrated using code snippets from the HH52 model as discussed in Section 2.1.3.

3.1.1 Values and Expressions

```
1 val E_K = E_R - 12
2 val i_K = g_K * pow(n, 4) * (V - E_K)
```

The code fragment above demonstrates the construction of expressions in the language, the result of which may be assigned to a named identifier utilising the `val` construct, as `E_K` and `i_K` are. Value definitions may occur anywhere within a model, defining an immutable value bound to the result of evaluating the expression on the right of the equals sign. An _ (underscore) may be used in place of a name for a value, this is useful when working with aggregate structures where not all values are required.

The expressions themselves may be atomic values, expressions (e.g. numerical or logical), the result of a function call, etc., with a syntax similar to most programming languages. They are the primary means for constructing the calculations that represent the body, i.e. f, in an ODE system defined as $y' = f(t, y)$.

Numerical terms and expressions constitute the majority of expressions within a model, operating on double-precision floating-point (FP) values. Integers are not supported within the system however we can use doubles to safely represent signed integers up to $\pm 2^{52}$. Boolean terms are present for use with the built-in logical and relational operators, a full list of operators is provided in Appendix A.2.1.

The system includes many built-in operators and functions that can be utilised within expressions and assigned to values, as can the results of user-defined components. This includes access to many numerical functions taken from the C-based mathematics library, `libM`; these functions

are listed in Appendix A.2.2. The current value of the independent variable t (*time*) during simulation can be accessed using the term `time`. This may be used to perform time-dependent computations, such as generating a stimulus current at specific intervals within a cardiac model, as seen in Listing 3.1. The unit term, `()`, may be used to describe an empty value, similar to the `void` or `None` type in many programming languages — it does not evaluate to anything useful and may be used with components that do not accept or return values.

Values obey lexical scoping rules, and multiple nested lexical environments may be created using braces to abstract complex model calculations without cluttering the outer namespace.

3.1.2 Components

```
1 val E_R = -75
2 // component to calculate membrane leakage current
3 component calc_i_L(V) {
4   val g_L = 0.3
5   val E_L = E_R + 10.613
6   return (g_L * (V - E_L))
7 }
8 val i_L = calc_i_L(V)
```

The sample above demonstrates the use of a component to abstract the calculation of I_L within `calc_i_L`; this component takes the current membrane voltage, and calculates and returns I_L which is assigned to a new value (as modelled in Section 2.1.3). Components represent the primary mechanism for structuring and abstracting expressions in a model within the DSL. They are influenced by the component system within CellML but adapted to work in a more programmable setting; where a component may represent a logical containment or encapsulation of a biological process, e.g. an ion channel.

As *Ode* is an operational language, they may be used for more common computational purposes than in CellML, such as to group, abstract and parameterise repeated computations. They are analogous to functions, transforming an input set of values to an output set and abstracting commonly utilised operations to increase readability, modularity and reuse. Multiple values can be passed and returned by utilising the tuple and record features described in Sections 3.1.4 and 3.1.5 for defining aggregate structures.

An important restriction we impose upon components is that they cannot be recursive. Components are also fully inlined during compilation, to improve simulation performance. When used

with the stateful modelling features, described in Section 3.1.6, it can provide for programmable, compile-time, model construction in a manner similar to LISP and Scheme macros [130].

Similar to values, components are lexically scoped, may be nested and access values from their outer-lying scopes, providing logical containment and encapsulation — such properties are observed in biological systems and models. CellML in particular has explicit support for defining containment and encapsulation abstractions, but requires manual linking of component input and output parameters. However by utilising component-calling and scoping semantics similar to those found in general-programming languages, the DSL can provide such functionality naturally, resulting in significantly simpler and smaller model definitions.

3.1.3 Piecewise Terms

```
val i_Stim = piecewise {
  time > 10 and time < 10.5 : 20,
  default : 0
}
```

The example above demonstrates the use of a piecewise statement assigned to `i_Stim`, where it is used to model the conditional stimulus protocol at various times during the simulation of a cardiac model to initialise an AP. Piecewise terms are a basic construct that enable conditional control-flow similar to a *switch* statement. They allow execution of a single expression chosen from the evaluation of a selection of conditional/logical expressions in a top-down order during each simulation timestep. Multiple conditions are allowed, and the construct terminates with a required default expression evaluated if all conditions fail. A deliberate limitation is the omission of looping control-flow constructs. When coupled with the recursive restriction on components this ensures program termination when constructing and simulating a model.

3.1.4 Tuples

```
component calc_i_K(tup) {
  val (V, n) = tup
  val E_K = E_R - 12
  val i_K = g_K * pow(n,4) * (V - E_K)
  return (i_K, E_K)
}
val tup = (-75, 0.325)
val (b, _) = calc_i_K(tup)
```

Tuples are an aggregate structure used to 'pack' heterogeneous expressions into a single object. They may be used to both group and pass commonly-related values around a model, e.g. passing and returning multiple values into components, and can be nested as required. The above example uses a tuple to pass two parameters, `V` and `n` to the `calc_i_K` component that calculates I_K from the HH52 model. On line 7 a tuple `tup` is created, from a comma-separated list of expressions in parentheses, and is used in Line 8 to pass multiple values to the `calc_i_K` component. Lines 2 and 8 demonstrate how we may use the multiple values within a tuple, the two elements are extracted and given new names.

3.1.5 Records

```
1 component calc_i_K(rec) {
2   val E_K = E_R - 12
3   val i_K = g_K * pow(rec#n,4) * (rec#V - E_K)
4   return (i_K, E_K)
5 }
6 val rec = { V : -75, n : 0.325 }
7 val (b, _) = calc_i_K(rec)
```

The tuple syntax is extended to support records, these are essentially named tuples that have a have a unique identifier for each element and observe a lexical ordering. We can place any expression into a record accompanied with a named field; similar to tuples this aggregate object can be used in the model as a single structure. The above code example is based on the tuple example from the previous section, using records instead. Line 7 demonstrates the creation of a record with two elements, this is passed to the `calc_i_K` component as before. The individual elements from a record can be accessed utilising the `#` operator and the field name, as demonstrated by `rec` above on Line 3 to calculate I_K.

We believe tuples and records will prove highly useful when dealing with logical/biologically-related data utilised throughout a model simply without complicating the model code, e.g. holding multiple constant parameters such as the model temperature, membrane area, and so on.

3.1.6 Modelling Features

We extended the basic numerical expression language with several constructs that support the mathematical modelling of biological systems — starting with the introduction of stateful values

that initialise and hold the model state. Following this, ODE constructs were introduced to support the creation of continuous, deterministic biological models, such as conventional cardiac electrophysiological models. With these extensions the *Ode* DSL may be used to model the state-dependent computations that occur within a single timestep of a biological model.

3.1.6.1 Initial Values

```
1 init V = -75
```

We introduce the concept of initial, stateful, values to the DSL; this is demonstrated by `V` in the code above that configures the initial value of the stateful membrane voltage. As their name suggests they enable description of the initial values of a system, i.e. $y(t_{start})$, where y is an unknown function to which we wish to generate an approximation solution, and t_{start} is the value of the independent variable *time* at the start of a simulation. Initial value problems are described further in Appendix B.1 and comprise the majority of mathematical biological models.

The initial values within the model code implicitly define the size of the system that we approximate during simulation. They may be placed anywhere within the model code, increasing model cohesion as opposed to CellML where all initial values must be explicitly defined and located at the start of the file/component. The initial values are effectively stateful values that change at each timestep h to reflect the approximate value of $y(t_n)$; they can be referenced and utilised within further computations within the model in the same manner as normal, immutable, values.

Initial values are not limited to a literal number and can be any complex expression that evaluates to a number, allowing for a wide variety of systems to be configured and customised by their initial values prior to simulation.

3.1.6.2 Deterministic Modelling (ODEs)

```
1 init V = -75
2 ode { initVal : V, deltaVal : dV } = -(-i_Stim + i_Na + i_K + i_L) / Cm
```

ODEs were added to support the creation of continuous, deterministic biological models, such as conventional cardiac electrophysiological models. The code above demonstrates use of the

construct to model the changes in membrane voltage due to ionic currents, this is taken from HH52 voltage calculation in Fig. 2.5.

An ODE definition requires a reference to an initial stateful value, i.e. y, and an expression defining y'[1] (termed the delta expression). This expression represents the equation for the derivative of the unknown function in first-order standard form, i.e. $y' = f(t, y)$. The expression can utilise any values in scope and must evaluate to a float. An optional identifier for this value (y') can be created by defining `deltaVal`. This may be referenced and reused if needed within later computations, for instance within another ODE definition.

Having described y' in standard form, and with an approximate solution $y(t_n)$ (starting with the initial value $y(t_{start})$), we can evaluate the operations at the current time t_n and determine y' at this iteration with respect to the current environment. This may be used to approximate $y(t_{n+1})$ using a variety of explicit numerical methods described in Appendix B.1.1. The system updates the linked initial value y implicitly, restricting access to stateful, mutable behaviour to the ODE solvers only and distinct from the model itself. In this manner we may numerically approximate $y(t)$ and simulate the ODEs that comprise a potential biological model.

3.1.6.3 Simulation Loop

With these modelling constructs the *Ode* DSL may be used to express and structure the state-dependent computations that occur within a single timestep during numerical simulation of a biological model. The `init` values set the size and initial state of the system, and the `ode` definitions describe how it is updated at each timestep. Both definitions may be placed within components to parameterise, abstract and reuse them, as basic `val` expressions can be. However it must be noted that the DSL does not support dynamically changing the size of the system during simulation. We partially-evaluate and inline all component calls to statically determine the system size during compilation. This process also occurs across module boundaries (see Chapter 4) and is described in Section 5.1. It results in a structureless, static, optimised system containing only the numerical calculations required to determine the system state at the next timestep.

The simulation semantics of this constructed model are those of a strictly evaluated, numerical data-flow system. The model represents a transformation over the model inputs (state-dependent

[1] y' is a shorthand form for $\frac{dy}{dt}$.

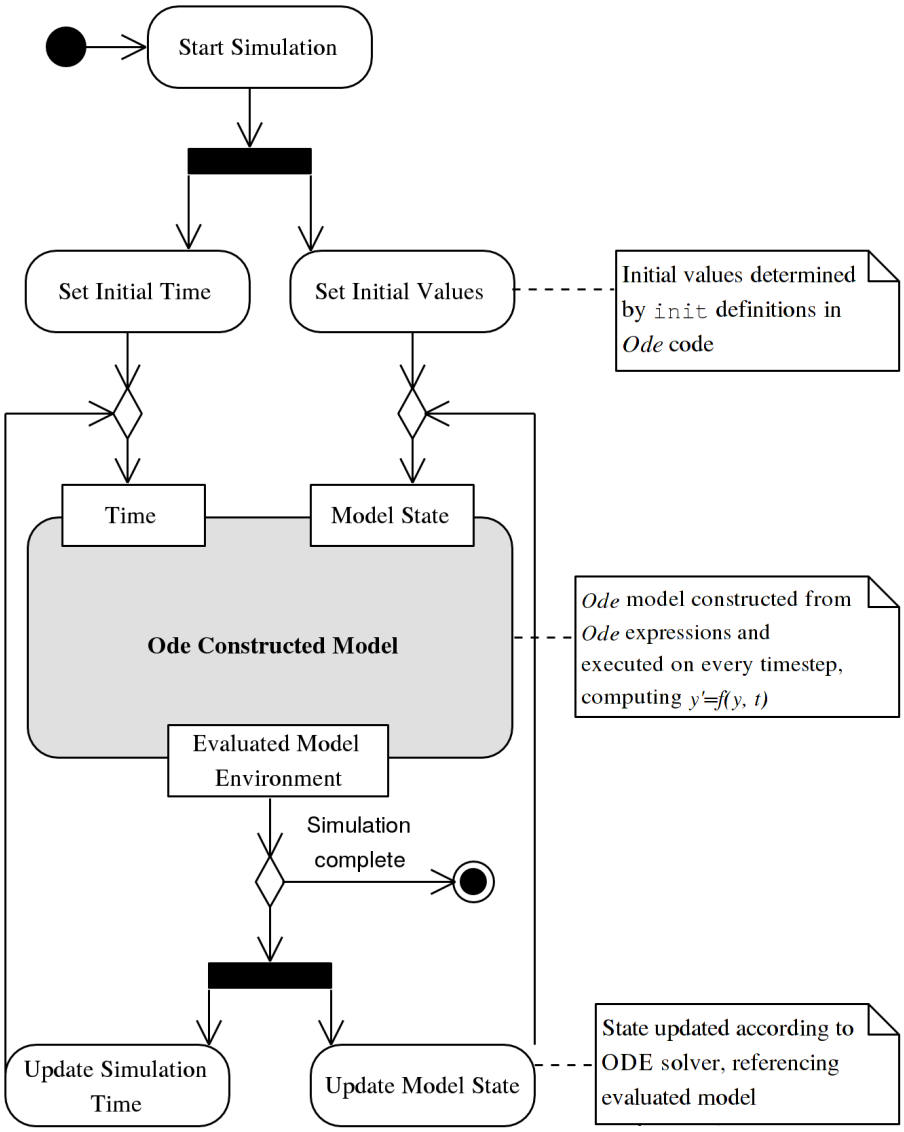


Figure 3.1: Overview of the *Ode* simulation structure — the *Ode Constructed Model* is generated from the user model code. This is evaluated within the simulation loop on every timestep using the current model state and time as inputs. The resulting evaluation environment is used to update the model state according to the chosen explicit solver, e.g. the forward-Euler ODE solver.

values defined by initial values and ODEs) to the model outputs (updated state values) with respect to the externally updated independent parameter *time*. This external parameter is explicitly accessible within the model environment and is updated at each timestep. Fig. 3.1 presents a high-level illustration of the simulation structure imposed by the *Ode* system. Comprehensive model verification, described in Section 3.3, and the lack of looping control-flow constructs in the language ensure that the constructed expressions will successfully evaluate and terminate during each simulation timestep. The lack of looping control-flow constructs does reduce the language expressibility, however it has not affected the construction of cardiac models thus far and we are unaware of any models within the domain that would require such functionality.

$$
\begin{aligned}
\mathit{odeStmt} &\leftarrow \mathit{modStmt} \mid \mathit{quantityDef} \mid \mathit{unitDef} \mid \mathit{convDef} \mid \mathit{exprStmt} \\
\\
\mathit{exprStmt} &\leftarrow \mathit{componentDef} \mid \mathit{valueDef} \mid \mathit{initValDef} \mid \mathit{odeDef} \\
\\
\mathit{componentDef} &\leftarrow \texttt{component}\ f\ (\mathit{id}_1, \ldots, \mathit{id}_n)\ \mathit{blockStmt} \\
\mathit{blockStmt} &\leftarrow \{\mathit{exprStmt}_1, \ldots, \mathit{exprStmt}_j,\ \texttt{return}\ (E_1, \ldots, E_n)\} \\
\\
\mathit{valueDef} &\leftarrow \texttt{val}\ \mathit{id}_1, \ldots, \mathit{id}_n = E \mid \mathit{blockStmt} \\
E &\leftarrow t_1 * t_2 \mid t_1 / t_2 \mid t_1 > t_2 \mid \ldots \mid t_1\ \&\&\ t_2 \mid t\ [\mathit{unitCast}] \\
t &\leftarrow (E) \mid \mathit{num}\ [\mathit{unitCast}] \mid \mathit{bool} \mid \mathit{piecewise} \mid f(E_1, \ldots, E_n) \mid \texttt{time} \mid () \\
&\quad \mid \mathit{id}\ [\#\ \mathit{id}] \mid (E_1, \ldots, E_n) \mid \{\mathit{id}_1 : E_1, \ldots, \mathit{id}_1 : E_1\} \\
\mathit{piecewise} &\leftarrow \texttt{piecewise}\ \{E_{C1} : E_{T1}, \ldots, E_{Cn} : E_{Tn}, \texttt{default} : E_{def}\} \\
\\
\mathit{initValDef} &\leftarrow \texttt{init}\ \mathit{id}_i = E \mid \mathit{blockStmt} \\
\mathit{odeDef} &\leftarrow \texttt{ode}\ \{\texttt{init} : \mathit{id}_i,\ [\ \texttt{deltaVal} : \mathit{id}_{delta}\ ]\ \} = E
\end{aligned}
$$

Figure 3.2: *Ode* language grammar for the core modelling language using a simplified variant of EBNF, where square brackets indicate optional constructs. The grammar assumes common production rules, including operator precedence and whitespace. Greyed-out production rules are described elsewhere in the thesis.

3.1.7 Grammar & Example

Fig. 3.2 presents the grammar for the core modelling *Ode* language using Extended Backus-Naur Form (EBNF). Listing 3.1 demonstrates several of these features with a larger sample from the HH52 model complete with annotations. The sample suggests several patterns we have used for structuring electrophysiological models, using components and nesting to capture biological concepts such as interfaces and containment respectively. Sample *Ode* and equivalent CellML models are provided in Appendix D.4 and Appendix D.3 respectively.

Listing 3.1 A example of the *Ode* syntax, annotated with comments, to highlight several modelling features and potential structural patterns. Model is adapted from the HH52 electrophysiological cell model listed in Appendix D.4.1

```
val E_R = -75 // Cell resting potential
val Cm = 1    // Cell membrane capacitance

/* Main HH52 Cell Component that defines the membrane voltage *************/
component membrane(_) {
  init V = -75 //Membrane voltage initial value, approximated over simulation

  /* Nested component returns the membrane leakage current ***************/
  component calc_i_L(V) {
    val g_L = 0.3           // set leakage capacitance
    val E_L = E_R + 10.613  // set leakage potential
    // calculate and return leakage current, using membrane voltage V
    return (g_L * (V - E_L))
  }

  // Stimulus Current, triggered every "period" ms
  // use `piecewise` to determine the trigger and set i_Stim
  // N.B "%" is the mod operator
  val period = 60
  val i_Stim = piecewise {
    time % period >= 10 and time % period <= 10.5 : 20,
    default : 0
  }

  // Ionic currents - (i_Na and i_K not shown)
  val i_L = calc_i_L(V)
  // ODE to model V
  // Calculated from sum of ionic currents over membrane capacitance
  ode { initVal : V, deltaVal : dV } = -(-i_Stim + i_Na + i_K + i_L) / Cm
  return (V)
}

val V = membrane(()) // Top level value assigned to the membrane voltage
```

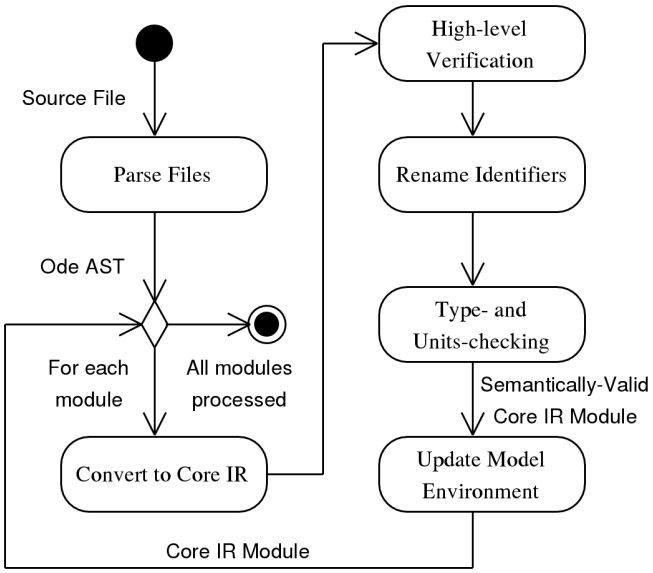


Figure 3.3: Main stages of the compiler front-end. From an initially-specified module, all referenced source files are parsed, converted into the *Core* IR, and undergo verification and transformation. The module environment is updated with the transformed model for potential simulation.

3.2 Implementation Details

This section describes several internal implementation details regarding the internal structure of the *Ode* system. The *Ode* DSL is implemented using the Haskell functional programming language with the GHC implementation on Linux[2]. The compilation pipeline for the front-end and middle-end of the compiler is presented, this utilises both the *Ode* AST and an internal intermediate representation (IR) termed *Core* that is more amenable to future optimisations and execution. *Core* is used as the IR within the compiler middle-stages upon which many operations and functionality is defined, this includes the module system and model verification stage described in the next section.

3.2.1 Compilation Pipeline

Conversion from *Ode* to *Core* occurs at the module level, described in Chapter 4, and is a straightforward mapping. The resulting *Core* model AST passes through several compiler stages and undergoes a series of verifying transformations, these are illustrated in Fig. 3.3. We start with basic verification, e.g. detecting references to undefined values or duplicate definitions. Control passes to the renamer, where definitions are renamed and replaced with unique identifiers to

[2]http://www.haskell.org/ghc/

remove scoping issues. Finally the verification stage infers and checks the types and units of all definitions within the model module, described in detail in Section 3.3. The output from these stages is a well-typed *Core* module accessible from the system model environment.

3.2.2 *Core* IR

The *Core* IR can been viewed as the core/primary modelling language, to which *Ode* acts as a user-friendly frontend to ease modeller development. The use of a well-defined intermediate modelling language allows development of multiple frontend languages for alternative specialised domains whilst running on a common modelling platform and benefiting from the same optimisations, presenting a *lingua-franca* for interoperable model definitions [42, 54]. Such an example is the high-level *Ion* markup DSL, for describing ion channel dynamics as presented in Chapter 6, this is currently implemented as a source-source transform into *Ode* but could target *Core* directly.

The *Core* language itself is a pure, strictly evaluated, variant of the simply-typed lambda calculus [7]. We base the implementation on the lambda-calculus partly to investigate the use of functional programming concepts to construct models and implement simulations. The lambda-calculus is a common formulation for programming languages with well-understood semantics for which known compilation techniques exist to generate optimised native code. The datatype used to represent a model within the *Core* language alongside the translation semantics from *Ode* to *Core* are provided in Appendix A.3.

We extend the core lambda-calculus terms with modelling specific ones, i.e. representing `init` values and `ode` constructs. This forms an immutable modelling IR we may use to express the numerical operations in a model constructed from ODEs. The `ode` construct may define the equation with respect to those calculations, i.e. $y' = f(t, y)$, such that we can calculate $y(t_{n+1})$ from $y(t_n)$ and perform a mutable update to the referenced `init` stateful value. Thus we strictly control access to state within the operational code that represents the model. This separation between the pure modelling core that only references the current state, and the mutable ODE constructs that may update it, eases code-generation and enables several optimisations (it is discussed further in Chapter 5).

3.3 Model Verification — Type and Units Checking

In this section we detail the creation of a type system for *Ode*. This provides strong, static type-checking of all expressions to create a well-typed model prior to execution, thus ensuring successful simulation. This also will provide compile-time based model feedback and aid optimisations that will facilitate high-performance simulation. The types of values may be inferred directly from the value definitions and explicit type annotations are not required.

The type system is extended to support units-of-measure in a novel and lightweight manner that allows for annotating values with physical units to ensure that all expressions are unit-checked within the model. This includes a programmable core units system with automated inference, derivation and conversion where possible, enabling minimal unit declarations when describing models of physical systems.

3.3.1 Type Checking

To help specify and validate models we have developed a static type system for *Ode*. This is applied to modules at the *Core* IR level and is based on the simply-typed lambda calculus [119]. The system is strongly typed with a fixed set of types, consisting of floats, boolean, unit/void, functions and tuples/records, mirroring the expressions outlined earlier in the chapter and described further in Appendix A.2.3.

To reduce complexity for modellers the types of expressions are fully inferred through usage in a model. Inference and checking occurs for all model expressions, including ODEs, the `time` term, etc. Modellers do not have to enter any type information as opposed to models created in commonly utilised modelling languages such as C++/Java. Models may be created as in dynamically/run-time typed languages such as MATLAB or Python (although with the numerous DSL features), yet retain all the benefits of strong, static/compile-time type-checking.

These benefits include capturing errors in model construction and verification of the correct composition and interaction of computable expressions during simulation (see Section 2.3.2). Types are used to ensure that a model may be compiled successfully and that simulation/evaluation will always succeed. They facilitate rapid modeller feedback during development and compilation, rather than at runtime, improving the model development process and leading to robust models.

Such comprehensive feedback regarding the types within a model eases refactoring and integration into larger models via typed interfaces.

Types act as a specification for an expression (including the units-of-measure within an expression), and can be used to generate static interfaces for expressions at function and module boundaries (as described in Section 4.3). Furthermore type information will aid the construction of efficient simulation code, enabling many low-level optimisations and the generation of simpler code by virtue of having performed all type-checking code prior to simulation.

3.3.1.1 Inference Rules

The types of expressions and modules are fully inferred and checked at compile-time, even across parameterised modules (discussed in Chapter 4). The type inference system is based on an implementation of the Damas-Hindley-Milner algorithm [119]. This algorithm creates unique type variables for terms whose possible values are constrained through usage in expressions; these constraints are unified to form a single concrete type for each term, if possible. The type constraint rules for *Core* IR expressions are given in Appendix A.4. As our type system is simplified to suit the domain such that annotations are not required, the only constraint needed defines equality between two types; for aggregate types this implies equality within the composite elements.

Having used the rules to generate type constraints on the model expressions, we attempt to unify them to obtain the final types. The algorithm iterates through the set of constraints, substituting type variables with equivalents and checking for equality according to each constraint. If all variables are successfully substituted and no type variables remain, we have a well-typed model with a single type for each expression. Otherwise, if the types mismatch or type variables remain, we do not consider the model to be well-typed and stop compilation to provide modeller feedback.

3.3.2 Units Checking

Equations and models of real-life physical processes often include units-of-measure, used to ensure that the equations and values are dimensionally consistent. As noted in Section 2.3.2, this is analogous to the concept of types within programming languages. We want to enable modellers to utilise units within their models, providing an additional form of model verification.

As such we extend the type systems to include a units-of-measure system that is lightweight as to not burden modellers yet programmable. This unit-checking system ensures that all expressions in a model are dimensionally consistent. The work is influenced by units-of-measure systems seen in functional programming languages as discussed in Section 2.3.2, and represents the application of ML-style units-inference to the biological modelling domain [78, 79, 131, 137]. However, in contrast the referenced approaches, we found that by restricting unit annotations to be monomorphic, we were able to implement full unit-checking and inference using logical unification [119] with several new constraint rules. We feel this limitation is acceptable within the intended domain of mathematical biological modelling.

We extend the units system to support programmable units creation and auto-derived unit conversion functions. Furthermore we introduce unit-checked modelling functions and mathematical operators. This is particularly useful when modelling biological systems, allowing the modeller to further specify the model requirements in relation to observed experimental data. The system is comprised of two components, static unit checking/inference and automated unit casting/conversion.

3.3.2.1 Overview — Syntax & Semantics

The units-of-measure system is user-defined and programmable, consisting of quantity, unit and conversion definitions. These are placed within modules that are accessible globally when imported during model construction. Below we detail the constructs that form the units system. To increase clarity they are illustrated with code fragments that demonstrate the system using basic, everyday units rather than those used within biological modelling.

Quantities and Dimensions

```
1 quantity Mass = Dim M
2 quantity Velocity = Dim L.T^-1
```

Quantity declarations define a particular dimension with a name; they can define both base quantities and higher-order derived quantities. A quantity is comprised from a vector of the seven SI-defined dimensions $(L, M, T, I, \Theta, N, J)$ [139], where each dimension is represented by an integer index that indicates its order. The above sample creates a base quantity `Mass` of a single

dimension, alongside a derived quantity `Velocity` of several dimensions.

Units

```
unit m { dim : L, SI : true }
unit s { dim : T, SI : true }
```

Base units form the building blocks of SI units from which all other units-of-measure may be derived. A base unit defines a unit within a single member of the 7 SI dimensions. There is a universally recognised base unit for each member of the SI dimensions. In the above example we define `m`, *metres*, as a base unit for the quantity *length* with the dimension *L*.

A base unit declaration also accepts an optional boolean `SI` parameter; this declares the base unit to be an SI unit (as opposed to non-SI units such as *feet*). If true, the implementation will automatically define metric-prefixes for the base unit, e.g. `mm` and `km` for `m` above, in addition to automatically deriving conversion expressions between said units. This enables the rapid creation of entire families of units.

```
derived m_per_s { unit : m.s^-1 }
derived V { unit : kg.m^2.A^-1.s^3 }
```

Derived units are formed by powers, products or quotients of the base units and are unlimited in number. They are in turn associated with derived quantities previously described. For example the quantity *velocity* is derived from the base quantities of *time* and *distance*.

In the above code demonstrates such a derived velocity unit, `m_per_s` using the base units metres per second $m.s^{-1}$. As seen, a derived unit has a name and its dimensions may be expressed in terms of base units through the `unit` element. We demonstrate an encoding of *voltage*, as used in electrophysiological models, in terms of its base units. Within the implementation, derived units represent a syntactic alias for their constituent base units and are expanded when encountered.

Conversion expressions

```
conversion { from : m, to : ft, factor : x * 3.2808 }
```

We introduce conversion expressions to convert automatically between user-defined units within the programmable units system. These utilise a sub-DSL within the main DSL for expressing the particular conversion function between two base units of the same dimension. The sample above defines a conversion construct between m and ft that may be achieved via the given expression.

The `conversion` construct contains elements defining the `from` and `to` base units to convert between, and a `factor` element that defines the conversion expression. The grammar for the conversion expression language is provided in Fig. 3.4. It encodes basic numerical operations, $(*, /, +, -)$, literal numerical values, and an identifier x that is bound to the `from` unit.

This allows modellers to programmatically express conversions between many types of units, with the expressions powerful enough to cover the SI-system and units that are not zero-based, e.g. Fahrenheit[3]. A conversion construct defines a one-way mapping; a pair of conversion constructs is required to create a bi-directional mapping between two compatible base units, each with inverse `factor` functions.

Assignment and Casting/Conversion

```
1 val x = 3 { unit : m }
2 val y = x { unit : ft }
```

The unit declarations outlined in the previous sections can be used to build a custom units subsystem that supports conversion between compatible derived units. We now consider how units may be utilised in our code to construct unit-checked expressions and models.

Firstly we may annotate numbers with a unit-of-measure attribute within model computational code. This can be achieved simply annotating a literal number with an optional `unit` element that may be a base or derived unit. Line 1 in the above sample demonstrates this, creating a value `x` of unit m. The dimensions of the unit are stored within the *float* type, and the absence of such a unit annotation implies a dimensionless value.

Unit-annotated numerical values may be cast and converted to specific units using the very same unit attribute. This attribute may be applied to unit-aware expressions where it acts as a joint casting and conversion operator. This is demonstrated in Line 2 above, where `y` is created

[3]Logarithmic units such as *bels* are not supported.

from casting `x` from m to ft. The validity of the cast is determined during unit-checking and may not be applied to dimensionless expressions.

In many cases such manual casting will not be required due to the unit-inference system described in Section 3.3.2.3. However, on occasion, inference can lead to ambiguous behaviour, and manual casting is required to justify the modeller's intent. As an example, the unit operation $\frac{m^2}{km}$ would result in a unit of $m^2.km^{-1}$; this would have to be manually cast to m or km when used elsewhere (or any other unit of dimension L).

Casting and conversion requires that the existing unit and the resulting unit are compatible, i.e. of the same dimension, and that it is possible to derive a valid conversion expression from the collection of previously defined base-unit conversions. The derivation algorithm is described in the Section 3.3.2.4.

Built-in operators

We extend the built-in operators to include several unit-aware functions and update the types of existing mathematical functions to ensure that they are unit-aware. In most cases this requires ensuring that the inputs and outputs to functions are dimensionless (see Appendix A.2.2).

```
1 val x = upow(2 { unit : m }, 2) // x is in m^2
2 val y = uroot(x, 2)             // y is in m
```

An important addition is the introduction of unit-aware power and root functions, termed `upow` and `uroot` receptively. Their use is demonstrated in the above example to square and take the root of a unit-aware value. As seen, they accept a unit-annotated numerical expression and a positive integer representing the value of the power/root, returning the calculated expression with correct unit. They are required due to the previously mentioned monomorphic restriction on unit values. The standard power and root functions operate on dimensionless values only, and thus we introduce these intrinsic unit-aware functions that specifically support polymorphism on the input parameters.

The `time` term that represents the current time t_n during a simulation is annotated with a unit of dimension *Time*. The chosen base unit can be configured from the console prior to running a simulation from, *seconds* (with metric prefixes), *minutes* and *hours*. The system provides automated conversion functions for these units, allowing models to utilise values with time

$$
\begin{aligned}
\mathit{odeStmt} &\leftarrow \mathit{modStmt} \mid \mathit{quantityDef} \mid \mathit{unitDef} \mid \mathit{convDef} \mid \mathit{exprStmt} \\
\\
\mathit{quantityDef} &\leftarrow \texttt{quantity}\ \mathit{id} = \mathit{dim}\ \mathit{num}_1\ \mathit{num}_2\ \mathit{num}_3\ \mathit{num}_4\ \mathit{num}_5\ \mathit{num}_6\ \mathit{num}_7 \\
\mathit{unitDef} &\leftarrow \texttt{unit}\ \mathit{id}\ \{\texttt{dim} : (\texttt{L} \mid \texttt{M} \mid \texttt{T} \mid \texttt{I} \mid \texttt{O} \mid \texttt{J} \mid \texttt{N}), [\texttt{alias} : \mathit{id}], [\texttt{SI} : \mathit{bool}]\} \\
&\quad \mid \texttt{derived}\ \mathit{id}\ \mathit{unitCast} \\
\mathit{unitCast} &\leftarrow \{\texttt{unit} : (\mathit{id}_1\hat{}\mathit{num}_1).\cdots.(\mathit{id}_n\hat{}\mathit{num}_n)\} \\
\\
\mathit{convDef} &\leftarrow \texttt{conversion}\ \{\texttt{from} : \mathit{id}, \texttt{to} : \mathit{id}, \texttt{factor} : \mathit{convExpr}\} \\
\mathit{convExpr} &\leftarrow t_1 * t_2 \mid t_1 / t_2 \mid t_1 + t_2 \mid t_1 - t_2 \mid \mathit{convTerm} \\
\mathit{convTerm} &\leftarrow (E) \mid \mathit{num} \mid x
\end{aligned}
$$

Figure 3.4: *Ode* language grammar for the units-of-measure subsystem DSL using a variant of EBNF as in Fig. 3.2.

specified in one unit whilst the simulation proceeds under another, or even submodels to be composed with differing time unit constraints.

Finally the `ode` construct is modified in conjunction with the `time` term to provide unit-checked ODE definitions. Considering a system in standard form, $y' = f(y,t)$, it ensures that t has the same unit as the configured `time` term and that y', the delta expression within the `ode` construct, is also defined in terms of that same unit.

These operator extensions demonstrate the implementation of unit-checked modelling at the expression levels, including the ODEs that comprise the model. It provides a rigorous mechanism for constructing unit-aware models, and in conjunction with casting and automated conversions is novel within mathematical modelling languages. It will be particularly useful when used with the module system to construct models from multiple sources by implementers that obey differing unit-conventions.

3.3.2.2 Grammar

Fig. 3.4 presents the grammar for the units-system language in *Ode* using EBNF. Units-annotated examples of the HH52 and BR77 models demonstrating this syntax are presented in Appendix D.4.

3.3.2.3 Inference Rules

As per the type-system, the programmable units system is partially-inferred utilising several inference rules for numerical expressions. These rules make use of a constraint system similar to

the type inference rules but with several types of constraints possible between units.

As with the type equality constraint, a unit equality constraint ensures that input and output units are equal (or not present, i.e. dimensionless), when performing addition or subtraction operations. Another constraint ensures that the output unit is derived from the addition or subtraction of the input units, thus used to check multiplication and division operations. Additional constraints are used to check power operations and to ensure that units are of the same dimension when performing unit-casting. The unit inference rules and constraints are listed in Appendix A.5.

Units-annotated versions of the HH52 model and a modular version of the BR77 model are presented in Appendix D.4. It may be noticed that several expressions in these example models contain many unit-annotations. This is due to the strict unit-checking we have implemented. As discussed, we explicitly disallow ambiguous unit-aware operations to take place, e.g. adding a unit-aware and dimensionless value, or using dimensioned values within transcendental functions. This requirement could be relaxed in the future by altering the unit constraint rules.

3.3.2.4 Implementation Details

Unit checking and inference is performed as part of the type-checking process, the type constraint rules are extended with the units constraint rules. These constraints are checked and solved during the unification phase of type-checking. We unify the units variable generated by the inference rules with a similar unification algorithm to that used for type constraints. Unification over the set of unit-constraint types occurs until a fixed-point emerges; at which point if unit-variables remain we declare unit-checking to have failed (potentially due to a lack of annotations within the model code), otherwise the model is said to be unit-checked.

Units in *Ode* are created as an instance of a particular dimension vector. Values may then be explicitly converted between units of the same dimension vector only if a safe conversion factor may be found within the global system scope.

To satisfy a specified unit conversion we generate a graph of all available conversion rules for a particular dimension. The conversion rules form a directed graph between base units nodes for each dimension, with conversion functions as the edges between them. Fig. 3.5 illustrates a graph that would be created implicitly by the units commands, in this case it provides a sample system for converting between units of type L. To obtain a compatible conversion function between base units we find the shortest directed path between the *from* and *to* unit. If this exists, the system

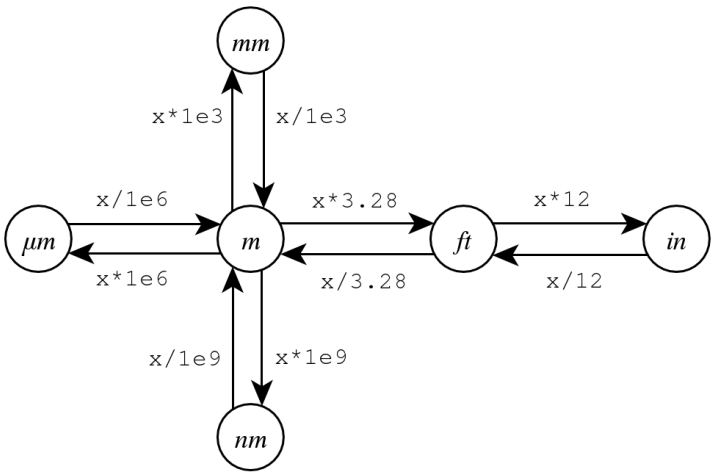


Figure 3.5: A simple example graph as would be generated by the programmable units systems to derive the expression needed to convert between units of the same dimension. Using this graph for the dimension *L*, converting a value from *nm* to *in* would return a conversion expression of `(((x/1e9)*3.28)*12)`, this would be optimised during compilation to `(x*3.9e-8)`.

will chain and inline the multiple conversion functions along the path to derive a valid conversion expression[4].

For complex derived units, i.e. those whose dimension vector has a length greater than 1, we consider each dimension independently and combine the resulting conversions. This process is described using pseudo-code in Algorithm 1 — it involves rearranging the *from* and *to* units symbolically for each set of units to be converted in each dimension, modifying and cancelling compatible units on each side of the conversion. Having determined the final units for conversion, we search the conversion graph to derive a valid conversion function that may be applied to obtain the required unit.

All derived conversion expressions are translated from the conversion sub-DSL into low-level *Core* code during compilation. This occurs within the code-generation stage, where the model code is automatically rewritten to transparently apply conversion instructions. Such inlining is amenable to static optimisations and will result in minimal run-time performance loss for unit-casting/conversion. The process is described further in Section 5.1.3.

This section has demonstrated how we have achieved transparent, programmable, unit conversions and casting at a high level within the model code with only slight performance loss, as is demonstrated in Chapter 5.

[4]Finding the shortest path has the benefit of reducing accumulated FP-error of the resulting inlined expression.

Algorithm 1 Pseudo-code describing the algorithm used to derive unit conversions from a unit termed $fromUnit$ to one termed $toUnit$. The algorithm uses the dimension graphs (see Fig. 3.5) to convert between the set of base units specified by $fromBaseUnits$ and $toBaseUnits$ within the units in each dimension. The algorithm returns a single expression $convExpr$ that encodes the conversion.

Require: access to unit dimension graphs
  function ConvertUnit(fromUnit, toUnit)
    $convExpr \leftarrow x$ ▷ initialise conversion expressions with x (i.e. id)
    for each dimension dim **do**
      $fromBaseUnits \leftarrow \{x \in fromUnit \mid x$ base unit in $dim\}$
      $toBaseUnits \leftarrow \{x \in toUnit \mid x$ base unit in $dim\}$

      ▷ remove base units common to both sets
      $commonBaseUnits \leftarrow fromBaseUnits \cap toBaseUnits$
      $fromBaseUnits \leftarrow fromBaseUnits - commonBaseUnits$
      $toBaseUnits \leftarrow toBaseUnits - commonBaseUnits$

      ▷ iterate over base units, deriving and inlining conversion expression
      while $(fromBaseUnits \neq \emptyset) \wedge (toBaseUnits \neq \emptyset)$ **do**
        $fromBaseUnit \leftarrow$ select and remove any base unit from $fromBaseUnits$
        $toBaseUnit \leftarrow$ select and remove any base unit from $toBaseUnits$
        $baseConvExpr \leftarrow$ using conversion graph for dim derive expression between $fromBaseUnit$ and $toBaseUnit$
        error if path not found between base units in graph
        $convExpr \leftarrow$ inline $baseConvExpr$ into $convExpr$
      end while
      error if $(fromBaseUnits \neq \emptyset) \vee (toBaseUnits \neq \emptyset)$
    end for

    return $convExpr$
  end function

3.4 Simulation Execution

We describe the mechanism for loading and configuring models and performing simulation through the system interface that provides many benefits to modellers and model development. A DSL user interface (UI) is important to enable productive use by domain experts and the efficient integration of the tool within a modeller's existing workflow. It is a more qualitative aspect of the research, concerned with tooling, error and debugging support, development environments, batch operation support, output formats and analysis. The *Ode* DSL itself is described using a textual syntax, as is the declarative *Ion* DSL in Section 6.2. However a GUI frontend may be developed, for example similar to LabView[5], to increase usage by modellers unfamiliar with computer programming.

Having verified a model we would like to run simulations according to model- and experiment-specific requirements and parameters. A mechanism is required to enable users to configure the simulation parameters, compile the model, and run the simulation over the requested time interval to simulate the model with the chosen numerical solver.

Finally we need to store the approximate, generated solution to enable post-simulation, off-line analysis of results. To achieve this the system exports the current model state, represented by the set of state values $y(t_n)$, at regular intervals t_i during the simulation into an efficient custom binary format described in Appendix A.6. A small analysis framework has been developed in Python that supports the output file-format, this will be extended to include features for cardiac model analysis.

3.4.1 Simulation Configuration

A model in *Ode* is independent from the parameters used to describe a specific simulation instance, e.g. the ODE solver, start time, timestep, etc., with the *Ode* DSL used solely for describing the model under investigation. This enables sharing of model code that is unencumbered with simulation parameters, paving the way for repeatable, verifiable simulation results and helping to facilitate functional model curation [32].

The system provides a textual interactive console environment (also termed a top-level or read-eval-print-loop (REPL)) that forms a UI, independent from the main modelling DSL, from

[5] `http://www.ni.com/labview/`

Listing 3.2 A example of the console syntax used to configure a simulation, setting the simulation time parameters and output filename.

```
import HH52Model
:startTime 0
:stopTime 10
:timestep 1
:output "results.bin"
:simulate HH52Model
```

which modellers can load, configure and execute simulations in an interactive and batch manner. This will enable modellers to produce reliable, repeatable simulations and by virtue of the just-in-time (JIT) compilation mechanism described in Chapter 5 provides opportunities for modellers to rapidly (re-)configure models and simulations to perform investigations into specific model characteristics.

3.4.1.1 Syntax

The console has a simple command-driven interface to configure many internal implementation and simulation options. Essentially it is another, high-level, DSL for configuring the system with a basic command syntax of the form,

```
:command <option>
```

where `<option>` may be an optional integer, float, string or identifier as required. Listing 3.2 illustrates the use of these console commands to configure the simulation parameters such as t_{start} and t_{stop} and the timestep h; set output parameters such as the file name and output interval t_i; and finally start the simulation of a chosen model. The full list of console configuration commands are provided in Appendix A.1 and its use is demonstrated in the tutorial in Appendix D.1.

3.4.1.2 Usage and Scripting

The console interface represents the UI to the *Ode* system and has three major modes of operation. By default the system operates in an interactive mode and commands are manually entered one after the other. To aid modellers this environment supports tab-completion, interactive help and has a history of previously used commands.

If a file-path is passed on the command-line upon invocation the system enters a batch-processing mode. In this mode the contents of the referenced file are interpreted as a list of

pre-defined console commands. This allows the static creation of commands that may be distributed with a model to enable repeatable simulations and results by independent modellers and to facilitate model curation.

The final, and in our opinion, most powerful mode, uses the interactive environment as a scripting-destination, enabling run-time programmable usage and control. This can be enabled by initialising *Ode* with a named file-system *pipe* or by utilising UNIX input redirection. When used in such a manner, the system effectively becomes a library accessed within a larger program whose API is defined by console commands to control and construct simulations. This enables controlling *Ode* from external tools, such as Bash scripts or Python, to construct more-complex, programmable, simulation scenarios.

The configuration UI and operating modes provide many benefits and modelling use-cases. For instance modellers may batch large numbers of related simulations together, computationally generating each specific configuration. This is useful when parameterising a model, performing multi-dimensional parameter sweeps, configuring parameter estimation simulations, in a repeatable and controlled manner. Separating the simulation configuration from the model enables testing between models using the same configuration and could be extended further to automatically compare against experimental data to provide full functional curation (see Section 2.3.4.1). Finally, a model and its subcomponents may be loaded and initialised with parameters from the console at run-time utilising the module system defined in Chapter 4. The console language could be extended in the future to provide additional features, such as controlling parameter sweeps and automatic generation of sensitivity matrices. For instance, we have developed a Python-based tool to create parameters sweeps easily that are then exported into code-generated *Ode* modules. These modified models are then compiled and simulated automatically from the tool using the console interface.

3.5 Discussion

This chapter presented the research and design involved in the creation of the *Ode* DSL, including the basic syntax and modelling constructs. We introduced features to perform general numerical computation and to structure biological systems in a flexible, compartmentalised manner that may be modelled in a continuous deterministic fashion using ODEs. Further language details and working models are provided in Appendices A and D.

Several implementation details for the front- and middle-stages were discussed. These operate on the *Core* internal model representation that is extended throughout this thesis to provide the requisite modelling and simulation features.

We described mechanisms for verifying *Ode* models, including a type-system and a units-of-measure system for ensuring the correctness of expressions and the physical values they represent. Types are used to ensure the expressions within the model are valid, to provide rapid feedback to the modeller during development, and to enable high-performance simulation. This was extended to provide a lightweight, programmable, units-of-measure system that enables creating and assigning units to values and expressions. Unit-aware expressions are automatically checked to ensure a dimensionally-consistent model, with a novel implementation providing conversions and derivation as required. The system includes the unit-checked calculation of ODEs and higher-level mathematical operations such as powers and roots. Both type checking and units-of-measure checking are lightweight verification techniques that we have applied to biological modelling, used to ensure that the model is constructed correctly and can be compiled. *Ode* implements further syntactic and semantic checks to aid model code verification. Units-checking may also be used as validation tool to check the dimensional consistency of a verified model implementation.

We described how the integrative system enables both model development and simulation configuration in a robust manner that encourages repeatable results, vital within the modelling community. This is provided by a console interface to interact with the system and configure simulations. This may be used interactively, in a batch mode, and finally in a scripting manner that enables programmatic generation/configuration of models and simulations externally.

The *Ode* DSL described in this chapter represents the front-end used by modellers to develop their models. The creation of a usable syntax with sensible semantics is vital to allow modellers

easily to create, develop, modify and simulate their models and in effect acts as the UI for modelling life-cycle. It is a vital first step in creating tools and domain models we can use to apply concepts from software engineering to this domain. Whilst it is difficult to determine quantitatively the ease-of-use of a language we believe basing the syntax on widely-used (and recommended for their simplicity) high-level systems such as MATLAB and Python provides a good starting point. The following chapter extends the base *Ode* DSL to include a module system for encapsulating model components logically within replaceable, reusable units. In Chapter 5 we introduces the compilation system for *Ode* to generate an efficient simulation of an *Ode* model and we again extend the DSL to include stochastic modelling support in Chapter 6.

Chapter 4

Model Composition & Reuse

Models are often programmed as static, monolithic, interdependent blocks of code, however many similarities exist within and between them — for instance a model of a potassium channel could be shared and reused between cardiac models [105, 107]. It would be highly beneficial to split models into separate, well-defined building blocks that may be combined and reused to create more complex systems, as often occurs in software engineering. This facilitates code reuse and sharing between modellers, leading to a separation of concerns that enables higher levels of abstraction when developing models. This would enable a collaborative development process whereby model components may be developed and composed by modellers working across domains, similar to the usage patterns observed in open-source software. Such modularity is a key component of modern programming languages and is necessary to build flexible, reusable models from defined components. This could be used, for instance, to reproduce and modify published models. The ability to import previously defined components in some form is provided in several modelling languages, including CellML 1.1 [93, 141]. However CellML provides only a basic import feature that simply loads a previously defined model into the current namespace. This enables the static composition of models, whereby each parameter shared between model components must be linked together. As described in Section 2.3.1.2, this is a manual and time-consuming process, resulting from the limited modelling abstractions available in the XML-based DSL [35].

Instead we are interested in the programmable generic construction of parameterisable modules. We believe this will allow for the use of design patterns seen in modular software development within this domain. Typed, interface-based modules will enable the independent development and collaborative construction of model components. Furthermore, it will facilitate

the reuse and reconfiguring of such components to generate custom models for *in silico* experimentation. Such work could not be performed on top of CellML without drastically altering the core CellML approach. CellML does not have a type-system to support typed module interfaces or to type-check module composition. Nor do its modelling abstractions provide suitable power to express the modular design patterns described in Section 4.1. This is due to the differing modelling approaches taken, where CellML investigates the fixed definition of model equations with a focus on curation; and *Ode* investigates a programmable, operational approach influenced by (functional) programming languages.

Firstly we examine how modules may be used to develop reusable, component based models that mimic biological structures whilst utilising techniques common to software engineering when designing large-scale systems. This includes concepts such as abstraction, encapsulation, and generics; alongside compile-time polymorphism and aggregation, to design and structure complex models within a modular framework.

This knowledge is used to influence the design of a typed module system for *Ode* to structure biological models into reusable components that may be shared and reused between models; the syntax and implementation details are discussed. Modules in *Ode* consist of a set of definitions that are publicly exported through a typed signature. They may import other modules and thus act as a barrier for defining independent code, with well controlled access mechanisms.

We would like to have the ability to modify and specialise imported modules for particular scenarios, for instance to reuse model components with differing constant parameters. As such the module system is extended to support parameterised modules, influenced by OCaml functors [101, 104, 132]. This introduces a form of interface-based model construction that enables the specialisation of reusable modules. Thereby facilitating modelling through the modification of parameters and equations used within a module. Parameterised modules allow modules to take other modules as parameters, allowing common behaviour to be described with the particularities abstracted away. They provide great flexibility and abstraction, allowing compile-time configuration of investigations into parameter variations, including batch simulations and sensitivity analysis.

The module system represents a flexible, type-checked means with which to reuse model components. We discuss several aspects of the module system, including the type system used to

specify a module's requirements and enable safe module composition. We also address the use of module repositories used to structure models into reusable libraries that can be collaboratively developed and distributed. Finally we discuss use of the programming language formed by the module system, and evaluated during model compilation, to implement several powerful abstractions.

We demonstrate the utility of the module system using examples from a modular form of the Hodgkin-Huxley (HH52) model [69] listed in Appendix D.4.1. To clarify visually the design of modular *Ode* models we utilise UML class diagrams, using a notation described in Appendix D.2. Finally, we perform a case study into the use of parameterised modules to abstract a particular ion channel, I_{Na}, the sodium channel, within a cardiac model through a series of cardiac models with a shared history.

4.1 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, biologists, mathematicians, and computer scientists. Models themselves are derived from data collected from various studies and are developed in an iterative fashion, with newer, generally more accurate, models based on existing ones [105].

As such a need exists to 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 several features that allow such collaborative modelling in a structured and safe manner.

A flexible, typed-checked module system has been created that enables the modification and reuse of existing models. In this section we summarise several techniques made possible by the module system to structure and develop large-scale reusable biological mathematical models, inspired by methods and patterns used in software engineering [12, 129]. This forms an initial set of patterns for structuring cardiac models that we utilise and extend further within the simulation study in Section 4.4.

Modular programming involves describing a system in terms of encapsulated modules and the interactions between them through well-defined interfaces. This type of interface-based programming leads to a separation of concerns and enables collaborative modelling by users focussed on differing areas. 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 [12, 129] (see Section 2.3).

Extensibility is a property of systems design that is a measure of both the degree to which a system can introduce new functionality and the effort required, 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 invaluable to encouraging the longevity of software.

We look at how these desirable aspects can be obtained through model structure using the DSL features presented in Section 4.2; demonstrating their use in a simulation study within Section 4.4. When applied appropriately, they can be used to enable greater code reuse and extensibility when structuring complex biological models. The focus of this research has been to apply software engineering techniques that allow models developed in our DSL to be highly reusable and extensible, facilitating rapid composition to investigate certain parameters of interest and allow segments of models to be interchanged.

4.1.1 Encapsulation

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, is encapsulation. We can view modules as being 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, providing a high-level interface to the module functionality. We would also like to hide and encapsulate the internal implementation of data structures through the type system, although this was not fully completed during our research and is discussed further in Section 7.2. 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. Explicit export definitions within module declarations facilitate implementation hiding and help to achieve this goal. Fig. 4.1 provides an example of how we may encapsulate several cell-level parameters from the HH52 model into a module, the diagram uses a notation described in Appendix D.2.

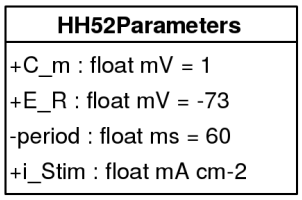


Figure 4.1: UML class diagram indicating how several related model elements may be grouped and encapsulated within a module. Here the original cell and stimulus parameters for the HH52 model are encapsulated into a *parameters* module.

4.1.2 Generics

Generics, provided via parameterised modules, enable modellers to operate on encapsulated objects in an abstracted manner, enabling polymorphic reuse and substitutability, as commonly seen in C++ templates. It is a style of programming in which expressions 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.

4.1.3 Subtyping

Our implementation provides structural subtyping 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.

This can be used in a manner similar to class hierarchies in object-orientated (OO) languages, that is, as a mechanism by which to prescribe appropriate relationships between classes/modules in a model system. 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 and utilised, providing low-coupling between modules and allowing rapid development and substitution of alternate implementations. This is in contrast to *nominative* subtyping, where each class/module must explicitly *inherit* from a named existing parent to be considered a sub-type, leading to rigid hierarchies [12].

If used properly, subtyping, coupled with generics, can improve the understandability of model

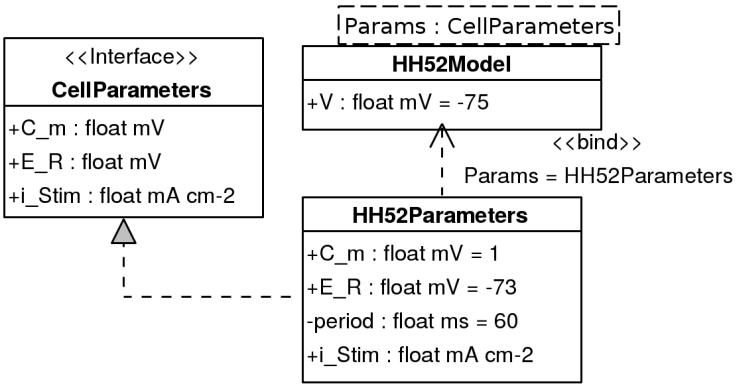


Figure 4.2: Class diagram indicating how a model, `HH52Model`, accepts a generic module parameter. The binding relationship indicates assignment of a module to a functor argument, generating a new module from the *template*. The assigned module must implement, or be a subtype, of the `CellParameters` interface, e.g. `HH52Parameters`.

code by helping to minimise the conceptual distance between code and the real-world system which the code models. For instance, Fig. 4.2 presents a use of generics to define an abstracted interface for representing the model parameters in `CellParameters` that is implemented and applied by the subtype `HH52Parameters`.

4.1.4 Aggregation & Composition

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-style inheritance, achieving reuse statically during compilation rather than at run-time. Delegating to multiple modules provides reuse in a manner reminiscent to multiple inheritance in OO languages [12, 51].

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 strongly-typed substitution whilst avoiding deep inheritance hierarchies often seen in OO-languages; composition and aggregation are often seen

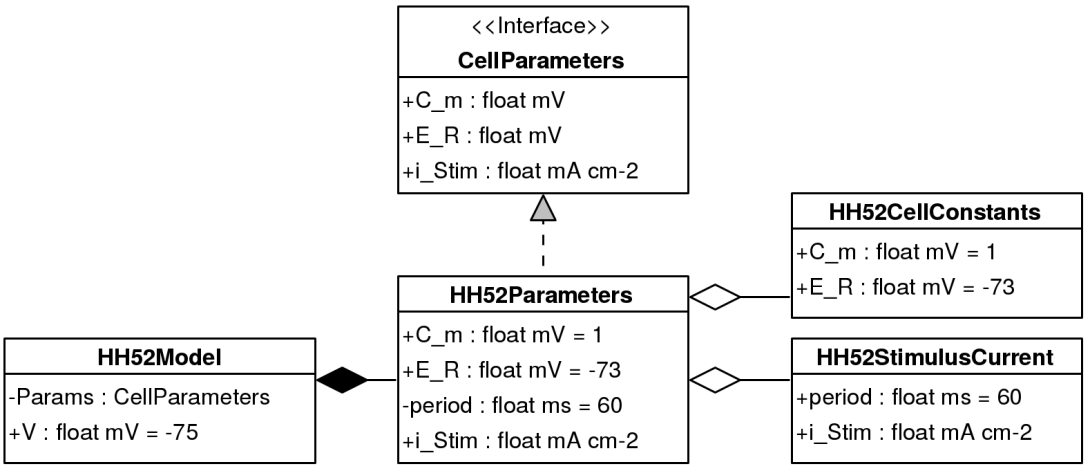


Figure 4.3: Class diagram demonstrating use of containment and aggregation to provide code structure and reuse. We compose the parameters module with the main model, whilst specifying that it is a subtype of `CellParameters`. The `HH52Parameters` module implements this interface via the aggregation and reuse of sub-modules.

as a more flexible option in place of inheritance [51].

When parameterised modules are utilised with aggregation they allow generic code reuse, abstracting the particular instances until simulation time. This provides compile-time substitutability, in contrast to the run-time substitutability provided by OO-style inheritance, and a powerful mechanism for rapid model construction and modification. This may be used to facilitate dependency injection and implement *mocking* and *proxying* objects [74], commonly seen in agile software development and testing. Fig. 4.3 presents an example of how both composition and aggregation may be used to structure and reuse code for use with the HH52 parameters module.

The following section presents the DSL module system in detail, describing how modular models may be developed from the code perspective. These architectural features, patterns, and structural techniques are later put to practice in the case study in Section 4.4 that investigates the initial construction of a module repository comprised from reusable, substitutable, cardiac model components.

4.2 Module System — Syntax and Semantics

In this section we describe the *Ode* module system used to structure our models into modules consisting of collections of related value and component definitions that form some logical (and potentially biological) grouping. We describe their syntax and semantics, including constructs for declaring modules, and how to reference them externally. We discuss parameterised modules, enabling compile-time substitution of components and generic, interface-driven development that provides further opportunities for abstraction and model reuse. These are illustrated with fragments of *Ode* model code and are textual representations of several UML diagrams used in the previous section to depict the design of a modular HH52 cell model.

The module system was inspired by Modula-3 [18] and OCaml [90, 104], where in the second case it is effectively a higher-level language that enables the programmable creation of independent modules [89]. The *Ode* module system operates similarly; providing variable definitions, references, function/functor abstraction and application, all at the typed module level. This provides extensive power to reconfigure models for particular tasks during compilation.

4.2.1 Module Definition

```
module HH52Parameters {
  val E_R = -75 { unit : mV }
  val Cm = 1 { unit : mF.cm^-2 }
  val period = 60 { unit : ms }
  val i_Stim = piecewise {
      time % period >= 10 and time % period <= 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 or a user-defined relationship. For instance, the code fragment above introduces the syntax for creating a module that encapsulates several cell-level parameters for use in the HH52 model, and represents the textual equivalent of the UML class diagram provided in Fig. 4.1. A module declaration uses the `module` keyword followed by the module name and the module body contained within braces.

Several modules may be defined within a single file, where the directory path in which the files are placed forms 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, as demonstrated in Section 4.4.

4.2.2 Importing & Using Modules

```
module HH52Model {
  import CardiacElec.HH52.HH52Parameters as Params
  // leakage current
  val E_L = Params.E_R + 10.613 { unit : mV }
  val i_L = 0.3 { unit : mS.cm^-2 } * (V - E_L)
  // ... other channel models ...
  init V = -75 { unit : mV }
  ode { init : V } = -(-Params.i_Stim + i_Na + i_K + i_L) / Params.Cm
}
```

A module may import another module through an import declaration in order to references its expressions. This is demonstrated in the above code sample, illustrating the composition of a module representing several cell-level parameters from the HH52 model (as was described textually in the previous subsection and illustrated visually in Fig. 4.1) with the HH52 cell model module. 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 cell model imports a module `HH52Parameters` from a file called `HH52` within the `CardiacElec` directory of an available and loaded module repository. An optional `as` construct may be used to provide an alias for the module.

When processing an import declaration, the module is retrieved according to the available repositories and loaded into the global module environment, this occurs once per module reference with the system performing checks to avoid circular dependencies. The visible definitions of an imported module are again accessed using a dot-notation on the qualified module name, demonstrated by references to `Params` above. They are used in the same manner as any local expression.

All modules within a file can be imported into the current module by utilising a `*` wildcard in place of the module name, as in the following sample. Such modules are qualified by their

module name to avoid namespace clashes. The example below demonstrates this, loading all modules related to the HH52 model into the base cell model,

```
module HH52Model {
  import CardiacElec.HH52.*
  val E_L = HH52Parameters.E_R + 10.613 { unit : mV }
}
```

Modules must be explicitly imported for use within a module, however modules may also be configured and operated at the file and console levels with differing semantics. This is a result of the programmable nature of the module system. We may import modules at the file-level, using the same syntax, to provide aliases for modules defined elsewhere within the current module hierarchy. The console environment (see Section 3.4.1) is extended with the same import syntax. Modules may be loaded into the interactive environment, configured and constructed to represent a simulation-ready model. This presents a programmable system for simulating models that may be scripted and called via external processes.

4.2.3 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, the exported and required interfaces are matched and checked by the module type-system, as described in Section 4.3.1, to ensure successful composition.

By default a module's exported interface is comprised from all top-level definitions. However a modeller may configure the visibility of certain definitions and thus modify the interface. This may be achieved by structuring the model code such that only exportable definitions are visible at the top-level, or by explicitly declaring the visible definitions to export (similar to *public/private* access modifiers in OO languages). This can be achieved through the use of the `export` notation when defining a module. The following example demonstrate its use to modify the parameters module defined in Section 4.2.1 such that it more closely resembles the encapsulated module in Fig. 4.1,

```
module HH52Parameters {
  export (E_R, Cm, i_Stim)
```

```
3     val E_R = -75 { unit : mV }
4     val Cm = 1 { unit : mF.cm^-2 }
5     val period = 60 { unit : ms }
6     val i_Stim = piecewise {
7         time % period >= 10 and time % period <= 10.5 : 20 { unit : mA.cm^-2 },
8         default: 0 { unit : mA.cm^-2 }
9     }
10 }
```

As demonstrated, the command accepts a comma-separated list of exportable definitions. It enables information hiding and encapsulation of a model's internal data, as discussed in Section 4.1.1. In this manner only the vital information is exported in the module interface, mimicking biological mechanisms of compartmentalisation and containment.

4.2.4 Parameterised Modules

Utilising modules to group related definitions enables modularity and code reuse, however the imported modules and the abstractions within them are concrete and 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 a suite of protocols to compare simulation runs against experimental data [32].

In several languages *parameterised* modules, also termed *functors*, present a flexible and powerful means for accomplishing this. They are implemented as a compile-time construct that may be used to create typed, generic components in a similar manner to OCaml functors [104] and C++ templates [43], allowing strongly-typed substitutability of components with differing implementation details.

Parameterised modules enable the substitutability of compatible components, yet may be specialised through type signatures/interfaces that a generic component must implement. Modules may be parameterised by other modules leading to the creation of complex, specialised modules via a form of aggregation. This flexibility and abstraction allows for rapid investigations into the effects of parameter variations, for instance when performing sensitivity analysis, from a single code-base — techniques for developing such reusable models were provided in Section 4.1.

4.2.5 Parameterised Module Definition

```
module HH52Model(Params) {
  // leakage current
  val E_L = Params.E_R + 10.613 { unit : mV }
  val i_L = 0.3 { unit : mS.cm^-2 } * (V - E_L)
  // ... other channel models ...
  init V = -75 { unit : mV }
  ode { init : V } = -(-Params.i_Stim + i_Na + i_K + i_L) / Params.Cm
}
```

The above example demonstrates the use of parameterised modules within *Ode*, where the specific parameters module utilised by the HH52 cell model in Section 4.2.1 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 that operate over modules, they take a set of input module arguments and return a new, transformed module derived from the input module definitions.

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. Appendix D.4.2 extends our cardiac module example to utilise parameterised modules, demonstrating their use in structuring large-scale models that can be developed in a generic manner, and (re-)used and modified at compile-time as needed.

4.2.6 Parameterised Module Application

```
module HH52Original = HH52Model(HH52Parameters)
```

The above code fragment demonstrates applying a concrete version of the HH52 parameters module to the base cell module, resulting in a complete version of the HH52 model, as was illustrated in Fig. 4.2. As seen above, functor application uses a syntax similar to function application, the end result is similar to template binding in C++. 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 within the system.

A current implementation limitation is that parameterised modules must be fully applied and

$$
\begin{aligned}
\mathit{moduleStmt} &\leftarrow \mathit{moduleDef} \mid \mathit{importStmt} \mid \mathit{applyModule} \\
\\
\mathit{moduleDef} &\leftarrow \texttt{module}\ \mathit{id}\ \mathit{moduleBody} \mid \texttt{module}\ \mathit{id}\ (\mathit{id}_1, \ldots, \mathit{id}_n)\ \mathit{moduleBody} \\
\mathit{moduleBody} &\leftarrow \mathit{odeStmt}_1, \ldots, \mathit{odeStmt}_n \\
\\
\mathit{importStmt} &\leftarrow \texttt{import}\ \mathit{id}_1.\cdots.\mathit{id}_n\ [\texttt{as}\ \mathit{id}_m] \\
\\
\mathit{appModule} &\leftarrow \mathit{id} = f(\mathit{id}_1, \ldots, \mathit{id}_n) \mid \mathit{id}
\end{aligned}
$$

Figure 4.4: Basic *Ode* language grammar in EBNF, covering module definitions, module importing, and use and application of parameterised modules. The module language is built on top of the existing *Ode* grammar provided in Section 3.1.7.

$$
\begin{aligned}
\mathit{odeStmt} &\leftarrow \mathit{modStmt} \mid \mathit{quantityDef} \mid \mathit{unitDef} \mid \mathit{convDef} \mid \mathit{exprStmt} \\
\\
\mathit{modStmt} &\leftarrow \mathit{importStmt} \mid \mathit{exportStmt} \\
\mathit{importStmt} &\leftarrow \texttt{import}\ \mathit{id}_1.\cdots.\mathit{id}_n\ [\texttt{as}\ \mathit{id}_m] \\
\mathit{exportStmt} &\leftarrow \texttt{export}\ (\mathit{id}_1, \ldots, \mathit{id}_n)
\end{aligned}
$$

Figure 4.5: Extensions to the *Ode* grammar (see Section 3.1.7) for the module system. This includes declarations for importing modules and exporting definitions within modules.

return a concrete module, they cannot be partially applied with several modules incrementally or return parameterised modules for further specialisation.

Functor application occurs at the file-level, where the parameters are visible modules defined in the same file or accessible from the imported module environment. We can also apply functors from the interactive console, constructing new modules within the system environment. This enables the custom configuration of models just prior to simulation that exhibit certain behaviours for investigation. When used in conjunction with the scripted console it provides a programmable means for rapidly constructing entire families of specialised models that can be controlled by an external process, compiled, and simulated in succession.

4.2.7 Grammar

The grammar for the module language is provided in Fig. 4.4. The grammar for *Ode* was also updated to allow importing modules and defining exported values, as demonstrated in Fig. 4.5.

4.3 Implementation Details

We present several areas of the *Ode* module implementation, including the type system that determines module interfaces and supports parameterised module application. We introduce the concept of module repositories, collections of independent hierarchies of model modules that may be imported, reused and extended as part of the model development process. This enables collaborative modelling, allowing entire repositories to be updated independently and concurrently by modellers. Finally, we discuss the programming language formed by the module system to construct models. Further details regarding the module language implementation are provided in Appendices A.7 and A.8. It enables the programmable generation of models comprised from typed modules, enabling many mechanisms for model structure and development as were presented in Section 4.1.

4.3.1 Module Type System

Types are used at the module level to ensure the correct usage and safe composition of modules and allow for the separation of module interfaces from concrete implementations. The types of the externally visible expressions within a module form a type signature, as shown in Fig. 4.6, that is essentially the module's type. Similarly, all references to an imported module in a model implicitly creates a type signature that the imported module must implement.

Module signatures are used during type-checking to ensure that modules can be composed successfully, either when a module is explicitly imported or applied as a functor parameter. If the signatures are compatible, module evaluation will succeed, where the provided module must be equal to or a subtype of the required module. The module system implements structural subtyping, where one module is said to be a subtype of another if it contains the required signature as a subset of its own signature, i.e. has a similar 'shape'. This differs to nominative subtyping commonly found in object systems such as Java/C#, where a subtype must be part of an explicit inheritance hierarchy, and can be thought of as a statically-checked variant of 'duck'-typing found within dynamically-typed languages.

This allows imported modules to be swapped for alternatives that export the same interface, enabling rapid modification to models and represents a flexible, strongly-typed manner to reuse

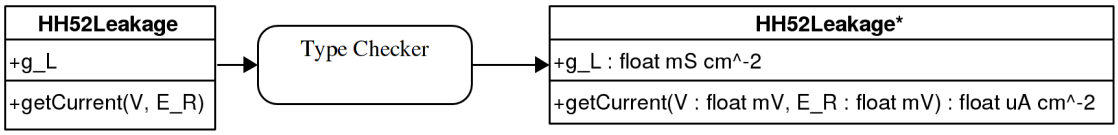


Figure 4.6: Typing a standard module, using the `Leakage` module from the HH52 model. The user-generated untyped module is provided as input, containing a value definition `g_L` and component `getCurrent`. The type-checker returns a typed module, where the definitions have been given inferred types.

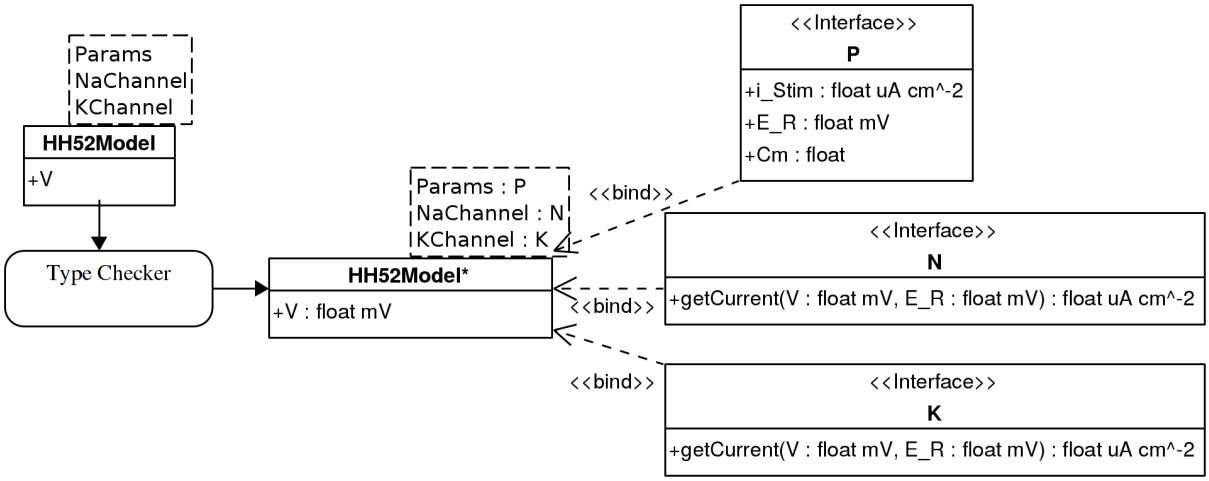


Figure 4.7: Typing a parameterised module, using the HH52 cell model. This module has a single parameter, the membrane potential `V`, and is parameterised with three generic modules, representing the cell parameters and currents I_{Na}, and I_K. The type-checker infers the type for `V` and determines the typed interfaces, `P`, `Na`, and `K` required to satisfy this model module.

model components. For instance, common functionality may be implemented using functor abstractions that accept module arguments containing specific parameters or expressions. Modellers may experiment with multiple implementations that expose the same interface, utilising differing ion channel models within an electrophysiological model. Type signatures are also used to specify the module requirements, encoding the calculations, data, types and units that comprise a module. This may be used as a form of documentation for modellers when comparing modules and substituting them with compatible alternatives. Figs. 4.6 and 4.7 visually indicate the process of typing standalone and parameterised modules, again using sections from a modularised HH52 model provided in Appendix D.4.2.

4.3.1.1 Inference Rules and Unification

Type-checking in *Ode* occurs on a per-module basis, the type signatures and subtyping relationships for modules are fully inferred. This builds upon and utilises the type inference rules for *Ode* expressions discussed in Section 3.3.1, where references to external elements form a set of

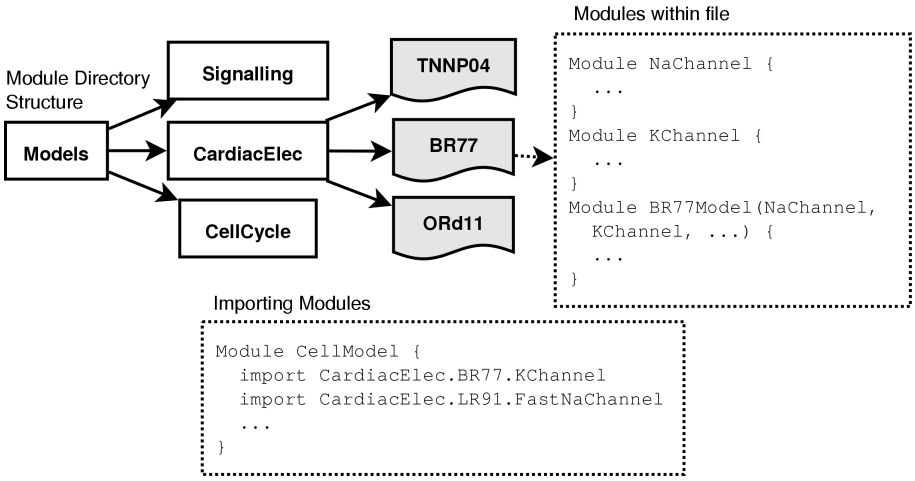


Figure 4.8: A potential structure of models within a biological model repository termed `Models`. Directories are used to separate models by their domain, within which we use files to represent each model, i.e. `BR77`. These then contain a modularised form of the model, i.e. `NaChan`, to encourage collaborative development and reuse. Such a hierarchy is used in Section 4.4.

constraints on an external module. We apply an equality constraint between modules that ensures two modules have the same signature, or that one is a subtype of the other. The set of constraints are unified to generate types for all externally accessible expressions within a module, i.e. the module signature. The module type inference rules are detailed in Appendix A.8.

The module-level constraints are unified in a similar manner to expression-level types described in Section 3.3.1. For standard modules, we require unification to succeed and substitute all type variables for concrete types. For parameterised modules, unification proceeds until no further substitutions are possible and the type variables are stored as a partially-complete signature within the module. The type information obtained from concrete module arguments during functor application is then used to complete unification.

4.3.2 Module Repositories

As mentioned in Section 4.2.1, multiple modules may be created within files located in a directory-based module hierarchy. This hierarchy corresponds to a module repository where modules are referenced by their path relative to the repository root. This may be used to structure and organise biological models to encourage modular development and reuse, as demonstrated in Fig. 4.8.

By default the implementation utilises two root locations to lookup referenced modules, the current directory and a distribution defined directory to be used for implementation-supplied

modules[1]. Additional repository locations may be defined and used from the console in a similar manner to Java's classpath mechanism[2].

This file-based approach to module development brings structural and organisational benefits seen in general-purpose programming languages, such as Java and OCaml, to the modelling domain. It associates each module with a uniform resource identifier (URI) composed from its repository, directory name and module name. We can further use the module interpreter and symbolic-links to create aliases to modules that may be accessed from multiple URIs. It also encourages the use of distributed version control systems (DVCS), such as Git[3] to store, version and distribute the repository, enabling independent module development within a distributed, collaborative system. This will enable modellers to use, develop and reuse the latest modules developed elsewhere, in a manner similar to the rapid open-source development observed on communities such as GitHub[4]. The type and units systems can be used to ensure that the entire repository is always consistent and verified during both individual development and remote repository updates. This could be combined with extensive simulation tests within a continuous integration framework to further ensure the validity of the model repository [74]. To demonstrate this we have created an initial modular repository and framework for describing cardiac cell models in a collaborative manner, described in Section 4.4.

4.3.3 Module Interpreter

A modeller can configure the available module repositories and start a simulation by building, importing, and applying modules, then finally specifying the main model module to simulate from the console — effectively 'programming' the module system. This creation and application of modules forms a top-level module programming language. As in OCaml, this language is a form of the simply-typed lambda calculus that operates over modules [104, 132].

The system interprets commands in the module language just prior to model compilation and simulation; parsing the initial module, following all derived module imports, verifying and building a complete, simulation-ready model from its constituent module parts. The implementation creates new modules, performs module lookup, and evaluates parameterised modules with

[1] On UNIX-like systems this is `$HOME/.ode/repo`.

[2] The classpath is detailed further at `http://docs.oracle.com/javase/tutorial/essential/environment/paths.html`.

[3] `http://git-scm.com/`

[4] `http://www.github.com/`

respect to a module environment containing the uniquely-named modules. The implementation is described further in Appendix A.7. A visual example of this process is provided in Fig. 4.9, where the type-checked HH52 cell model from Fig. 4.7 is instantiated using several concrete modules that represent the cell parameters and ion channels, to generate a new, simulation-ready, HH52 model variant.

This late-bound module initialisation and verification decreases dependent coupling between modules and enables parameterisation and specialisation with no simulation performance penalty [76] due to JIT/run-time compilation. When evaluating the application of parameterised modules, the module body is copied alongside references to all definitions from the imported modules, resulting in a new, standalone module. This process of partially evaluating and unfolding a typed module during application is demonstrated in Fig. 4.9, again using the example from Appendix D.4.2 as was visually depicted in Figs. 4.6 and 4.7.

This interpreted environment facilitates the dynamic construction of models just prior to simulation at compile-time through the *Ode* console. However, as described in Section 3.1.6.3 the system has a static size during simulation. Modules are partially-evaluated and inlined, as are component calls, to generate an unstructured and flattened numerical code block, described further in Section 5.1. The final, constructed (and then flattened) model may be simulated using the language implementation described in the following chapter. The module interpreter allows complex systems to be modelled and simulated through the programmable, piecewise construction of modules prior to simulation, enabling a wide range of investigations into model behaviour.

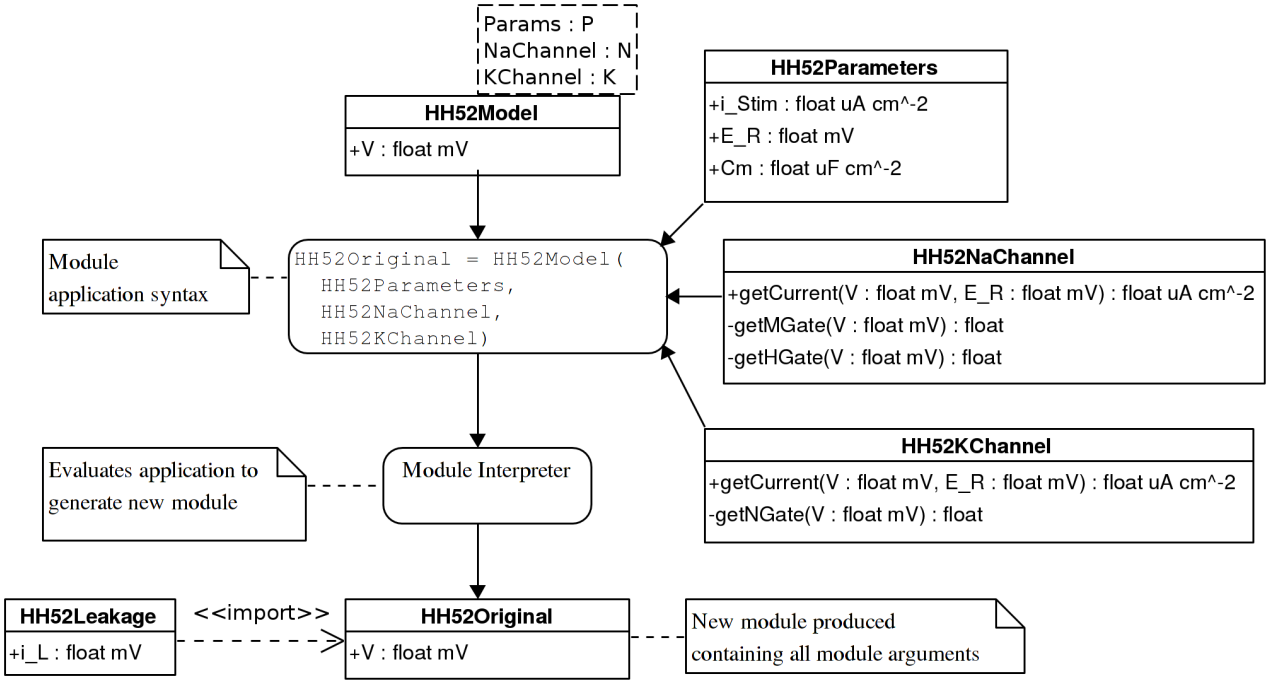


Figure 4.9: Use of the module interpreter to evaluate a parameterised module application to generate specialised modules. The parameterised HH52 cell model is provided as input to the interpreter, with the `HH52Parameters`, I_{Na}, and I_K modules as arguments. Module evaluation results in a new, standalone HH52 model that internally contains the model arguments, and statically imports the HH52 leakage module. The full model code is provided in Appendix D.4.2.

4.4 Simulation Study — Modular Cardiac Models

In this section we utilise the module system and abstractions it provides to design a component-based modular modelling framework. We hope this will eventually enable the rapid development, reuse and investigation/experimentation of cardiac electrophysiological models in a collaborative fashion [107]. This acts as a test of the DSL capabilities to model continuous deterministic systems effectively.

We present the aims of the study, which are related to the aims of the thesis as a whole, within the context of a modelling use-case using published models. In the methodology we cover the architectural decisions involved in creating a modular cardiac framework, based on the techniques and patterns discussed in Section 4.1. This effectively represents our primary piece of research in this study, which is more focussed on enabling effective modular model design than on the simulation results themselves. It includes the construction of a model repository of published cardiac models modularised at the ion channel level. This facilitates the development and reuse of cardiac models in a collaborative manner from their constituent ion channels.

To demonstrate the modular system structure we simulate several programmatically constructed composite models from our repository that use multiple sodium ion channel implementations, and examine their emergent behaviour. We exemplify our work with a combination of *Ode* sample code and UML diagrams to convey the key modelling and architectural concepts.

4.4.1 Aim

The aim of this study is demonstrate how modules and the abstraction features they provide may be effectively used to design and structure biological models that enable model reuse and exhibit low-coupling and high-cohesion. We investigate structure and patterns that may be used to develop late-bound, replaceable and extensible models that can be independently designed and composed safely through module signatures.

This is accomplished within the context of developing a modular cardiac electrophysiological framework that parameterises modules at the ion channel level, with the view that ion channel models are developed and evolved from common experimental data. Our use case is influenced by research in [105] that investigated the common history between cardiac and ion channel models,

however we conduct similar simulations through programmable modular composition rather than manual model construction.

We conduct several simulations using our sample framework, generating AP curves for custom models created from the compile-time substitution of ionic channel modules between models. This is intended to show the ease with which a well-designed framework lends itself to reuse, extensibility and flexibility for modellers to create new models to investigate particular phenomena. Furthermore it demonstrates the use of the type system to ensure subsequent model validity.

4.4.2 Methodology

We investigate and construct a modelling architecture suitable for creating models in a flexible and extensible manner composed 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 complete cardiac models are safely composed in a scripted manner from modules in the repository. They are simulated to examine the effect of alternate channel implementations, whilst demonstrating model reuse and late-bound model construction.

4.4.2.1 Cardiac Models

The models are all cardiac ventricular AP models from a variety of species. They are listed in Table 4.1 and chosen to demonstrate the iterative development process and reuse of parameters and equations in future models. They also form the reference models used to test the DSL implementation in Chapter 5.

We have grouped the models into two sets within the repository, as indicated in Table 4.1. This is based on a natural split in the model's species and complexity (see Section 2.1.3) and eases interoperability, as models in the first set express I_{Na} using $uA.mm^{-2}$, whereas those in the second set use $pA.pF^{-1}$. Within each set we designate a base model, denoted with a * in the table, that represents the cell model whose *Na* channel is both substituted and also reused across the models in each set.

Set 1 — Mammalian			**Set 2** — Human		
Model	**Ion Channels / Pumps**	**State Variables**	**Model**	**Ion Channels / Pumps**	**State Variables**
BR77 [9] *	4	8	TNNP04 [133] *	12	16
LR91 [96]	6	8	IMW04 [75]	11	67
LRd94 [94, 95]	11	12	TP06 [134]	13	20
			GPB10 [63]	13	38
			ORd11 [109]	15	41

Table 4.1: Cardiac ventricular electrophysiological models used within study, detailed further in Section 2.1.3. The models are separated into two sets, with the base model used for comparisons in each set denoted with a *.

4.4.2.2 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 (as described in Section 2.1). We abstract and modularise at this point as most cardiac cell experiments, and subsequent research data, occurs at the ion channel level. The ion channels within a cardiac cell have a known ontology and 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 a single canonical form. This abstraction can be expressed easily within our module system, and 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 channel models was determined that enables their use and replacement within cell models. This interface is illustrated in Fig. 4.10 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. Listing 4.1 contains a segment from the sodium channel of the BR77 model [9] that exports this module interface. It thus implements the type-signature required for reusable channels within our framework. The full model is provided in Appendix D.4.4, alongside a CellML version for comparison in Appendix D.3.2.

All ion channel modules must expose this signature to be compatible with the cardiac models in our framework. A cardiac model structurally depends upon and contains the ion channel

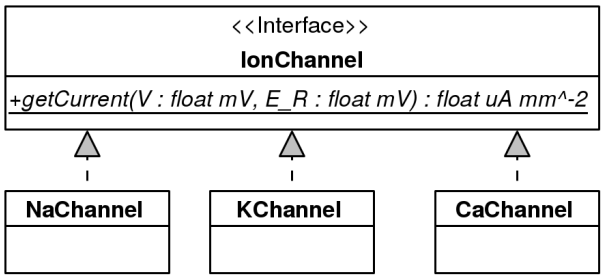


Figure 4.10: `IonChannel` is the interface used to create reusable ion channel models. It consists of a single visible component, `getCurrent`, that computes I_{Ion} for the given ion for use within the membrane voltage calculation. All compatible ion channel models within the system must implement this interface, as `NaChannel` and `KChannel` do.

Listing 4.1 BR77 sodium channel *Ode* code extract, taken from Appendix D.4.4, depicting component `getCurrent` that implements the (required) module signature shown in Fig. 4.10. This component calculates I_{Na} using (not shown) channel gating equations.

```
/* Channel - Sodium Current exporting ion channel interface ***************/
module BR77NaChannel {
  export (getCurrent)
  // BR77 NaChannel gate components (e.g. getMGate(V), etc.) omitted ...
  /* Returns the ionic current generated by this channel ******************/
  component getCurrent(V, E_R) {
    // channel parameters
    val E_Na = 50 { unit : mV }
    val g_Na = 4e-2 { unit : mS_per_mm2 }
    val g_Nac = 3e-5 { unit : mS_per_mm2 }
    // Returns the ionic current generated by this channel
    val i_Na = ((g_Na * pow(getMGate(V), 3.0) * getHGate(V) * getJGate(V)
                + g_Nac) * (V - E_Na)) { unit : uA_per_mm2 }
    return (i_Na)
} }
```

modules, requiring that they expose the `IonChannel` signature in order to be compatible. Fig. 4.11 illustrates this structure used to associate a generic cardiac model termed `CellModel` with its related ion channels, and it is further demonstrated with sample *Ode* code in Listing 4.2. This signature is implicitly specified through use of the channel models within the cardiac module model code, and is checked at compile-time by the type system to ensure that only verified ion channel objects are utilised.

We have used encapsulation to separate definitions for ion channels into modules that expose a specified interface. We now demonstrate how aggregation, subtyping and encapsulation again 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, e.g. channel gates (see Section 2.1), that are then encapsulated and extended in later, more complex channel model subtypes through aggregation (mirroring the real life development of such models). We

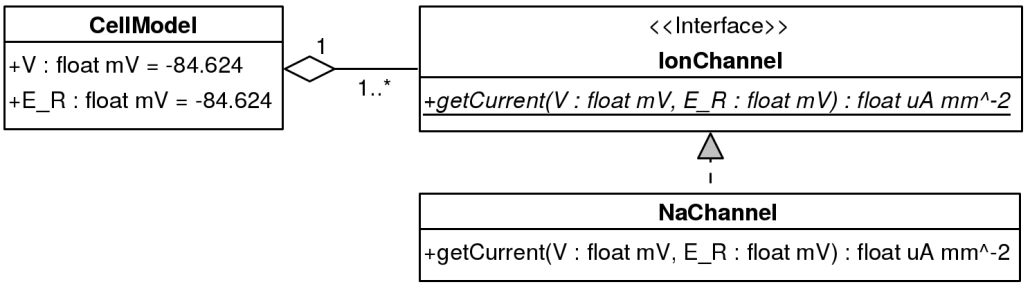


Figure 4.11: Modular structure of a typical cardiac model in the repository. The cell model represents the cellular membrane and contains references to a undefined number of ionic channels, each implementing the required `IonChannel` interface. Thus the cell model can determine I_{Ion} for each channel to calculate the membrane voltage.

Listing 4.2 *Ode* code illustrating the composition of several ion channel models within a typical cellular model in the repository, matching the structural diagram in Fig. 4.11. We use explicit `import` commands to make the specific ion channel models available within the cell module.

```
module CellModel {
  // Import the channel models ...
  import CardiacElec.BR77.BR77KChannel
  import CardiacElec.LR91.LR91NaChannel
  // General cell parameters
  init V = -84.624 { unit : mV }
  val E_R = -84.624 { unit : mV }
  val C = 0.01 { unit : uF_per_mm2 }
  // Channels currents - from modules, other currents omitted ...
  val i_Na = NaChannel.getCurrent(V, E_R)
  val i_K = KChannel.getCurrent(V, E_R)
  // Calculate voltage from sum of currents
  ode { initVal : V } = ((i_Stim - (i_Na + i_K + ...)) / C) { unit : mV.ms^-1 }
}
```

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 of the subtype. These techniques are demonstrated in Fig. 4.12, where we create multiple versions of the `NaChannel` that use aggregation and delegation to reuse and specialise existing code that implements the channel gates and parameters, whilst still implementing the required `IonChannel` interface. Listing 4.3 provides *Ode* code that demonstrates this design pattern for the BR77 *Na* channel model shown in the figure.

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. This is demonstrated in Fig. 4.13, where the model, `BR77Model`, contains the generic module reference `NaChannel` that exposes the `IonChannel` interface. We do not

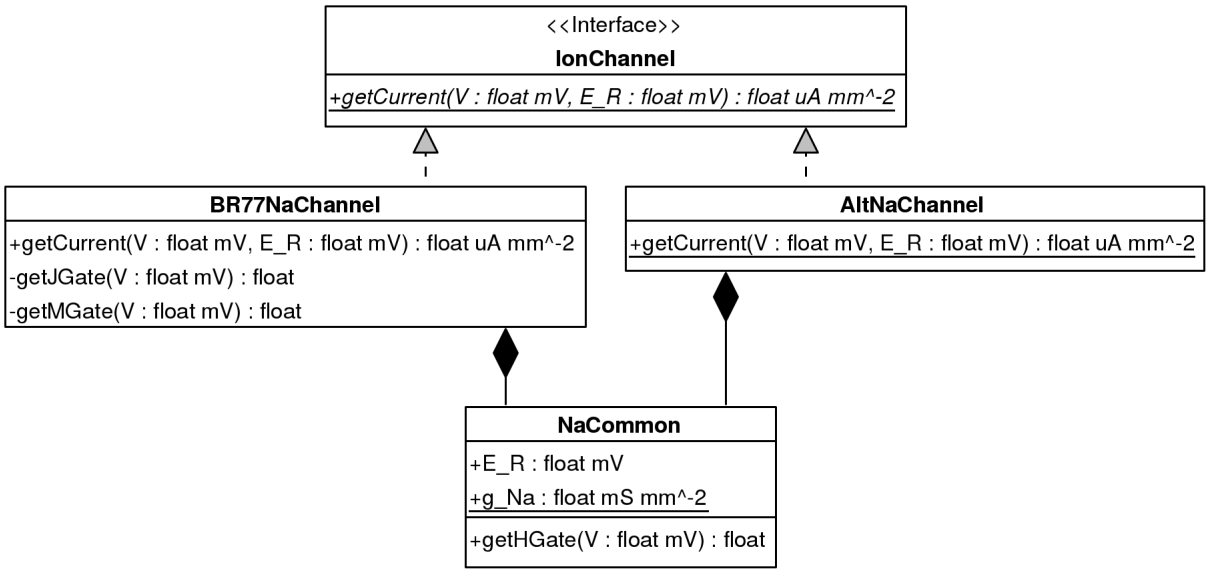


Figure 4.12: Use of encapsulation and aggregation to create reusable module components within the model framework. `BR77NaChannel` and `AltNaChannel` provide differing, but compatible, implementations of the interface. They both aggregate and reference shared data provided by `NaCommon`.

need to specify during model development which specific `NaChannel` implementation we are referring to. Listing 4.4 illustrates this structure with a segment from the BR77 cell model in *Ode* parameterised by a module representing the sodium ion channel. This may be provided by any ion channel model implementing the correct module signature. The full model is provided in Appendix D.4.4.

At simulation-time the `BR77NaChannel` [9] or `LR91NaChannel` [96] modules may then be safely provided as a generic parameter to the `BR77Model` module, and other compatible models. When applying generic objects the compiler ensures that the signatures of modules provided as parameters are consistent with the model specification. Providing an inappropriate type as a parameter will present the modeller with a compile-time error, ensuring that models will be correctly defined and composed prior to simulation.

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 and model design patterns aid model development by increasing cohesion and decreasing coupling.

Listing 4.3 *Ode* code fragment implementing the UML structure provided in Fig. 4.12. The code demonstrates the use of encapsulation and aggregation to construct an implementation of `BR77NaChannel` that reuses shared data from `NaCommon`.

```
/* Common data reused/shared between Sodium channel models ***************/
module NaCommon {
  val E_Na = 50 { unit : mV }
  val g_Na = 4e-2 { unit : mS_per_mm2 }
  // Channel gate - h
  component getHGate(V) {
    // ... calculation of 'h' omitted ...
    return (h)
} }

/* Channel - Sodium Current exporting IonChannel interface ***************/
module BR77NaChannel {
  export (getCurrent)
  import CardiacElec.Common.NaCommon
  // Other BR77 NaChannel gate components, e.g. getMGate(V), omitted ...

  /* Returns the ionic current generated by this channel *****************/
  component getCurrent(V, E_R) {
    val g_Nac = 3e-5 { unit : mS_per_mm2 }
    val i_Na = ((NaCommon.g_Na * pow(getMGate(V), 3.0) * NaCommon.getHGate(V)
                * getJGate(V) + g_Nac) * (V - NaCommon.E_Na))
    return (i_Na)
} }
```

4.4.3 Results

This section presents the results obtained from our study into modular model development, including the creation of a model repository containing initial modularised cardiac models implementing the discussed architecture, and examples of the types of custom-configured simulations such a framework can provide.

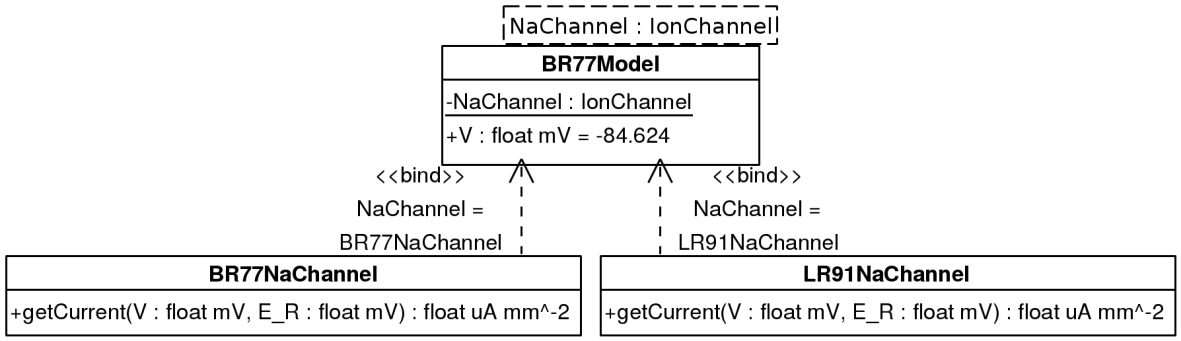


Figure 4.13: Use of generics within the model framework. Either `BR77NaChannel` or `LR91NaChannel`, implementing the `IonChannel` interface, can be applied as module argument to the base `BR77Model` cardiac model to represent the generic `NaChannel`. As in Fig. 4.2 we indicate the application of a parameters module using a binding relationship.

Listing 4.4 BR77 code extract indicating use of the parameterised `NaChannel` module within the cellular model, as per the modular structure shown in Fig. 4.13. The channel current is used with other (not shown) BR77 channels to calculate the transmembrane voltage.

```
/* Main BR77 Cell Model ************************************************/
module BR77Model(NaChannel) {
  // Cell parameters
  init V = -84.624 { unit : mV }
  val E_R = -84.624 { unit : mV }
  val C = 0.01 { unit : uF_per_mm2 }
  // Other channels (e.g. Slow Inward, Time-dependent Outward) omitted...
  // Channel - Sodium Current from parameterised module
  val i_Na = NaChannel.getCurrent(V, E_R)
  // Calculate voltage from sum of currents
  ode { initVal : V } = ((i_Stim - (i_Na + i_s + i_x1 + i_K1)) / C)
}
```

4.4.3.1 Model Repository

We created a model repository containing representations of the cardiac models listed in Table 4.1, where modules are used to represent several ion channels within the cell. The main module that calculates the membrane voltage for a cardiac cell model is parameterised by its fast-sodium channel, `NaChannel`. This model structure is illustrated in Fig. 4.14, depicting the process of binding a chosen I_{Na} model to the cardiac model under investigation at compile-time to generate a new simulation test-case. This simulation-time construction of custom models is performed through the *Ode* console interface using the module interpreter described in Section 4.3.3. The commands used to generate the BR77 and LR91 model combinations are provided in Listing 4.5.

The ion channel modules are organised within the repository by their originating model, and separated into sub-modules influenced by the model design and natural biological structure of the system, rather than following a fixed ontology. This is influenced by the hypothetical hierarchy provided in Fig. 4.8. The repository enables modellers to create new models based on specific specialisation or usages of modules in order to investigate specific hypotheses, such as mutated ion channels or those affected by drugs [94]. Additionally, by placing the module repository under version control we create an environment suitable for interactive and collaborative model development. This enables multiple modellers to independently develop and modify the module subcomponents that comprise a simulation-ready module, with the *Ode* verification features ensuring the repository's internal consistency.

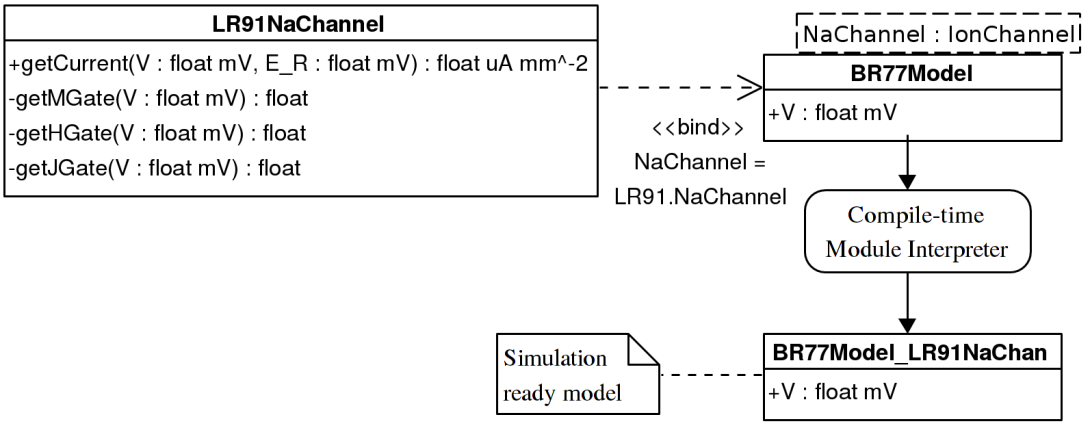


Figure 4.14: Compile-time application of modular cardiac model components. Here we apply the LR91 I_{Na} model to the BR77 cellular model for compile-time evaluation. This results in a new custom model, `BR77Model_LR91NaChan`, that may be simulated to investigate the change on the BR77 AP.

Listing 4.5 Sample from *Ode* console script used to compose custom cardiac models from cellular and ion channel module components at compile-time to perform the *in silico* experiments used in this study.

```
// import BR77 model components
import CardiacElec.BR77.BR77Model
import CardiacElec.BR77.BR77NaChannel

// import LR91 model components
import CardiacElec.LR91.LR91Model
import CardiacElec.LR91.LR91NaChannel

// construct original models and custom models
module BR77Original = BR77Model(BR77NaChannel)
module BR77Model_LR91Chan = BR77Model(LR91NaChannel)
module LR91Model_BR77Chan = LR91Model(BR77NaChannel)
```

4.4.3.2 Model Simulation

We compiled and ran simulations that demonstrated the substitution and impact of the *Na* channel within each set of models listed in Table 4.1. This was performed in a scripted manner at compile-time, as shown in Listing 4.5. These simulations were inspired by the work in [105], and the modular framework was structured to enable such model modification and experimentation rapidly and safely without error. The simulations were performed using a forward-Euler ODE solver with the parameters listed in Table 4.2. The initial conditions were unchanged from those specified in each model's original publication [9, 63, 75, 94–96, 109, 133, 134].

Using the first set of models, we initially composed each cardiac base cell model with its corresponding sodium channel and simulated the resulting complete model to generate the reference AP curves depicted in Fig. 4.15. The sodium channels from the remaining models

Model	t_{start} (ms)	t_{stop} (ms)	h (ms)	Output Period (ms)
BR77 Set	0	5000	0.001	0.5
TNNP04 Set	0	5000	0.001	0.5

Table 4.2: Simulation parameters used to generate the results for the modular study.

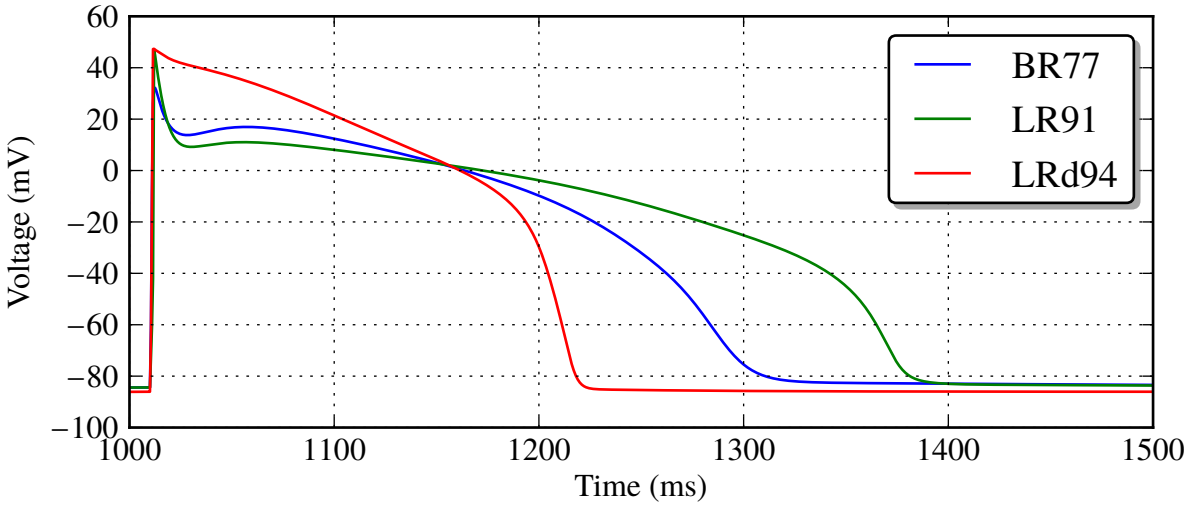


Figure 4.15: AP curves from applying each I_{Na} channel into its matching cellular model using modules, testing the modular design and acting as a reference.

in the set were then applied to the BR77 cell model to generate a new family of models whose AP curves are presented in Fig. 4.16. 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; this is expected, as sodium ions are responsible for AP initialisation.

We then performed the inverse operations, taking the I_{Na} BR77 model and applying it into the remaining cellular models to investigate its influence. The AP curves from the simulation of these models are presented in Fig. 4.17, where replacing the I_{Na} from the LR91 and LRd94 models increases the duration of the AP and alters the peak AP. However the variations are minor and suggests some commonality between the models and their experimental data. 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 configurable models.

We perform a similar series of custom, compile-time compositions and simulations using the second, more complex, set of models, where the TNNP04 model acts as the base model under investigation. The results are presented in the same form, and vary more wildly due to the

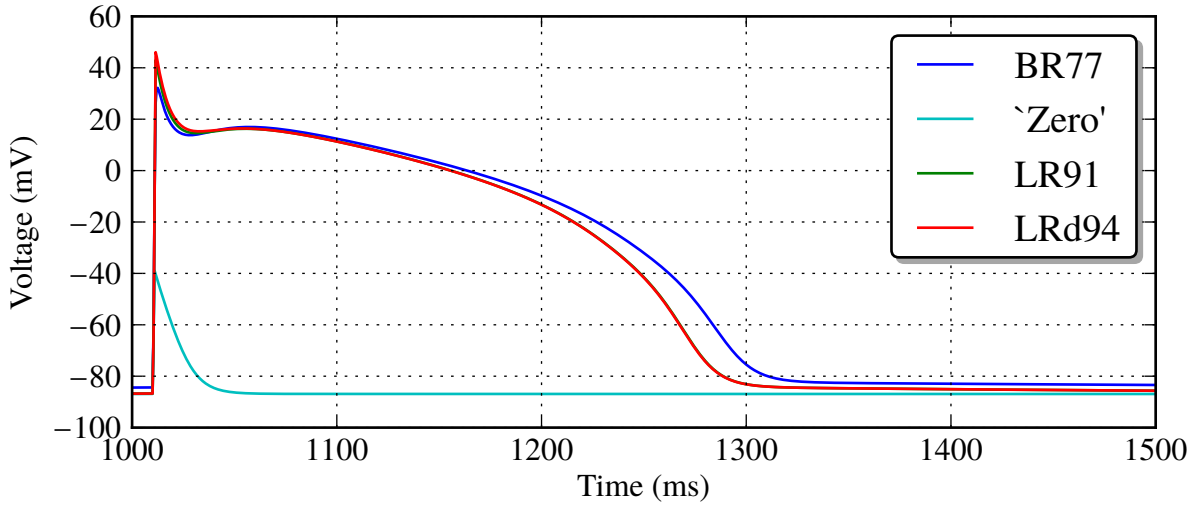


Figure 4.16: AP curves from applying alternate I_{Na} channels into the BR77 cardiac cell model using modules. The channel models in the set generate similar APs, suggesting commonality.

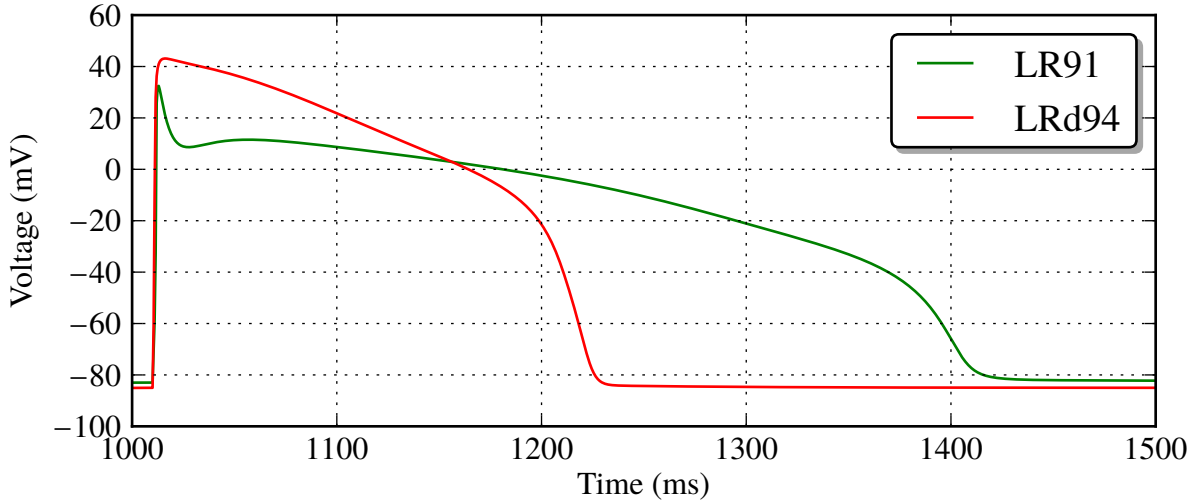


Figure 4.17: AP curves from applying the BR77 I_{Na} channel into alternate cardiac cell models using modules. The BR77 channel slightly alters the AP peak and duration in the models.

increasing model complexity. Fig. 4.18 plots the reference APs generated from models composed with their corresponding channel.

Fig. 4.19 plots the APs generated from applying the remaining sodium channels within the set to the TNNP04 cell model. It shows that most alternate sodium channels do not greatly alter the TNNP04 model AP, although the GPB10 I_{Na} model shortens the AP duration and I_{Zero} omits the AP peak, again expected as sodium ions are responsible for AP initialisation. Finally Fig. 4.20 plots the APs generated from the applying the TNNP04 I_{Na} model to the remaining cellular models in the set. Changes are seen to the AP peak in several cases and it results in an unrecognisable AP for the GPB10 model with an exaggerated peak. [63] states that the GPB10

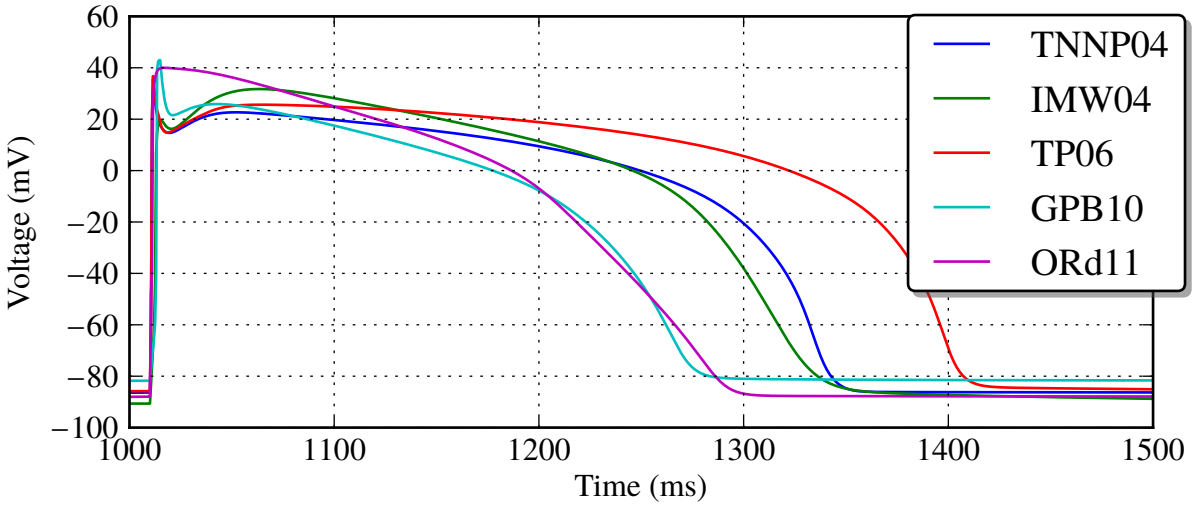


Figure 4.18: AP curves from applying each I_{Na} channel into its matching cellular model using modules, again acting as a reference.

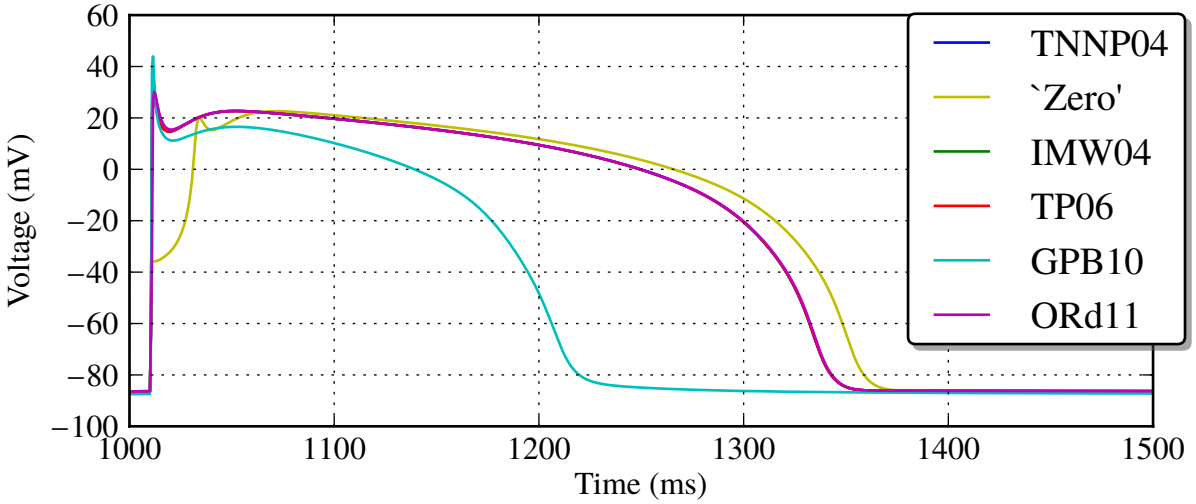


Figure 4.19: AP curves from applying alternate I_{Na} channels into TNNP04 cardiac cell model using modules. The channel models in the set generate similar APs apart for the GPB10 channel.

model is entirely derived from newly obtained experimental data, potentially explaining the large variation in generated APs and suggesting some similarity between the *Na* channels in the remaining models.

4.4.4 Discussion

In this study we have looked at software engineering patterns and abstractions that the *Ode* DSL provides for biological modelling that enable modularity, encapsulation, reuse and extensibility. We demonstrated their use and the potential for collaborative modelling through the creation of a repository of cardiac electrophysiological models. We feel this work forms a set of practices and

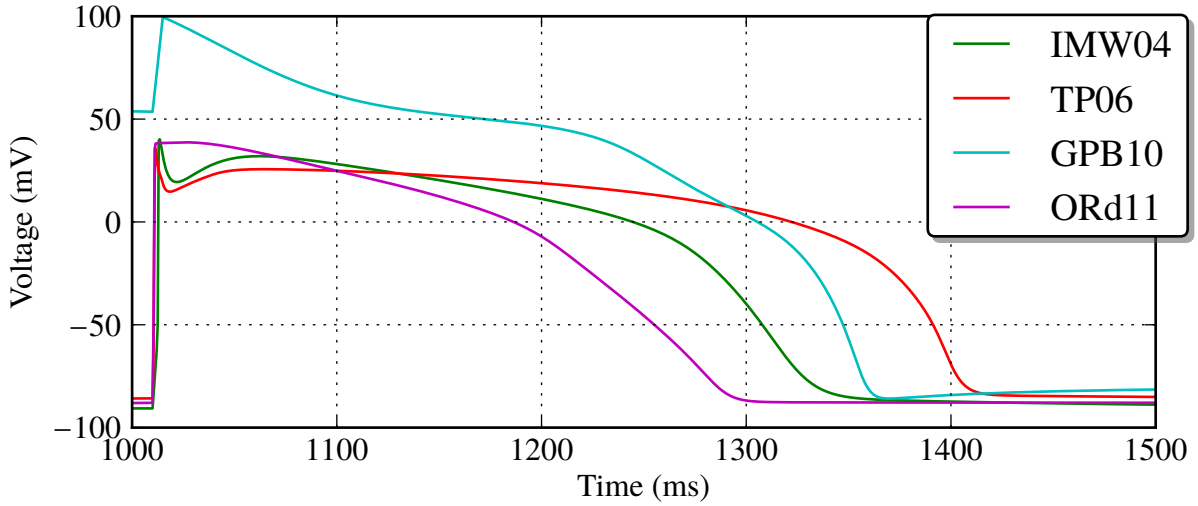


Figure 4.20: AP curves from applying TNNP04 I_{Na} channel into alternate cardiac cell models using modules. The channel causes slight AP variations in most models, although greatly affects the GPB10 AP.

patterns that initially define the use of model-based software engineering in our DSL.

The repository may simply be used to reproduce existing models. However a modular repository for collaborative and iterative model development enables other use-cases, facilitating model module reuse and the creation of alternative implementations. We demonstrated this through the creation and mixed usage of ion channel representations from a variety of models in our simulations. The text-based DSL and module system naturally map onto the facilitates provided by distributed VCS systems, enabling the usage of modules within a continuously updated repository in a collaborative fashion. When combined with the strong verification features of the DSL this provides a robust environment for collaborative model development. This could be extended with testing features from agile development, where entire scripted test-suites could be created that test the models within the repository upon each committed change [74]. Such continuous integration would validate and simulate the models, the results of which could be compared with previously generated data to detect and minimise regressions. This would effectively be performing functional curation against previously developed models [32].

Our use-case driven design process has yielded a modular framework which allows a particular family of cardiac electrophysiological models to be extended intuitively and with relative ease. In a large part, 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. We presented the design patterns and abstractions used in creating such a reusable model framework that may be extended and applied to other biological modelling domains, forming a physiological pattern language.

Several simulations were performed from the programmable composition and substitution of cardiac models and fast sodium channels within our repository. These serve to demonstrate the application of the architectural approaches discussed to construct reusable, modular models using the abstraction features provided by the DSL, rather than representing biologically-meaningful results. Encouragingly, a recognisable AP trace was generated from most composed models, demonstrating the similarities between models representing common biological systems and hinting towards the iterative development and reuse of biological data between models. In several cases the substituted sodium channel altered the peak AP response, as expected considering the role of Na^{2+} in the cardiac cycle (see Section 2.1.1).

The ion channel interface, illustrated in Fig. 4.10, proved suitable for modularising simpler cardiac models, considering each ion channel as an individual element that is only dependent on the cellular transmembrane potential V_m. However this view does not account for ion exchangers and pumps found in more complex models that can affect the channel current. Developing these models further to utilise a more complex interface that fully captures cellular and channel interactions would be a major undertaking conducted in conjunction with modellers and physiologists. However the eventual framework would provide many use-cases for modellers to programmatically construct specialised models and conduct unique investigations easily.

We envisage several benefits this approach may bring to the cardiac electrophysiological modelling community. Firstly, models developed in this fashion obtain the software engineering benefits observed from using modularity — that is, ease of development, separation of concerns and reduced maintenance. We believe, as do the authors of [105], that cardiac subsystems, e.g. ion channels, will eventually coalesce into canonical representations as further experimental data becomes available. Modularity would enable development and reuse of a specific component as a single entity within complementary and competing modelling investigations into cardiac behaviour. In the meantime modularity enables substitution of alternate representations based on different experimental data, or those that exhibit differing characteristics. For instance, as in [109], channel models that explicitly utilise HH-formulation over Markov-formulation representations

for performance and complexity reasons to aid model development and simulation.

Ion channels are often based on reused experimental data and then modified to fit a model. Interchanging differing channel representations enables studies into the phenomenological background of specific channels. These can be used to trace the lineage of experimental data such that it can be explicitly related to parts of a model to increase understanding, as demonstrated in [105]. A related benefit is enabling *in silico* experimentation in relation to multiple experimental data whilst deriving a common model. For instance, current research into modelling the hERG gene that encodes the delayed rectifier current I_{Kr} in cardiac muscle. In [138] the authors compare different possible hERG models for activation kinetics with experimental data and use the error between simulation and experiment to attempt to infer the most appropriate model. These could be developed as interface-compatible channel modules that may be automatically substituted and composed to create custom models for simulation as required. The benefits listed above are just several we believe modularity will provide. As mentioned it would require substantial effort on the part of biologists, physiologists and modellers to determine the correct interfaces and develop models in a modular manner. Our research represents only the starting point for such work.

We have demonstrated how generics may be used to facilitate the creation of a reusable repository of cardiac model components parameterised at the ion channel level. We have used examples from models implemented in our modular framework to exhibit how encapsulation, aggregation and subtyping may be used to enable compile-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. Such abstraction techniques increase cohesion and decrease the coupling of modules. However, their usage comes with a cost; modellers are required 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 well suited to teams working together in a collaborative fashion. Even though the up-front investment involved in structuring one's designs in a modular fashion may not be immediately relevant or beneficial, the ability to rapidly modify code enables a more adaptive and agile approach to long-term, reusable, model creation.

4.5 Discussion

In this chapter we extended the *Ode* DSL to provide structured, typed modules with well-defined import and export semantics. This can be used to group and encapsulate related functionality into module components that may be shared and reused within other models. Modules provide an abstraction over functionality that is captured by the module signature. This can be used to create more complex models, enabling a separation of concerns during model development. Parameterised modules provide a means to structure common reusable abstractions that may be altered for specific use-cases. They may be safely mixed and composed to easily generate specialised modules. Module signatures provide a means to separate module interface from implementation, allowing for alternative implementations that export compatible type-signatures. This enables the dynamic creation of custom models just prior to simulation at compile-time through the module interpreter; during simulation the generated models are static with a fixed system size.

The type system was extended to operate upon modules, ensuring that modules may be successfully composed and simulated with respect to their types and units. This occurs at compile-time, providing feedback to modellers regarding the correctness of their modules prior to simulation. We believe that the structural subtyping system used to determine module compatibility is more flexible than the nominative subtyping systems used in many OO-languages, providing program flexibility as seen in dynamically-typed systems in conjunction with strong static guarantees. Implicitly inferring all module signatures greatly increases the ease with which modellers may use the system, although this is not without drawbacks as in some cases explicit type signatures can provide valuable documentation. We also demonstrated how the module system enabled the programmable construction of specialised models at simulation-time. Typed module interfaces allow the system to check and verify the construction and composition of a model, similar to type-checking expressions in the previous chapter. Furthermore the module system aids validation by enabling the use of multiple representations of model subcomponents when performing *in silico* experimentation.

We discussed the use of modules and DSL abstraction features to structure and develop complex models based on several identified physiological modelling patterns. For instance, we

modularised cardiac models using ion channels, identified an initial cellular/channel interface, further used abstraction and encapsulation at the channel/gate level, and generics to abstract model parameters and high-level cellular components. This allowed us to construct reusable components, exhibiting low-coupling and high-cohesion, in a manner similar to large-scale software projects. Our modular patterns made use of module repositories that can be collaboratively developed, shared, and reused in a manner similar to libraries and packages in general-purpose languages. We investigated model-based software engineering from both a theoretical viewpoint through the introduction of the module type system and interpreter, and a practical one covering model curation, deployment, collaborative development and design patterns [68].

Our research into reusable model development and model architecture in *Ode* was put to use in a case study. This involved the development of a systems architecture that supported the creation of a family of reusable, parameterised cardiac models with modularised ion channels that communicate across strongly-typed interfaces. It also served as a minor biological modelling investigation into the use of components from differing models to mimic the iterative development and reuse of experimental data underpinning biological models, demonstrated through the simulation of models containing substituted sodium channels.

4.5.1 Modular Biological Modelling

The simulation study presented an opportunity to exhibit our work on the module system, showcasing the abstraction features to develop reusable, modular biological models. Within this section we present several further uses of the module system that may aid the modelling process.

Firstly, utilising generic objects to model ion channels within cardiac systems would enable more advanced or detailed representations of ion channels, perhaps representing newer experimental data, to be placed and substituted into existing models. Alternate representations of an ion channel may be utilised within a model, for instance to study in detail the function of a particular ion channel property within an integrative cell model or even to reduce the computational complexity of a model [109]. We may apply functional curation at the module level, expressing individual ion channels in our case, to ensure these alternate representations can be successfully tested both against each other and observed experimental data. The module system allows us to abstract the model parameters and experimental protocol from the model code itself, allowing

each model representation to be simulated and compared accurately under the same environment when testing. This was demonstrated with abstraction of the HH52 cell model parameters using UML figures and *Ode* code in Sections 4.1 and 4.2, combined with use of the *Ode* console (see Section 3.4) to compose modules and script simulations, as demonstrated in Listing 4.5.

New cardiac models may be created by modellers that utilise existing ion channel models. Thus via aggregation the DSL can capture the reuse of models that occurs at the mathematical level in cardiac model development, thereby increasing robustness which is especially important if these models are ever used in a predictive pharmaceutical or clinical setting. We can also create hybrid models that include ion channel representations derived from a variety of models, for instance when performing phylogenetic studies into the derivation and creation of models based on limited experimental data [105]. This can be done automatically by the module system in a collaborative manner simply by utilising ion channel modules from differing models when instantiating a generic cardiac model module, as demonstrated by Fig. 4.14 in the case-study.

Finally parameterised modules may be used to alter models for a particular simulation, e.g. when performing sensitivity analysis of the model to parameter fluctuations, or to alter equations, e.g. when modelling ion channel changes and resulting effects on the AP caused by mutation or drug block [75]. As an example, type-correct generic adapter modules may be created that simply reduce the current of a generic ion channel parameter by 50%, as demonstrated in Listing 4.6. This would enable investigations into drug block in an abstracted manner, similar to the 'Zero' *Na* channel used within the case study. Alternatively they may be used to alter the calculations to clamp various calculated parameters, to modify and abstract over the stimulus protocol, or to validate and verify experimental data without altering the original model.

Models created using the techniques that we have developed 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. In the following chapter we detail the *Ode* backend used to simulate these models. This includes optimisations to efficiently implement the advanced modelling and structural features described in this chapter.

Listing 4.6 *Ode* code illustrating how we may create a generic adapter module that reduces the current of an ion channel by 50%. Both the input module and the returned output module implement the `IonChannel` interface and may be substituted without code change.

```
module CurrentBlock(IonChannel) {
  val block_amount = 0.5
  /* Returns and modifies ionic current generated by this channel *********/
  component getCurrent(V, E_R) {
    val i_Ion = IonChannel.getCurrent(V, E_R) { unit : uA_per_mm2 }
    return (i_Ion * block_amount)
}
```

Chapter 5

Simulation Implementation

Having defined the *Ode* language and created a suitable set of models, in this chapter we describe the implementation of the system backend that provides efficient model simulation. The implementation simulates a verified model according to the DSL semantics over a specified time interval and timestep, whilst saving the model state to disk at regular intervals for offline analysis. The implementation aims to address several of the issues raised in Section 2.4, with a particular focus on computational efficiency utilising a just-in-time (JIT) native code compiler that generates optimised and specialised simulation code for each model. A basic interpreter for the DSLs was developed to act as the reference implementation for testing simulation correctness. As discussed in Chapter 3, we created a DSL rather than extend CellML as we were interested in the introduction of a programmable operational DSL, with features such as typing and modularity seen in general-purpose languages, to the modelling domain. Similarly we did not implement *Ode* within an existing functional language, such as OCaml, as we require low-level control over the compilation process to produce optimised, CPU-efficient code. For instance, we wish to ensure that all structural abstractions can be implemented at compile-time with negligible run-time cost.

We intend to create a simulation platform that utilises multiple compilation stages and run-time code generation to create optimised model simulation code. We believe that the compilation stages will be able to resolve and pre-compute all state-independent expressions in the model, and provide detailed semantic information to the code-generation backend for low-level optimisations.

To achieve this the model will be first converted into a lower-level intermediate representation (IR), termed *CoreFlat*, that makes control-flow explicit and is amenable to optimisations. From this we utilise the LLVM code-generation library [86] to generate simulation and model

computation code at simulation-time. The DSL semantics will enable the effective use of the extensive and sophisticated LLVM optimisations. We introduce the concept of a *simulation kernel* that represents the low-level numerical operations performed within each simulation timestep and is extremely performance sensitive. This is expressed directly within the backend IR. We implement several configurable optimisations that will hopefully enable large improvements in simulation performance. These include full module and function inlining, common low-level optimisations such as constant propagation and common sub-expression elimination, and a novel auto-vectorisation optimisation.

We plan to code-generate custom, model-specific, ordinary differential equation (ODE) solvers in LLVM assembly during the model compilation process. These can be directly linked and integrated within the model at simulation-time, enabling further performance enhancements through link-time optimisation (LTO) [3, 86]. An integrated adaptive solver, based on CVODE, is developed to simulate efficiently certain classes of models.

This late-bound, staged approach, in conjunction with the careful design of the DSL semantics and knowledge of the LLVM process, results in a competitive high-performance simulation engine. This novel approach to simulation development is required as the limits of numerical computation via traditional general-purpose languages are reached for modern and future architectures. High-performance simulation will enable development of more complex and detailed models. Run-time linking and optimisation of the model and simulation engine will enable further model-specific optimisations.

To test both our implementation and optimisations we include simulation and performance benchmarks against a variety of other numerical simulation platforms using a selection of cardiac ventricular electrophysiological models developed in *Ode*.

Fig. 5.1 indicates the general overview of the *Ode* backend discussed in this chapter. In Section 5.1 we detail the compilation pipeline, consisting of transformation stages that result in an optimised model within the backend IR. In Section 5.3 we describe the translation of the backend IR into LLVM instructions, providing details of the low-level simulation structure and performing simulation benchmarks. Finally in Section 5.4 we describe several optimisations implemented at both the backend IR and LLVM IR level, and demonstrate their effectiveness.

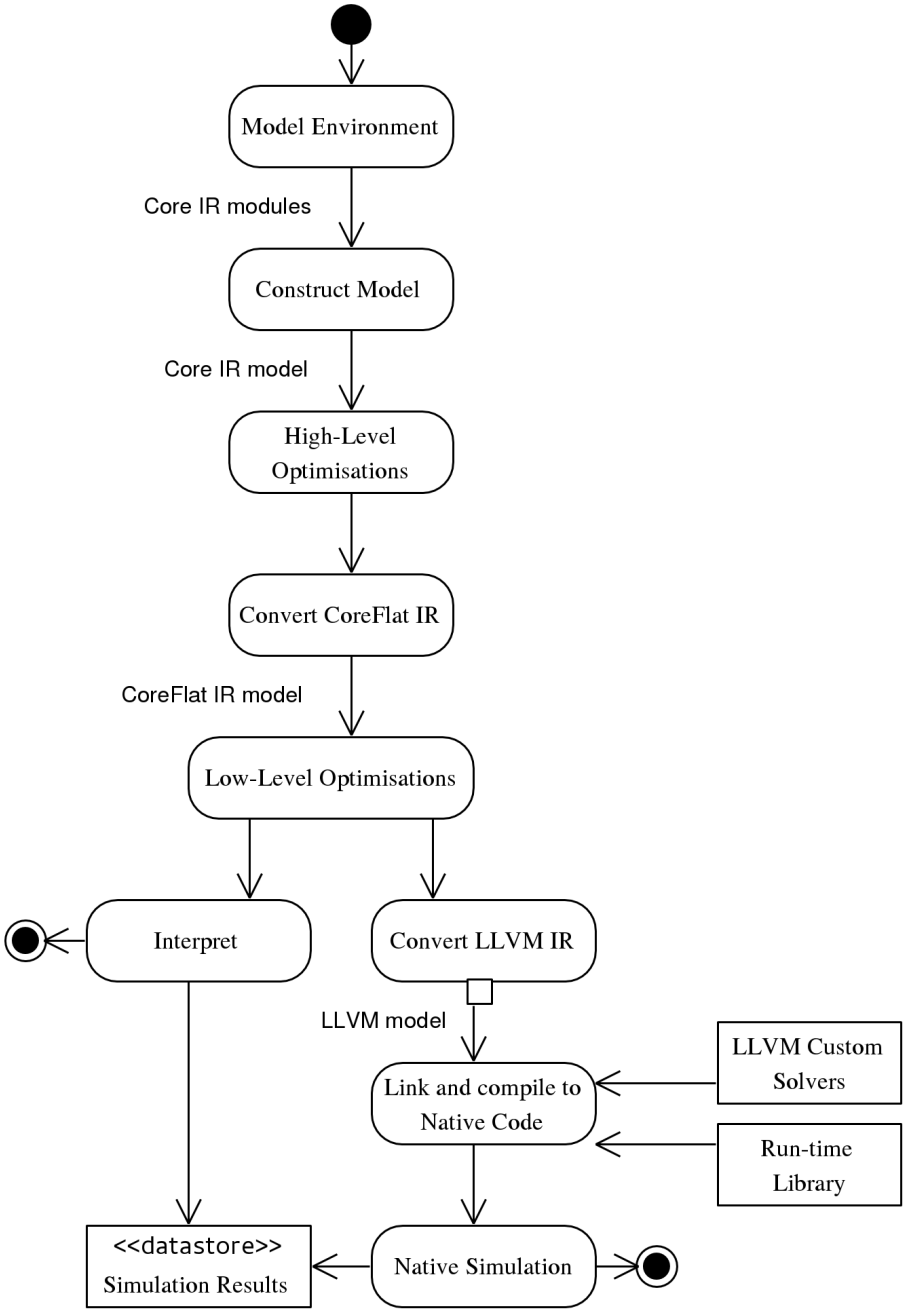


Figure 5.1: General overview of the *Ode* backend, starting with the selection of an available module for simulation. The dependent modules are pieced together into a single model, and several high- and low-level optimisations are performed. This optimised model may be interpreted or converted into native-code via LLVM for high-performance simulation.

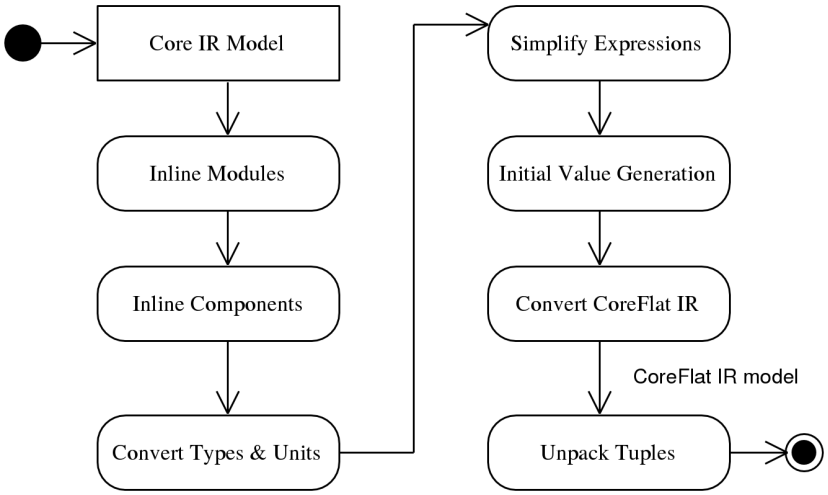


Figure 5.2: Flowchart depicting the *Ode* backend pipeline stages, with each stage optimising and/or simplifying the model code representation.

5.1 Compilation Stages

Ode uses a traditional compiler structure, with distinct front-, middle- and back-end stages, as illustrated in Fig. A.1. The implementation frontend reads an *Ode* module, runs consistency checks and translates the module into the *Core* IR. Within the implementation middle-end, a *Core* module is type- and unit-checked, verified and its modules are imported via the module language. At this stage the system has constructed a main model module that has been verified and found suitable for simulation.

The middle- and back-ends converge at this point. When configuring a *Core* module for simulation, the implementation passes the verified model through multiple stages, optimising, simplifying and adding metadata at each stage, resulting in a *CoreFlat* IR model that can be simulated by the backend. These stages are illustrated in Fig. 5.2. The implementation will generate optimised, specific, solvers for this final *CoreFlat* model and execute the simulation according to the user-defined simulation parameters.

This section describes several of the more important compilation stages that comprise the implementation middle- and back-end. These stages aim to generate a low-level representation of the entire model to enable partial evaluation/pre-computation [76] and whole-program optimisation [3, 5]. When combined with the run-time code-generation process described in Section 5.3 it provides an end-to-end, optimised model development and simulation pipeline.

Transformation stages applied to the *Core* IR include flattening of all modules and inlining of

all components, resulting in a single block of numerical simulation code. At the same time, type information is erased, and unit conversion expressions are applied. Some pipelines stages are explained elsewhere in this chapter, for instance *Optimisations* in Section 5.4 and *Conversion to* CoreFlat *IR* in Section 5.2. Others, such as a whole-model optimisation to track and unpack values from aggregate structures, are omitted for space reasons.

5.1.1 Module Inlining

The module system described in Chapter 4 provides powerful mechanisms to structure models into reusable components, with support for generic modules, code reuse, and OO-like concepts such as aggregation and inheritance. The module system was designed such that all features are implemented at compile-time with no run-time cost, with the intention that structural abstractions do not impact high-performance simulation.

In Section 4.3.3 and Appendix A.7 we detailed the mechanisms of the module interpreter over the module term language, resulting in a simulation-ready module. Within the backend we recursively import and inline all dependent module code referenced from the selected module into a final, custom module. This process is illustrated in Fig. 5.3 and provides several advantages. It allows the entire model to be optimised later-on as a single unit, providing whole-program optimisation. It ensures that the model is always comprised from the latest definitions of its module components, and that all modular abstractions are removed and unused during run-time. Combined with run-time code generation it may be thought of as providing the performance benefits of static module linking with the flexibility of run-time dynamic linking.

However, as per static linking, module inlining increases the final code size. This occurs whenever modules are reused and imported in multiple locations or used as functor arguments. This in turn increases cache pressure and decreases numerical performance. As such the inlining algorithm acknowledges previous imports and patches all references within the inlined code to refer to the previous import locations rather than duplicating code. Parameterised module imports are handled in the same way, thanks to the partial application and evaluation of functors within the module interpreter. Due to the type-checked module system, this patching and module reuse can occur without causing errors during model simulation.

Finally module inlining results in the entire model code being available within a single

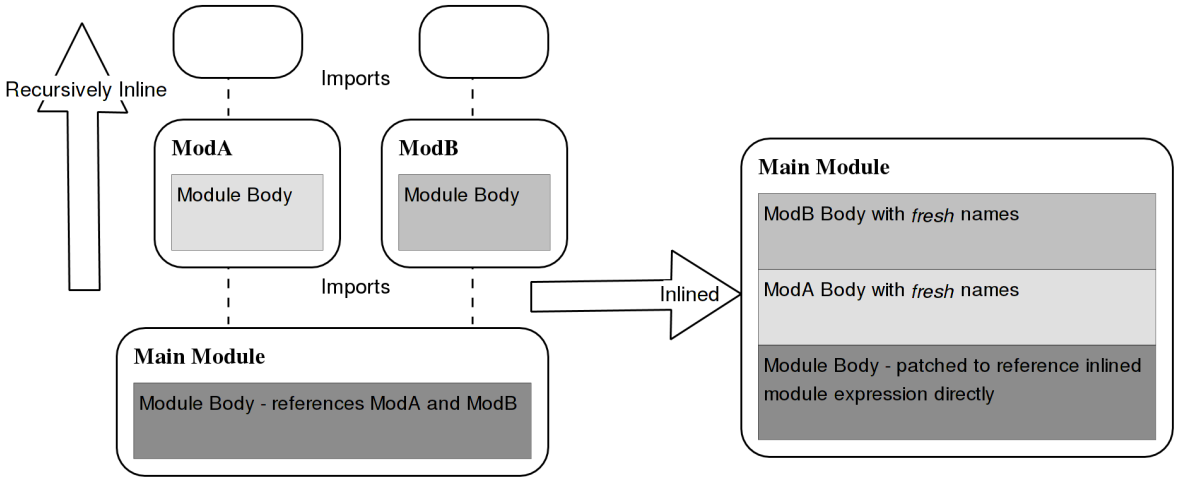


Figure 5.3: Illustration of the module inlining process. Starting with the main module, all dependent modules are recursively processed and inlined within one another, resulting in a single module containing the entire model.

translation unit. Thus all subsequent model-specific and lower-level LLVM optimisations [86] operate on the whole-program/model, widening their scope and reach. Inlining also acts as a form of dead-code elimination, where only referenced modules from the available and loaded module environments are exported into the final model simulation code.

5.1.2 Component Inlining

Components are used within *Ode* to abstract model computations for reuse. They could be implemented within the backend using function abstractions and calls. However we would like to avoid function call overhead, such as saving the program counter, passing parameters, branching and potential pipeline stalls, within the highly performance-sensitive generated simulation code.

The limited flow-control within *Ode* makes it possible to fully inline all components recursively to the top-level, introducing fresh variables at each step to avoid name clashes. This will minimise the basic-blocks within the native model code, and constant folding and propagation may be applied during code-generation to remove any additional variables.

Component inlining presents a trade-off between function call overhead, and larger code size and potential cache misses. Investigations into compiled output indicated that even highly complex biological models were only several kB in size and would fit within the instruction cache of most CPUs, validating the decision to perform inlining.

It is also semantically important that we copy and inline components as we require all initial values to be unique and bind to a single simulation construct — component inlining provides

a mechanism for reusing complex logic. Components then can be considered programmable templates that exhibit properties similar to the 'factory' design pattern in OO-languages [51], abstracting stateful constructs such as initial value and ODE definitions for multiple uses within a model. Finally, as per module-inlining, component inlining acts as a form of dead-code elimination [3], where unused components and definitions within the whole-model will be discarded.

5.1.3 Convert Units and Type

This stage removes redundant type information from the model having previously performed type-checking. Currently this involves converting units-annotated values to `Floats`, performing unit-casts and the application of auto-derived unit-conversion expressions.

As mentioned in Section 3.3.2, unit conversion functions are expressed in a restricted sub-DSL to ensure safe conversion of values between differing units within a dimension. Within the backend the conversion expressions are converted into *Core* numerical expressions that are injected into the model code for each cast, resulting in a value with the correct unit. Such conversion expressions will largely be statically evaluated and optimised out during the code-generation stage, providing minimal-cost complex unit conversion.

5.1.4 Initial Value Generation

Initial values of a system are mixed with time-dependent model code in *Ode* and can contain complex expressions (see Section 3.1.6). To determine their value prior to starting a simulation we perform compile-time evaluation. This simplifies the backend code-generator as it may focus only on the model simulation loop, and importantly enables storing initial values as double-precision floating-point numbers directly within the LLVM assembly. Initial investigations demonstrated that this provides performance improvements, triggering further optimisations within the LLVM code-generator and enabling the values to be stored in an optimal fashion within the native-code.

Evaluation is performed by a compile-time interpreter over a *Core* subset that captures the simplified model abstract syntax tree (AST). This basic interpreter performs a single traverse over the model AST, evaluating all expressions and recording all initial values. This is a form of partial evaluation that allows for further optimisations during code-generation.

5.2 *CoreFlat* IR

5.2.1 Overview

CoreFlat is the name given to our lower-level executable/simulation IR derived from the conversion and simplification of the *Core* IR. It is a simple, low-level numerical IR designed for easy mapping onto code-generators and assembly languages for a variety of hardware platforms, e.g. GPUs. It is the final *Ode* IR used to represent a simulation-ready model, used to express the numerical computations that occur within the timestep of a simulation.

The IR is influenced by a program structure known as Administrative-Normal Form (ANF) [47, 48], itself a simplified form of continuation-passing style (CPS) [77]. In ANF, all nested compound-expressions and inputs, e.g. *if* conditions, function/operator arguments, are lifted and bound to *fresh*, immutable values — in our case this process is somewhat simplified due to the previous inlining of all component/function applications.

Several optimisations and transformations are performed using this IR to ensure that the model is amenable to efficient execution. The conversion occurs within the backend pipeline (see Fig. 5.2) from an already simplified model in a subset of the *Core* IR. The main data-types for the IR and evaluation semantics for eventual model simulation are provided in Appendix C.1.

5.2.2 Basic Syntax and Semantics

A *CoreFlat* model is a 3-element tuple, (E, I, S), where E represents the simulation-kernel, evaluated upon each simulation iteration. This can be considered to represent $y'_n = f(y_n, t_n)$ for an ODE-based model. I holds the (pre-evaluated) initial values $y(0)$ and thus represents the model state; and S contains the set of simulation operations in the model, which currently consists of ODE definitions, used to calculate y_{n+1}.

The simulation-kernel, E, represents a small numerical simulation language. It closely maps the underlying hardware primitives and is used to express the immutable computations that occur per simulation timestep. The datatype is provided in Appendix C.1.1 and contains:

- Atomic, immutable values that may be created and referenced from an infinite set of unique identifiers. They may consist of (double-precision) numbers, booleans, the unit/void value, or the *time* term;

- A selection of built-in operators to be used with values, these are the same operators previously defined in the *Ode* language (see Appendix A.2);
- *If* -expressions to facilitate conditional control flow;
- Tuples to provide heterogeneous aggregate structures, alongside built-in operators to insert and extract values from them.

The values in I represent the current state of the system and are accessible in a read-only manner within the simulation-kernel E. They may be updated only by simulation operations defined in S, these are executed sequentially following an iteration of the simulation kernel for the current time. During simulation we initialise the environment with the current state value; evaluate the simulation-kernel for a timestep; and finally execute all simulation operations with respect to the evaluated state of the simulation kernel, implicitly updating the model state values in I.

This separation ensures that state(-values) may only be modified by simulation operations. This eases implementation of the simulation kernel and may aid creation of optimisations. Finally it makes extending the DSL with new simulation operators much simpler (as demonstrated in Chapter 6).

Values within *CoreFlat* are typed via an extremely simple type system that maps over basic CPU primitives. Semantics for the conversion stage of the backend pipeline from the *Core* IR subset into *CoreFlat* simulation IR are presented in Appendix C.1.2. These are followed by evaluation semantics for the *CoreFlat* IR model in Appendix C.1.3. However we mainly concern ourselves with the operational semantics observed with our reference interpreter.

5.2.3 Interpreter

An interpreter was created in parallel with the language definition to solidify the core concepts and determine the simulation semantics, as described in Appendix C.1.3. It also acts as a reference system to determine simulation correctness, particularly with respect to the JIT compiler.

The interpreter performs a recursive descent pass of the model *CoreFlat* AST during each simulation iteration/timestep. It exhibits strict, call-by-value evaluation semantics. Performance is not a priority for the reference interpreter, and as such we implement only basic solvers for the possible simulation operations as described in Appendix B.

5.3 Specialised Code-Generation

The primary simulation implementation is a native-code compiler with just-in-time (JIT), ahead-of-time (AOT) and external library modes (as depicted in Fig. A.1) controlled by the Haskell-based *Ode* system. The compiler operates upon a model in the *CoreFlat* IR according to the evaluation semantics in Appendix C.1 and as implemented by the interpreter.

The *Ode* backend utilises the LLVM compiler framework (see Appendix C.2.1) to perform native code generation and execution [87]. We perform compile-time generation of model and simulation code using a typed intermediate assembly format termed LLVM bitcode [86]. The LLVM code generator makes use of the highly static nature of the generated bitcode and performs extensive optimisations to pre-process and remove extraneous run-time computations.

The system generates the model, solver code and simulation parameters into a combined *model-simulation* module, this process is illustrated in Fig. 5.4. As mentioned, the *CoreFlat* IR representing the model simulation-kernel is based on ANF, this is similar to the static single assignment (SSA) form used by LLVM [6] and simplifies the code-generation process described in Appendix C.2. This code-generation approach results in an extremely efficient, specialised, simulation-kernel to perform the numerical computations required at each timestep. The implementation builds a custom solver during compilation derived from the simulation operations in the *Ode* model, i.e. ODEs (and SDEs and SSA-reactions as described in Chapter 6). Several numerical ODE solvers have been implemented, these are described in Appendix B.1.1 and include a forward-Euler, a 4^{th}-Order Runge-Kutta and an adaptive solver. Adaptive simulation is supported by compiling the *Ode* model to native object-code and linking this to the CVODE adaptive solver library.

Having generated a custom model-simulation system, control may be passed to the generated code through a fixed entry-point that starts the simulation. This separation of concerns allows the high-level implementation to manage and optimise the simulation whilst the low-level custom code performs efficient numerical computation and simulation.

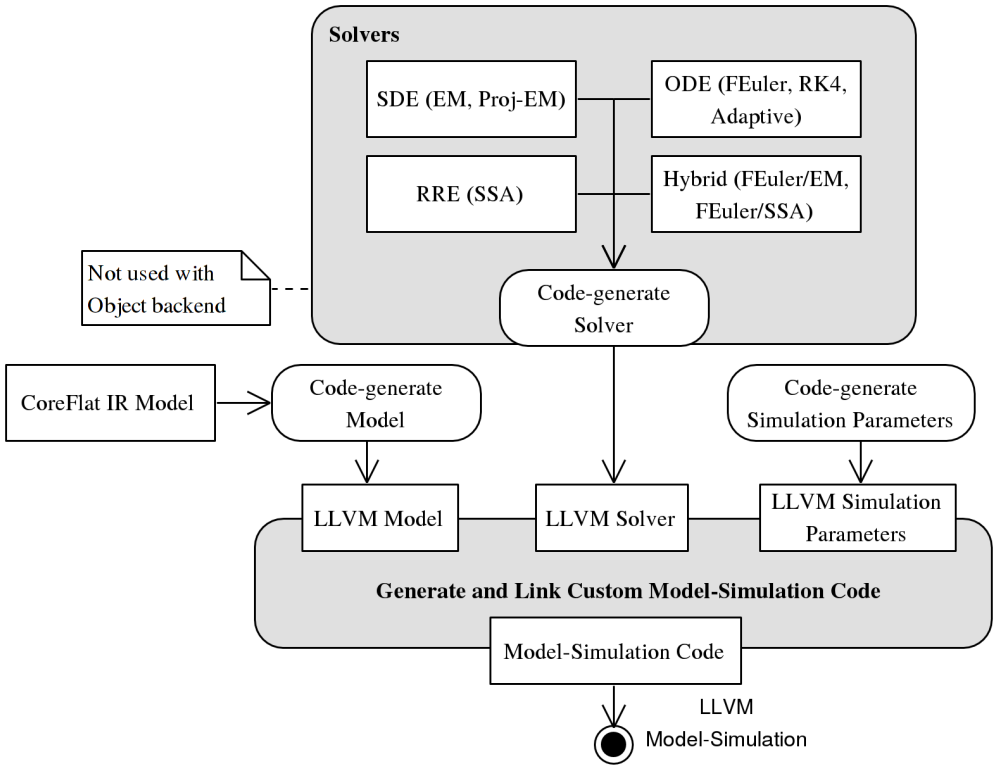


Figure 5.4: Generation of the model-simulation module. This comprises several components; the simulation kernel from the *CoreFlat* IR that represents the translated model code; custom-generated solver code generated from the simulation operations in the model; and finally state variables and configured simulation parameters, e.g. t_{start}.

5.3.1 Codegen Process Overview

During compilation the system generates a single LLVM module that represents both the model and custom simulation code with a well-defined single entry-point. This section and the complementing flowchart in Fig. 5.5 provide a high-level overview of the code-generation process that results in this module with three differing simulation modes for differing use-cases.

Just-In-Time (JIT) compilation is the primary simulation mode. The model-simulation code is generated in-memory and executed within the same process as the compiler. This should be used during model development as it provides efficient simulation directly from within the *Ode* console interface, thus facilitating rapid prototyping and feedback. An ahead-of-time (AOT) mode is also present, whereby the simulation is compiled to a standalone executable that may be shared and executed elsewhere. This is intended for use when performing multiple (potentially stochastic) simulations runs on several machines that do not need the *Ode* compiler installed; however it requires extra steps to create and manage the executable. Finally an object-code mode is provided, where the model only is compiled to a native-code library that can be linked and

utilised externally. This facilitates using the optimised model code with external high-performance solvers such as CVODE [27], or within multi-scale simulation environments such as Chaste [120]. A C-based foreign-function interface (FFI) has been defined to control the simulation from an external process, the API is described in Appendix C.4 and as previously mentioned is used by the adaptive solver within *Ode*. The combination of optimised model code with a high-performance adaptive solver will enable extremely efficient simulation, however requires extra development before simulation can occur.

When using the JIT or AOT mode, the model and custom solver and specified simulation parameters are generated within a single LLVM module. This combined model-simulation module is further linked with another module representing the *Ode* language run-time. This support library contains many pre-compiled helper functions required during simulation, such as file-output routines, and is described further in Appendix C.5.

This combined module is eventually compiled to native-code according to simulation-specific parameters regarding optimisation levels and mathematical accuracy (see Section 5.4). Further low-level details regarding the generation of LLVM-bitcode may be found in Appendix C.2.

5.3.2 Implementation Benchmarks

A goal of this DSL research is the creation of a high-performance, optimised, simulation implementation through the use of custom run-time code generation. In the following section we demonstrate the results of this approach through several simulation timing benchmarks, believing the implementation will enable rapid prototyping and efficient simulation that in turn will open new modelling applications. We expect simulations to be as fast as compilation through C, although greater performance is expected due to the restricted semantics of the language.

5.3.2.1 Methodology

Our strategy is to compare multiple parameters measured during simulation of the HH52 neuronal model and of several cardiac electrophysiological models of increasing complexity (see Section 2.1.3). We compare *Ode* models against similar variants developed and simulated in MATLAB, Python (using the NumPy package), and compiled C.

The initial model code in each alternate language was generated utilising the PCENV CellML modelling environment from curated representations obtained from the CellML online repository.

PCENV in turn utilises the CellML API[1] to generate the model code. The CellML API can generate model code in a variety of languages and is used by all current CellML environments for generating and running simulations. Thus, in addition to comparing *Ode* against C and other general-purpose languages, these benchmarks also act to compare the performance of models developed in *Ode* against those using the CellML DSL and contemporary simulation environments. Further benchmarks provide details on the performance characteristics of advanced *Ode* modelling features, discussed in Chapters 3 and 4, and the efficiency of the included ODE solvers.

The resulting generated model code in each language was modified to utilise the same output format to ease off-line analysis. The models were further altered to produce 10 stimuli over the course of the simulation and the timestep adjusted such that the native simulations would require ~1min of CPU simulation time, considered large enough to draw meaningful conclusions. Alternatively we could have left the timestep at a reasonable value and increased the length of the simulation. This would have required a similar number of iterations to necessitate ~1min of CPU time when using explicit fixed-step solvers, and the timestep was never small enough to be affected by FP precision issues, hence either approach sufficed.

Finally the PCENV-generated C code was manually optimised to be more efficient. This included linearising all model code to be located within a single function, and utilising `const`, `static`, and `restrict` declarations where appropriate to maximise the optimisation potential during compilation. We further modified the C code to utilise the *Ode* runtime support library, and complied it using the LLVM-based 'Clang' C compiler with the same optimisations as *Ode*[2]. By doing this we have attempted to maximise the performance achievable when compiling-via-C, as CellML does, such that the only difference between the C and *Ode* representations is the model and solver code generated in each case and thus valid comparisons can be made between them. Fig. 5.6 depicts a sample, taken from the TNNP04 model, of the AP traces generated during these simulations.

We have developed a benchmarking and analysis framework to perform simulations and automatically compare them according to several parameters. These include the simulation time, taken as the mean of CPU time over several simulations; the memory allocated by the simulation

[1] `http://cellml-api.sourceforge.net/`

[2] Only *safe* optimisations that preserved 'accurate' numerical behaviour were enabled, as described in the following section

Model	t_{start} (ms)	t_{stop} (ms)	h (ms)	Output Period (ms)
HH52	0	600	1×10^{-5}	0.12
N62	0	1000	2×10^{-4}	2
BR77	0	1000	5×10^{-4}	2
LRd94	0	1000	0.001	2
TNNP04	0	1000	0.001	2
ORd11	0	1000	0.002	2

Table 5.1: Simulation parameters used to generate performance results comparing *Ode* against multiple language implementations.

process (in Mb); the size of the executable binary code-segment size for native-code simulations (in bytes); and finally the numerical floating-point (FP) difference. This is the maximum absolute difference between calculated voltage and a reference result, denoted with an *, and is expressed in multiples of the hardware FP epsilon, ~2.22×10^{-16}. Fig. 5.6 visually depicts this FP difference in absolute terms, comparing the *Ode* and C simulations as was recorded during our simulations.

We are essentially using functional curation to compare and validate the models across differing implementations, using the FP difference to ensure that the models, and simulation systems, are correct and that we are obtaining repeatable results. As seen, the FP difference increases during the AP, becoming largest during repolarisation, before decreasing during the resting period. However, even at its largest the difference is insignificant compared to the magnitude of the voltage. It is caused by small FP changes that can occur due to the slightly different selection and ordering of instructions in each backend. These measurements are used within this chapter and in the detailed results tables provided in Appendix C.3.

5.3.2.2 Results and Discussion

Implementation Performance

Table C.2 in the Appendix presents the full data obtained from simulating several cardiac models using *Ode* and several alternate implementations. In each language the simulations were performed using a forward-Euler ODE solver with the parameters listed in Table 5.1. The initial conditions for each model were unchanged from those specified in their respective original publications [9, 69, 94, 95, 106, 109, 133]. The results in Table C.2 demonstrate that it is possible to have a high-level DSL, more suited than MATLAB or Python for describing biological models, yet capable of generating accurate simulations that execute faster than optimised C code.

Fig. 5.7 uses this data to plot the performance differences between native *Ode* and C simulations. In all cases *Ode* generates more efficient simulation code than C, from ~2% in earlier models to ~15-20% faster in more complex recent models. We believe the restricted semantics of *Ode* compared to C enables the generation of simpler low-level code that is more amenable to basic compile-time optimisations with regards to more complex models.

Looking again at Table C.2, the memory usage of *Ode*-generated simulation is comparable to C. It remains under 1Mb, highly beneficial for use within multi-scale models. The executable segment is smaller, due to the static semantics of the simulation-kernel and DSL code-generation scheme, fitting within a typical CPU's instruction cache. The accuracy of *Ode* simulations is comparable with the C, Python, and MATLAB simulations, often differing by only a few thousand epsilons. This provides confidence in the correctness of the code generation process and simulation algorithms. As an aside, the *Ode* interpreter exhibits timing characteristics within the same order of magnitude as the far more optimised Python and MATLAB environments; further development can improve the performance of this reference implementation.

Ode Feature Benchmarks

Table 5.2 presents results from simulating the HH52 model using multiple DSL abstractions and features described in Chapters 3 and 4, with each subsequent model building upon the previous with higher-level abstractions. The static, parameterised, and unit-checked HH52 model variants used in these simulations are provided in Appendix D.4 for reference. The simulations were performed using the same parameters shown in Table 5.1 for the HH52 model with the initial conditions unchanged from the original publication [69], as were listed in Fig. 2.6.

We see that the simulation time is largely unaffected by the abstractions and features. Furthermore the results indicate that often the same binary code is generated, and most importantly, simulation accuracy is not affected. This stems from the careful introduction of features in Chapter 4 that are largely compile-time based, and as demonstrated, provide abstractions with no performance or accuracy penalties.

Solver Backend Benchmarks

Table 5.4 presents data obtained from simulating the complex TNNP04 model [133] using the multiple ODE solvers in *Ode* alongside C-based equivalents. The simulations were performed

Implementation	**Sim Time (s)**	**Binary Size (B)**	**FP Diff (eps)**
Unstructured, flat, model code *	55.8	4245	*NA*
Abstraction of ion channels into components	56.0	4245	0
Nesting of components to mimic biological structure	56.2	4293	0
Addition of unit-annotations	55.7	4245	0
Unit-conversion of membrane voltage, V	56.2	4245	*NA*[1]
Separate model into static modules at ion channel level	55.8	4245	0
Utilise parameterised modules to represent generic ion channels	55.7	4245	0

Table 5.2: Benchmark results for simulating variants of HH52 model in *Ode* using increasingly higher-level abstractions and patterns. The simulation times are comparable, with *Ode* often generating identical code that does not impact accuracy.

with the parameters listed in Table 5.3, using a forward-Euler, RK4, and CVODE-based adaptive ODE solver. The initial conditions were unchanged from the original publication [133]. As the solvers generate differing simulation results, we provide the FP difference using absolute values rather than epsilons.

The RK4 solver is used as a reference, and unsurprisingly the simulation time is approximately 4 times that of the forward-Euler (see Appendix B.1), as is the compiled binary size. We again note that the *Ode* forward-Euler (FE) is more efficient that its C equivalent, however both require ~100min to complete. Both adaptive solvers complete a simulation run in ~50s, with the *Ode* adaptive solver slightly outperforming its C-based equivalent, as expected given the previous results in this chapter. The results generated by the adaptive solvers are consistent with those from the constant step solvers.

[1]Membrane voltage is unit-converted to V from mV, hence differs by a factor of 1000.

Parameter	Value
t_{start} (ms)	0
t_{stop} (ms)	1×10^{6}
h (ms)	0.001
Output Period (ms)	1
h_{min} (ms)	1×10^{-6}
h_{max} (ms)	0.5
Max. Number Steps	1000
e_{rel}	1×10^{-6}
e_{abs}	1×10^{-7}
Model Type	Stiff

Table 5.3: Simulation parameters used to generate performance results comparing solver implementations in *Ode* and C. The adaptive solver parameters are shown under the dotted line.

Implementation	Sim Time (s)	FP Diff (abs)
Ode RK4 *	24175.0	*NA*
C FE	6929.0	0.201
Ode FE	5951.0	0.201
C Adaptive	51.9	0.217
Ode Adaptive	48.4	0.226

Table 5.4: Benchmark results from simulating the TNNP04 model in *Ode* and C using alternate ODE solvers. The adaptive solvers perform a simulation orders of magnitude faster than the constant-step solvers yet with comparable accuracy. In each case *Ode* is more efficient than C.

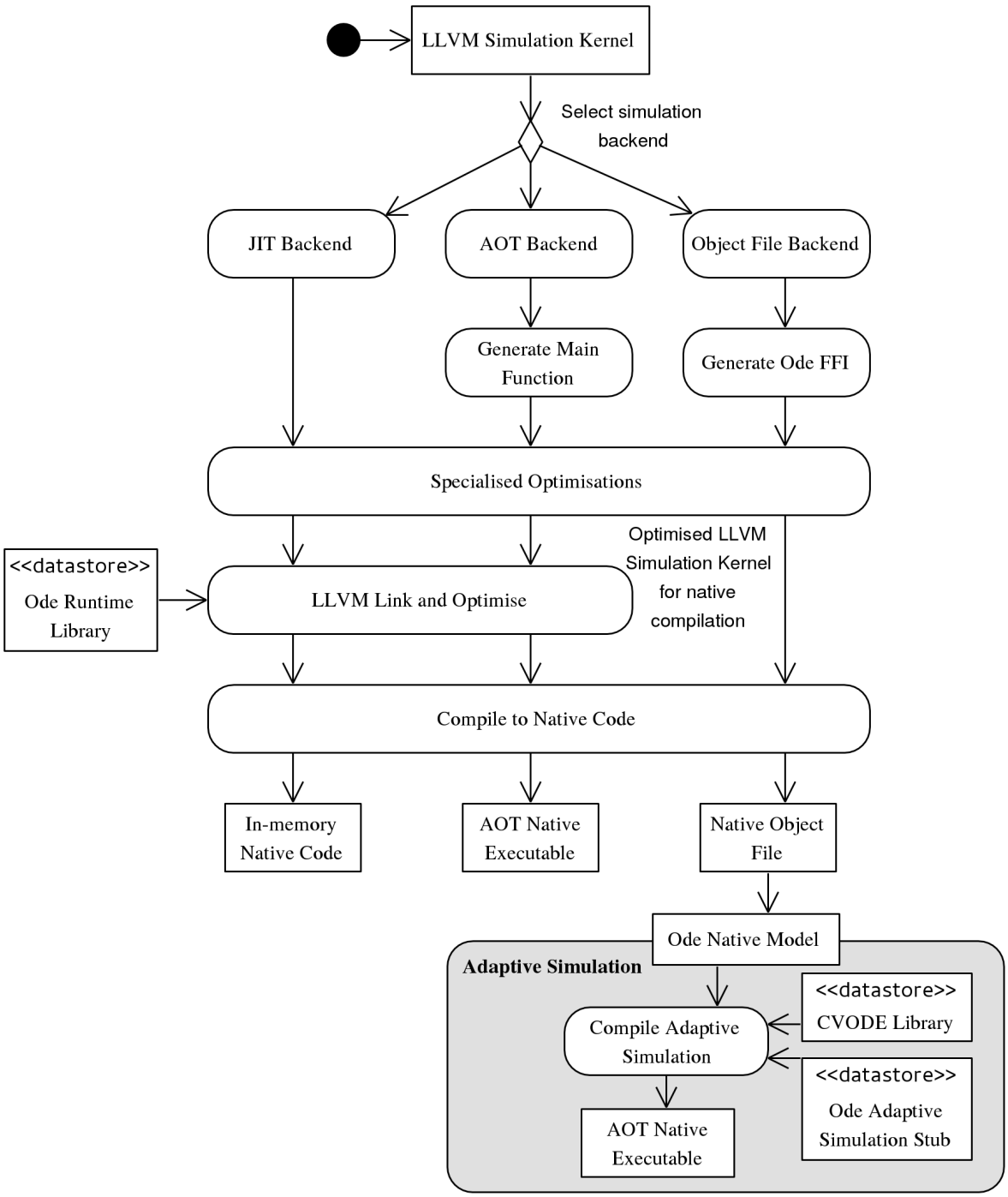


Figure 5.5: Illustration of the backend code-generation process from the LLVM model-simulation module to output and simulation formats. Three pathways exist depending on if JIT compilation, AOT compilation or an object-file for external integration is selected by the modeller. For each path differing optimisations and low-level processes apply, eventually resulting in a native-code representation of the *Ode* model. The object-file backend is also used to link the *Ode* model code to the CVODE adaptive solver.

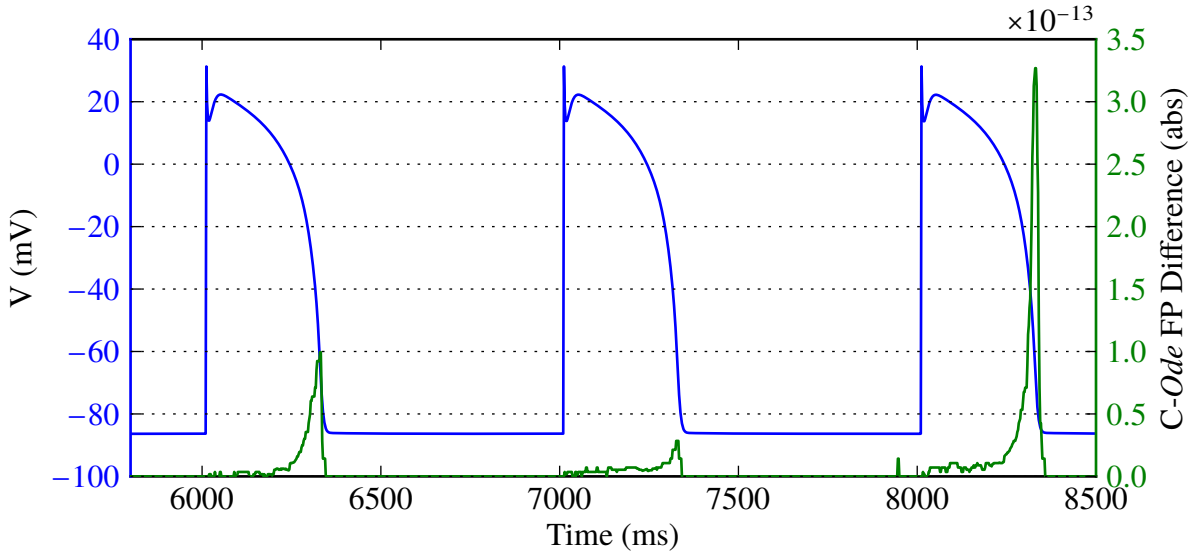


Figure 5.6: A typical sample of the AP curves generated as part of our benchmark simulations in *Ode*. The plot is taken from the TNNP04 model and includes the FP difference calculated from comparing the *Ode* and equivalent C simulation. The FP difference is insignificant compared to the magnitude of the voltage, suggesting that the simulations are comparable, and is caused by slight differences in low-level instructions used in each backend.

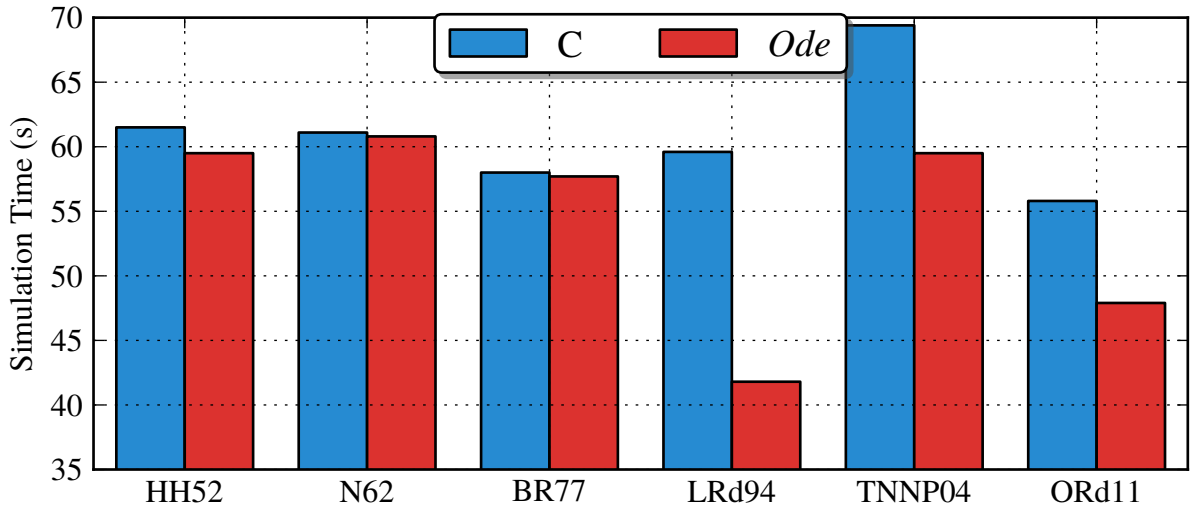


Figure 5.7: Plot comparing the efficiency of running simulations in *Ode* against those in C for several cardiac models. In each case *Ode* completes the simulation faster than C, with the gap widening for newer, more complex and larger models.

5.4 Low-level Optimisations

High-level model-specific optimisations and low-level, LLVM-based, assembly optimisations are performed within the *Ode* backend. The aim being to perform extensive pre-calculation upon the composite model and solver prior to code-generation.

We have already described several high-level optimisations and transformations performed during the compilation pipeline in Section 5.1, these include component inlining to remove function call overheads, and module flattening and specialisation. This section lists several of the lower-level model optimisations that take place at the *CoreFlat* and LLVM IR levels. Optimisations at the *CoreFlat* level are exclusive to *Ode*. However those at the lower LLVM level can be utilised by any language that uses this infrastructure, and are applied to both *Ode* and C models within the later benchmarks (with the exception of the *vecmath* optimisation).

5.4.1 Mathematical Operations

In general CPUs implement basic numerical operations, e.g. `add` and `mul` as primitive instructions. Whereas complex transcendental functions, e.g. `sin`, `log`, are instead implemented by system libraries such as `libm`[3]. On our target platform, x86-64 GNU/Linux, the GNU C library provides a general `libm` implementation with reasonable performance and accuracy. However, profiling several *Ode*-generated simulations indicated that over 80% of CPU time was spent computing transcendental functions within models. Hence support is provided for using custom, optimised, `libm` implementations that utilise specific CPU features far more efficiently[4].

Floating-point (FP) calculations form the bulk of computational mathematical models, however they can be fraught with difficulties and little-understood corner cases. These are often ignored by modellers, particularly when generating *ad hoc* simulation code, however such errors can drastically alter results [62]. Double-precision FP values have a fixed-width that only approximate the computed numbers; a loss of precision can easily accumulate as computation results are chained together. A benefit of DSLs is control of the resulting implementation, and as such we are careful not to introduce inaccuracies within our simulation code. For instance we calculate the current simulation time on each iteration rather than accumulating the value. When

[3]The functionality provided by `libm` is documented at `http://www.gnu.org/software/libc/manual/html_node/Mathematics.html`.

[4]For instance, the Intel Math library`http://software.intel.com/en-us/intel-mkl`.

running FP simulations we must ensure that the results are accurate to within a narrow tolerance. However math functions are only accurate up-to a certain value, defined as the *units in the last place* (ULP) [62], representing a balance between accuracy and performance.

Within our domain we would often like to sacrifice accuracy for performance. Our models are based on experimentally-obtained real-world data, where errors can be large, yet we can bound our implementation errors. Knowing that they are comparatively small, the implementation errors will not make a qualitative nor quantitative difference, and will be irrelevant in terms of interpretation of simulation results. As such the system implements two math modes, termed 'strict-math' and 'fast-math' depending on the FP accuracy required. The 'fast-math' mode enables several optimisations, detailed in the following section, and configures the LLVM code-generator to generate fast-math code itself. It triggers *unsafe* optimisations that are mathematically sound but usually disallowed due to the nature of FP arithmetic, and disables several CPU flags that track FP signalling features, e.g. checking for the presence of NaN's. This is not a problem within our modelling domain as biological simulations are generally expected to be finite.

5.4.2 Constant Computations and Simplifications

The immutable nature of the model simulation-kernel enables many constant optimisations during compilation. For instance, calls to complex math library functions using constant values may be computed at compile-time and inlined by LLVM. We can replace the remaining complex math functions with simpler implementations that trade numerical accuracy for performance, or even replace them with pre-calculated look-up tables [29].

Our approach thus far has been to apply compile-time calculations and simplify computations where possible, and to ensure that any remaining run-time calculations are as efficient as possible. We simplify many computations within the model backend and within LLVM, e.g. $\log(\exp(x)) \rightarrow x$, $x^1 \rightarrow x$ and $x^{0.5} \rightarrow sqrt(x)$. The backend also fully expands and inlines power functions x^n where $n \leq 8$ is transformed into $\prod_1^n x$, accumulating a negligible numerical error.

5.4.3 LLVM Optimisations

Using LLVM frees DSLs from implementing low-level optimisations, such as constant evaluation and propagation, and common sub-expression elimination; essentially our *Ode* implementation

gains many advanced optimisations found in high-performance C compilers. In turn this greatly simplifies the code-generation process. For instance, new variables and static computations may be (and are) introduced by processes operating on the *CoreFlat* IR safe in the knowledge that they will be optimised away. Furthermore, the DSL semantics enable the *Ode* backend to pass hints regarding purity, code-flow, and lack of exceptions to LLVM that permit further low-level optimisations.

To obtain high-performance we must ensure the backend generates efficient, highly-static, LLVM bitcode that may be utilised optimally by the LLVM native code-generator (this transform is detailed in Appendix C.2.1). For instance, the backend generates 'unboxed' values that only utilise LLVM primitive types with explicit control structures. Additionally model-specific knowledge enables the backend to statically allocate all memory, utilise fixed rather than indirect references and primitive rather than aggregate or indirect values where possible. Appendix C.2.4 provides further details regarding model-simulation bitcode generation.

Finally, the system is structured to take advantage of LLVM's link-time optimisation (LTO); LTO is used to optimise the model-simulation code and *Ode* run-time across the compiled object boundaries. This results in the inlining of all simulation-related code within the single binary entry-point and the specialisation of the *Ode* run-time to the specific model size and parameters. Thus we obtain many benefits of whole-program optimisation, resulting in the generation of highly-specific simulation code for a particular model.

After model-specific and low-level optimisations, our final optimisation approach involves hardware-based techniques common to HPC for generating fast numerical kernels.

5.4.4 Model Vectorisation

Efficient simulation requires use of modern CPU features and techniques, as were described in Section 2.4.3. One such technique is instruction-level parallelism (ILP), where the single-instruction multiple-data (SIMD) vector units of a CPU are used to perform numerical operations in parallel. The restricted semantics of the simulation-kernel provides the code-generator with many opportunities to reschedule and potentially auto-vectorise mathematical operations.

This approach to vectorisation differs from current vectorising C-compilers that attempt to auto-vectorise loops over structure-of-array (SoA) style data [11]. Such techniques are inappro-

priate for non-array based loops and heterogeneous operations. Instead it is possible to generate a dependency graph for operations, reschedule instructions, and automatically pack data used in multiple operations of the same type into vectors. These operations may then be performed in parallel via the vector floating-point hardware of modern CPUs. Such an algorithm is presented in [85], and is already part-implemented within LLVM [45].

However the current LLVM implementation is limited to vectorising primitive values only, whereas investigations within our domain indicated that the majority of simulation time is spent within complex transcendental functions. Ideally we would like to move such computation from scalar FPUs and perform them in parallel on the SIMD units. Fortunately the alternate Intel and AMD math libraries, mentioned in Section 5.4.1, provide vector variants of the standard `libm` transcendental functions for 2- and 4-width double vector units. Traditionally these functions are manually called from within performance sensitive low-level, such as visualisation and high-level mathematical libraries, we would like automatically utilise them when performing simulations. Thus we modify and extend the LLVM implementation to operate upon all transcendental mathematical functions within our simulation-kernel utilising a more suitable heuristic that takes into account the restricted domain of our model code.

Algorithm 2 provides an overview of the optimisation using pseudo-code. For all instructions within each basic-block (BB) we search for suitable candidate pairings of calls to the same transcendental function. Conflicting, i.e. dependent, candidate-pairs are discarded, and from the final set of suitable-pairings we select the nearest pairs to pack into vectors. The scalar transcendental calls are replaced with a single vector equivalent, and the vectorised result is unpacked for use with the remaining scalar model code.

We term this configurable optimisation *vecmath*. The majority of research and development occurred at the LLVM level in C++; this implementation is maintained by the author as a set of out-of-tree patches to LLVM release 3.3. We believe this to be a novel contribution, as thus far basic-block vectorisation has only been used to vectorise primitive CPU operations such as `add` and `mul` [85]. It has not been applied at a higher level to investigate vectorisation on transcendental function calls in the large that may be redirected to vectorised math libraries. Vectorisation of transcendental function calls will have a large impact within the numerical code that comprises most mathematical biological models. We had determined that ~80% of CPU time

Algorithm 2 Pseudocode depicting the *vecmath* algorithm, where $addr(i)$ returns the instruction address, $defuse(i)$ returns the set of instructions dependent on an instruction, $def(i)$ returns the identifier given to an instruction, and $arg(i)$ returns the input to an instruction. It scans LLVM basic-blocks for suitable candidate pairs calling the same math function. The most suitable pairs, i.e. those that are independent and logically closest, are combined into a vectorised call.

Require: LLVM module
  for each *bb* basic block in module **do**
    ▷ find pairs of math calls to vectorise
    $suitablePairs \leftarrow \emptyset$
    $candPairs \leftarrow \{(i1, i2) \mid i1 \in bb \wedge i2 \in bb \wedge i1$ and $i2$ call same math function $f\}$
    ▷ filter independent candidate pairs
    for $(i1, i2) \leftarrow candPairs$ **do**
      if $(i2 \notin defuse(i1)) \wedge (addr(i2) > addr(i1)) \wedge (i1 \neq i2)$ **then**
        $suitablePairs \leftarrow suitablePairs + \{(i1, i2)\}$ ▷ inputs are independent
      end if
    end for
    $closestPairs \leftarrow \{(i1, i2) \in suitablePairs \mid$ closest pairs in $suitablePairs\}$
    ▷ update pair in *bb* with *vecmath* call to f'
    for $(i1, i2) \leftarrow closestPairs$ **do**
      $bb \leftarrow bb - i2$ ▷ remove $i2$
      $bb(i1) \leftarrow (def(i1), def(i2)) = f'(arg(i1), arg(i2))$ ▷ replace $i1$ to call f'
    end for
  end for

is spent on such calls and altered the algorithm heuristics to aggressively vectorise in response. Furthermore the algorithm is applicable to any complex numerical code outside this domain.

Adding vector support was comprised of several stages. They are indicated in Fig. 5.8 and include implementing the above algorithm; adding intrinsics to LLVM for the vector transcendental functions; implementing lifting and lowering functions to convert standard C-style `libm` calls into the vectorise-able intrinsics; and creating bitcode 'shim' functions that map the vectorised intrinsics to a vector-math implementation.

Listing 5.1 demonstrates using the *vecmath* algorithm to provide ILP for a simple *Ode* code fragment. The pseudo-code sample is annotated with comments that describe the vectorisation of a single `exp` function call from three potential candidates. We describe the output using pseudo-assembly, whilst the generated LLVM bitcode is provided in Appendix C.3.

5.4.5 Optimisation Benchmarks

This section investigates the effectiveness of our optimisations when simulating ODE-based cardiac electrophysiological models. High-performance simulation is required to enable new

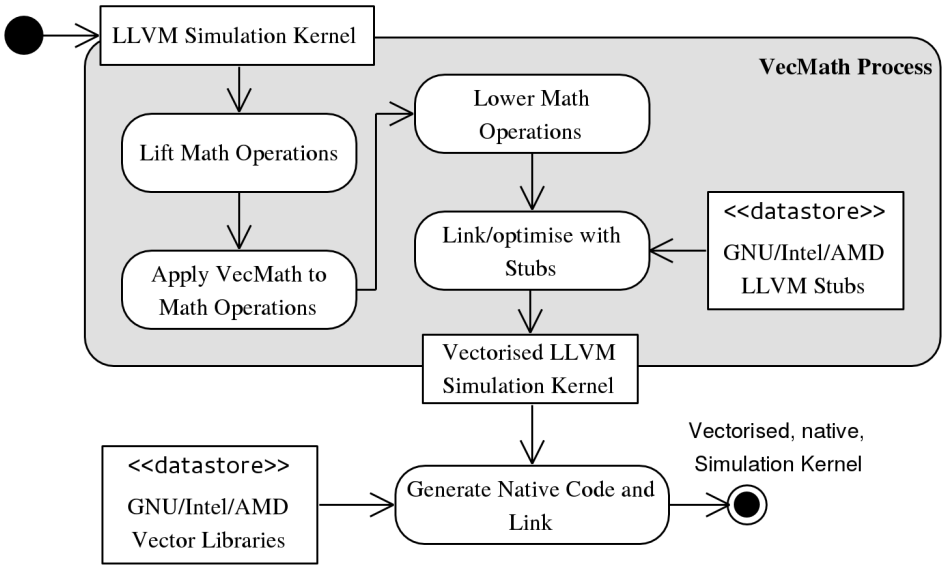


Figure 5.8: Backend processes involved in performing the *vecmath* optimisation. It requires converting all math function calls within the model into a vectorisation-suitable form, running the *vecmath* optimisation, and finally translating all vectorised calls into C-compatible representations for use with vector-capable math libraries.

modelling use-cases and facilitate simulation of ever-more complex models, this is reliant upon effective optimisations. We have already demonstrated faster than C performance whilst maintaining high-levels of structural abstraction in the previous section, and expect further gains.

5.4.5.1 Methodology

As with our previous simulations in Section 5.3.2, we investigate the effectiveness of each optimisation strategy through the simulation of the same models in *Ode*, comparing the results against unoptimised variants. We use the same *Ode* and C model code as in the previous section, and where possible compare each simulation to its equivalent in C using the same optimisations. For instance, all *CoreFlat* optimisations are exclusive to *Ode*; however low-level optimisations, both *safe* and *unsafe*, can and are applied to both *Ode* and C models at the LLVM level, allowing for valid comparisons to be made.

Having performed optimisation benchmarks, we apply the *vecmath* optimisation to the same set of models and run simulations. We test the algorithm on *Ode* models only as we believe it will be more effective on the optimised LLVM bitcode generated by the DSL backend. However the optimisation may be applied to any LLVM bitcode, such as that generated by C or even Haskell.

Listing 5.1 Application of *vecmath* to a sample *Ode* code fragment containing 3 calls to the `exp` function. The output from the optimisation is listed underneath using a pseudo-assembly syntax.

```
// initial code, assumes initial value "x"
val a = exp(x+1)  // dependent on "x", vectorisation candidate
val b = exp(a)    // dependent on "a", cannot be vectorised
val c = exp(x+2)  // dependent on "x", vectorisation candidate pair with "a"
val d = a + b + c // dependent on "a", "b", and "c"

// Generated Pseudo-code
tmp1 = x+1                      // temporary value
tmp2 = x+2                      // temporary value
vec1 = <tmp1, tmp2>             // build vector holding tmp inputs
vec2 = call vector exp(vec1)    // call the vector function,
                                //   calculation of "c" moved here
<a, c> = vec2                   // unpack the values from the vector
b = exp(a)                      // remaining unvectorised instruction
d = a + b + c                   // use calculated values as before
```

5.4.5.2 Results and Discussion

Optimisation Benchmarks

Table C.3 in the Appendix lists the full data obtained from simulating several cardiac models using the optimisations described in this section. The simulations were performed using a forward-Euler ODE solver with the same simulation parameters as the implementation performance benchmarks listed in Table 5.1. The initial conditions for each model were unchanged from those specified in their original publications [9, 69, 94, 95, 106, 109, 133].

The results indicate that *safe* optimisations make a large difference to the simulation performance, particularly on newer, complex models; they are more efficient than C whilst remaining as accurate as their unoptimised equivalent. These *safe* optimisation results are the same as the initial performance results shown in Section 5.3.2, and are provided simply to demonstrate the effectiveness of basic optimisations and to compare against further, *unsafe*, optimisations.

Fast maths optimisations result in even more efficient code, whilst altering the simulation results by only several thousand epsilons, an insignificant number. Constant computations and simplifications applied at the *CoreFlat* level enable even more efficient simulation in exchange for increasing the numerical difference from the unoptimised result; the extent of this difference varies between models however remains insignificant when compared to magnitude of the results themselves. This data is used to generate Fig. 5.9 that compares the most efficient *Ode* and C simulations across the models. It shows that the optimisations further widen *Ode*'s simulation efficiency over C, and in general is ~20-30% faster.

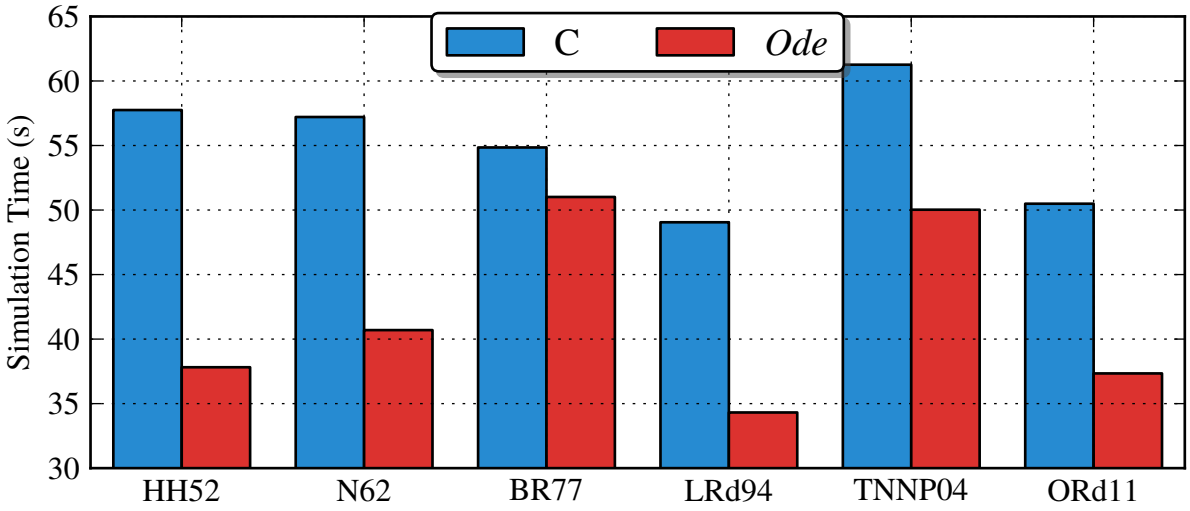


Figure 5.9: Plot comparing the efficiency of highly-optimised simulations in *Ode* against those in C for several cardiac models. In each case *Ode* is more efficient that C, with the optimisations further improving the results presented in Fig. 5.7.

Vecmath Algorithm Benchmarks

Table C.4 in the Appendix presents results obtained from applying the *vecmath* optimisations to several *Ode* cardiac models that are then simulated using multiple maths libraries on an Intel CPU with 2xdouble vector-width. This data is to generate Fig. 5.10, demonstrating the effectiveness of the *vecmath* algorithm in converting math functions into 2xdouble-width vectorised calls. The algorithm succeeds in vectorising most math functions, reducing the total number to ~55% of the original. Fig. 5.11 plots the simulation times of vectorised models using the Intel math library, where we see that vectorisation improves performance by ~30-40%. The simulations were performed using the same simulation parameters and initial conditions as in the optimisation benchmarks above. These results demonstrate the effectiveness of this specific optimisation on models in this domain.

A second set of results were obtained from simulation on a modern Intel CPU that supports both 2x and 4xdouble width vectors, these are presented in Table C.5 in the Appendix. We found that running the vecmath algorithm using 4xdouble vectors resulted in a larger reduction in math calls, reducing the total number to ~35%. Unfortunately this did not translate to a similar performance improvement, in many cases simulations are only 5-10% faster than those generated using 2xdouble-width vectors. We believe this is due to the immaturity of code generators supporting 4xdouble vector instructions, and further profiling is required.

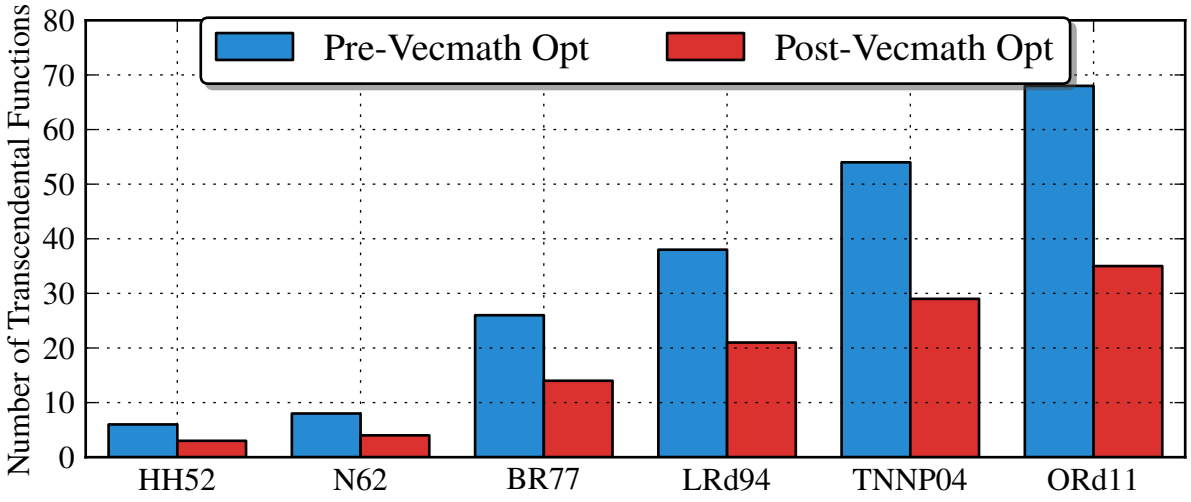


Figure 5.10: Plot demonstrating the effectiveness of the *vecmath* algorithm in vectorising transcendental functions in several cardiac models. The increasing complexity of models can be seen, with the *vecmath* optimisations succeeding in roughly halving the number of math calls.

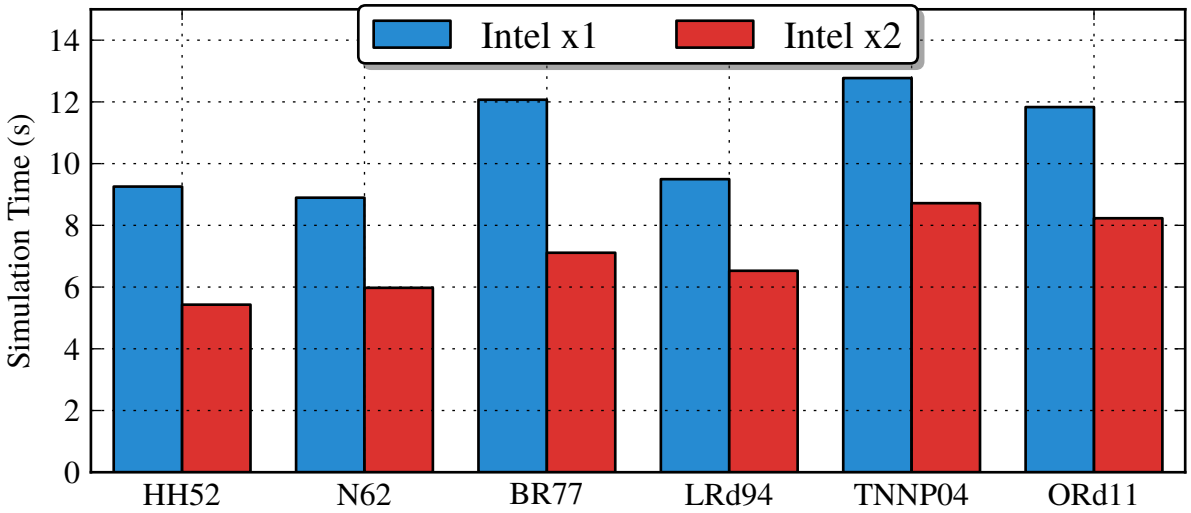


Figure 5.11: Plot demonstrating the efficiency of the *vecmath* algorithm using 2xdouble-width vectors against scalar code when simulating several cardiac models. A consistent and substantial performance improvement can be seen across the models, validating the optimisation.

5.5 Discussion

In this chapter we have presented a basic overview into the *Ode* implementation for high-performance simulation, with further details in Appendix C. The implementation includes a high-performance, optimised, native-code generation via JIT and AOT compilation and a reference interpreter. We presented the compilation process for an *Ode* model, including the *CoreFlat* backend IR, and stages that occur during transformation into the *CoreFlat* IR, then LLVM IR, and finally native code. The process includes many model-specific and low-level optimisations, often enabled by the DSL simulation semantics.

We detailed the native code-generation process from the *CoreFlat* IR utilising the LLVM code-generator and described the structure of a simulation-ready model. The various options and execution-modes for the code-generators were presented, including an object-file library for use with external simulators. We detailed our benchmarking process and provided simulation results for the native process using a variety of models against various simulation packages. The results were positive, indicating a ~5-20% improvement over low-level C simulations, for all solver types. The results were compared against the reference implementation using functional curation, and serve both as a demonstration of the correctness of the native implementation and as implementation-defined operational semantics. The results showed that the careful design of the high-level abstractions in *Ode* do not result in a performance penalty.

Finally we presented several optimisations implemented at a high model-specific level and lower-levels during code-generation. This includes constant optimisations, alongside a novel auto-vectorisation optimisation for the computationally-demanding transcendental functions present within the simulation kernel. We presented detailed benchmarking results for the system optimisations applied to several models against an unoptimised reference. Again the results are extremely encouraging, resulting in ~20-35% faster simulations over heavily-optimised C models.

Auto-vectorisation also provided large performance gains, enabling ~30-40% faster simulations compared with scalar simulations when performing many transcendental functions on 2xdouble width vector CPUs. These results demonstrated that performance may be dramatically improved without drastically altering or sacrificing accuracy or introducing further complexity, as occurs, for instance, with look-up tables. When all techniques are combined, we have

demonstrated up to ~45-60% faster simulations over low-level, optimised, scalar C code.

These encouraging performance results are due to the careful design of the *Ode* semantics and validate our initial decision to research and develop a new DSL rather than, for instance, extend CellML. As our C models are manually optimised versions of those generated by CellML, the same performance results and conclusions can be drawn and we may say that *Ode* enables simulations that are at least ~20-35% faster than CellML. The results also justify our decision to compile directly to native code, or rather the portable assembly provided by LLVM bitcode, instead of performing compilation-via-C. These techniques were discussed in Section 2.4.3, where we suggested that the intermediate language semantics may restrict performance. Alternatively, targeting a lower-level representation allows access to certain CPU features and the use of semantic metadata that aids LLVM optimisations and instruction selection. We believe this is responsible for the performance improvements over compiled-via-C CellML models.

Model verification, in particular, calculation verification, was demonstrated within the benchmarks through the use of functional curation to compare models against implementations in other languages/simulation systems. Further techniques from software engineering were demonstrated, such as model repositories that enable collaborative verification by multiple stakeholders, and simulation scripts to facilitate repeatable simulations and results.

This chapter described the DSL backend implementation, indicating how high-performance simulation may be provided from high-level *Ode* code. We have applied software engineering through the use of code generation to create optimised simulations that aid the computational modelling process. It allows modellers to focus on high-level abstractions and reusable model development, as discussed in Chapter 4, with the knowledge that their models can be simulated faster than C without requiring manual low-level code. High-performance simulation enables many new modelling scenarios and use-cases, facilitating an iterative development and simulation process, and making multiple and/or multi-scale simulations both efficient and feasible.

In the following chapter we extend the *Ode* DSL to support stochastic modelling and simulation through new *Ode* constructs and a new front-end DSL termed *Ion*. We extend the implementation from this chapter to include stochastic differential equations (SDE) solvers and a stochastic simulation algorithm (SSA) implementation (see Section 2.2), and demonstrate the highly-efficient simulation of discrete, continuous and hybrid stochastic models.

Chapter 6

Stochastic Modelling & Simulation

The previous chapters presented the research and implementation of the *Ode* DSL for its most common use case — modelling biological, specifically cardiac, systems in a continuous, deterministic manner. In this chapter we investigate DSL extensions to support stochastic modelling constructs based on biochemical reaction kinetics as discussed in Section 2.2.

Stochastic effects may impact the behaviour of biological systems, yet can be complex to model and efficiently simulate. For instance, *drug block* can reduce the population of an ion channel in a electrophysiological system such that the stochastic behaviour becomes significant. This may be modelled using a Markov-formulation that captures individual ion channel states and simulated through reference to biochemical reactions [67] (see Section 2.1.4). As demonstrated in Section 2.2, this may be simulated continuously and deterministically using ODEs, or stochastically, either from a continuous perspective using stochastic differential equations (SDEs), or discretely using the stochastic simulation algorithm (SSA). The chosen method will depend on simulation accuracy, computational efficiency and/or the desire to investigate the stochastic properties of the model.

These modelling methods are applicable to many biological systems governed by biochemical reactions, such as metabolic and regulatory pathways [103, 135]. As such, integrating stochastic support into our expressive and efficient modelling and simulation framework will provide use-cases and benefits outside of our domain of cardiac modelling [140].

We introduce SDEs in a similar manner to ODEs; an easier task having designed the core language and implementation. New evaluation and type rules are provided, with simulation performed using the Euler-Maruyama solvers [83]. SSA reactions are added to the DSL to

model discrete systems: this can be used to capture chemical reaction processes, or in our case the discrete conformational state changes of an ion channel. These will be simulated using an implementation of the direct-SSA [60]. We demonstrate these stochastic constructs with several examples and simulation benchmarks of a complex ion channel model. We believe our compilation approach will enable efficient SDE and SSA simulations, necessary as stochastic simulation is, often prohibitively, computationally intensive.

Finally we consider hybrid-stochastic modelling within the context of cardiac models, enabling multiple modelling constructs, i.e. ODEs, SDEs and SSAs that represent differing subsystems, to be placed within the same model and simulated. As an example, a Markov-formulation ion channel model may be placed within an integrative cardiac model, where the main voltage is simulated in a deterministic and continuous manner, whilst the individual ion channels are simulated using stochastic, discrete approximations. However the interaction of discrete and continuous elements within a model will require complex solvers that will independently simulate and update the elements as required [111]. Fig. 6.1 outlines the processes involved in compiling a stochastic *Ode* model into native simulation code.

We introduce a further biological modelling DSL, termed *Ion*, for declaratively describing individual ion channels for use within cardiac models; again demonstrating how DSLs may provide functionality to ease and regulate the process of model building. *Ion* is implemented as a front-end DSL that compiles to an *Ode* module; thus the previous software abstractions and techniques for reusable, high-level modelling discussed in Chapter 4 may still be utilised. This allows modellers to rapidly create new ion channel models, place them within an existing cardiac model, and utilise the new stochastic constructs trivially as opposed to performing lengthy and error-prone manual integration.

To demonstrate the stochastic implementations and the *Ion* DSL functionality we perform a small simulation study of stochastic electrophysiological models. This is based upon the Hodgkin-Huxley giant-squid axon model (HH52) containing a Markov-formulation potassium (K) channel. We describe the K channel using the *Ion* DSL and generate multiple ODE, SDE and SSA based *Ode* representations. These differing representations are integrated into the hybrid HH52-model and simulated.

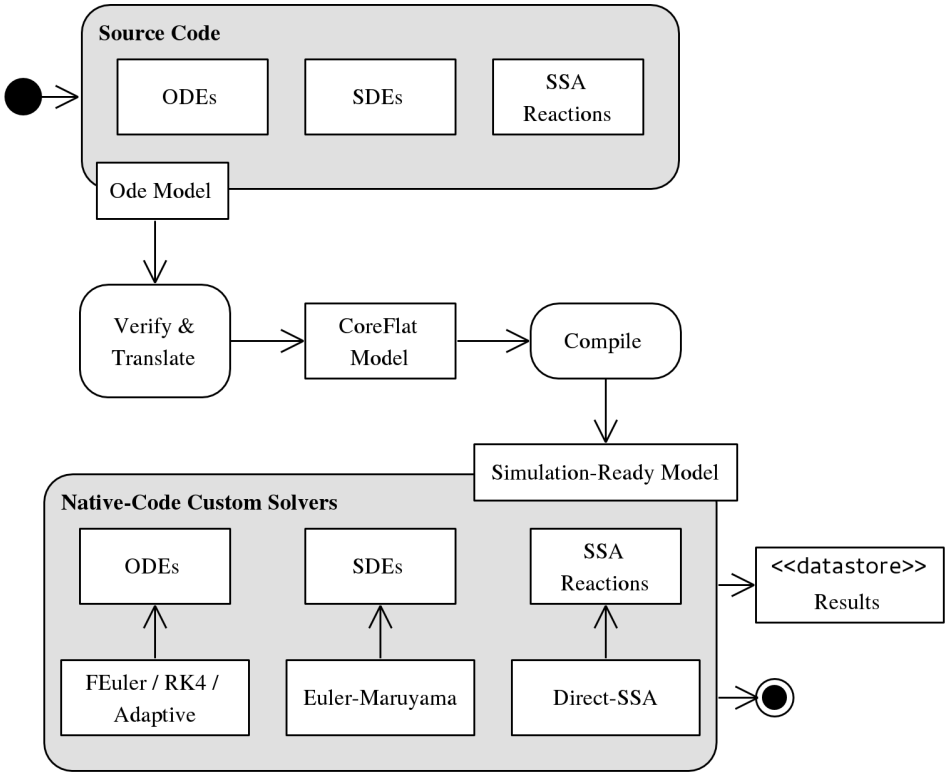


Figure 6.1: Conversion of an *Ode* model using ODEs, SDEs, and SSA-reactions into a native code simulation. The *Ode* source is translated into a low-level *CoreFlat* IR model (see Chapter 5) that is compiled using several classes of solvers depending on the model characteristics. The native-code is executed to generate a individual stochastic simulation run.

6.1 Stochastic Modelling

This section describes the changes to the *Ode* language and implementation necessary to support stochastic modelling. We introduce additional simulation mechanisms that support continuous stochastic modelling via SDEs, discrete stochastic modelling via the SSA, and hybrid-stochastic modelling of ODEs with either SDEs or the SSA. We use them in particular to model stochastically subcomponents of cardiac electrophysiological models. However they are applicable to a wide array of biological models (see Section 2.2).

We introduce the syntax and semantics of these constructs, describe their integration with DSL modelling and language features, and present the IR and backend implementation changes required for simulation. Highly efficient simulation is essential, first as discrete simulations are computationally very demanding, and second as stochastic models require many simulation runs to construct an ensemble of trajectories for meaningful analysis (see Section 2.2.2).

The stochastic constructs do not alter the core *Ode* semantics, i.e. the creation of an immutable simulation kernel representing time-based computations used to modify the system state. SDEs and SSAs join ODEs as 'simulation operations' that may modify the system state according to their respective simulation mechanism. This clear separation simplifies adding new simulation constructs to the language without drastically altering the core semantics.

Simulation benchmarks are presented for the SDE and SSA constructs. Our test case is a complex 17-state Markov-formulation *Ks* channel from the Decker canine model [38, 39], simulated over a range of input voltages. These simulations are compared against highly-optimised C implementations to determine the effectiveness of our model-specific code-generation approach from highly abstracted *Ode* models.

6.1.1 Continuous Stochastic Modelling

We model continuous stochastic processes using SDEs, as described in Section 2.2.4. SDEs are modelled in Itō form within the *Ode* frontend, and implemented using an explicit solver within the backend. This formulation describes SDEs using drift and diffusion expressions, with the diffusion expression containing a Wiener noise term Δw (see Appendix B.2).

Listing 6.1 SDE usage in *Ode*. This example SDE system models the 3-state reaction system from Section 2.2.4.1. The code structure follows the ODE tutorial provided in Appendix D.1. We use 3 `init` values, `X1`, `X2`, and `X3`, to represent the channel states. Several `vals` are created to hold computation results. `w1` and `w2` are `wiener` terms, representing realisations from $N(0,h)$ on every timestep. Finally SDEs are defined for each channel state; each SDE is associated with a diffusion and drift expression.

```
component getCurrent(V, a1, b1, a2, b2) {
    /* Setup initial values */
    init X1 = 1.0
    init X2 = 0.0
    init X3 = 0.0
    /* propensity computations */
    val n1 = 1/sqrt(1000)
    val prop1 = sqrt(abs(X1*a1 + X2*b1))
    val prop2 = sqrt(abs(X2*a2 + X3*b2))
    /* Setup state transitions using SDEs */
    val w1 = wiener
    val w2 = wiener
    sde {initVal : X1, diffusion : n1 * (-prop1*w1)} = -a1*X1 + b1*X2
    sde {initVal : X2, diffusion : n1 * (prop1*w1 + -prop2*w2)}
        = -(a2+b1)*X2 + b2*X3 + a1*X1
    sde {initVal : X3, diffusion : n1 * (prop2*w2)} = -b2*X3 + a2*X2
    /* Calculate channel current */
    val current = 36.0*(X3)*(V-87.0)
    return current
}
```

6.1.1.1 Syntax & Semantics

Listing 6.1 provides an example of the *Ode* SDE syntax to capture the SDE representation of the 3-state channel model provided in Section 2.2. The syntax additions consist of a `wiener` noise term and an SDE construct.

We add Wiener values as explicit terms that represent a fresh realisation from the normal distribution $N(0,h)$ generated upon each simulation timestep, these can be seen as `w1` and `w2` in Listing 6.1. We deliberately make the creation of Wiener terms explicit, rather than create a Wiener term implicitly for each SDE construct, as the same realisation can/must be reused for certain classes of models. For instance, when creating a stochastic-continuous representation of a Markov-formulation ion channel model (see Section 2.2.4) or when optimising biochemical reaction systems [99]. However, making Weiner terms explicit within the DSL makes the modeller responsible for their correct usage, as the diffusion term within each SDE in the system must be the product of a Wiener expression. Eventually we would like to enforce this constraint statically within the type system.

The SDE constructs shown in Listing 6.1 for `X1`, `X2`, and `X3`, are similar to those used to

$$
\begin{aligned}
\mathit{exprStmt} &\leftarrow \mathit{componentDef} \mid \mathit{valueDef} \mid \mathit{odeDef} \mid \mathit{sdeDef} \\
t &\leftarrow (E) \mid \mathit{num}\ [\mathit{unitCast}] \mid \ldots \mid \mathit{wiener} \\
\mathit{sdeDef} &\leftarrow \texttt{sde}\ \{\texttt{initVal} : \mathit{id},\ [\texttt{deltaVal} : \mathit{id},]\ \texttt{diffusion} : E\}\ = E
\end{aligned}
$$

Figure 6.2: *Ode* language grammar extensions to support SDEs, as in Fig. 3.2.

represent an ODE, requiring an initial value to reference and an optional identifier to hold the delta expression y'. The `diffusion` expression is placed within the `sde` element; this may be any arbitrary numerical expression deriving from a Wiener term. The `drift` expression is located to the right of the equals sign, representing the deterministic part of the SDE and is similar to the ODE delta expression. Fig. 6.2 provides the grammar for SDEs in *Ode*.

6.1.1.2 Implementation Details

Frontend

These constructs are added to the *Core* and *CoreFlat* IRs, and the conversion semantics from *Ode* to *Core* to *CoreFlat* and finally simulation are provided in Appendices A.3 and C.1. The stochastic constructs were added without drastically altering the core modelling semantics, utilising conversion rules similar to those already present for ODEs.

We introduced several new type and units rules concerning SDEs, illustrated in Appendix A.4. Type rules for the SDE and Wiener constructs are quite simple and closely follow those defined for ODEs. As mentioned the type system does not currently enforce the presence of a Wiener term within the diffusion expression. Units rules are also provided, however it is not possible to successfully units-check the Wiener expression. This expression contains the square-root of the timestep unit (typically *ms*), but our implementation does not support non-integer dimensions currently.

Backend

To implement the SDE constructs we alter the backend implementation, including the reference interpreter and the LLVM-based native-code compiler. For Wiener terms, we re-calculate and realise new random values upon each invocation of the simulation-kernel per timestep. This makes simulation-kernel non-deterministic between each interval, however the computed values remain

immutable within each invocation. Hence we can think of the kernel as being parameterised by the state of a random-number generator (RNG) r, in addition to time and the current model state, i.e. representing $(y', r') = f(t, y, r)$ for ODEs, where r' is the next, new state of the RNG after a timestep[1]. The *Ode* runtime-library includes fast random number functions for this purpose (see Appendix C.5). The library utilises the WELL512 RNG algorithm [112] to generate uniformly distributed values, these are then used to generate normally-distributed values through the Box-Muller transform [13].

The SDE constructs within a model are converted into simulation operations within the *CoreFlat* representation, as with ODEs these are processed following evaluation of the simulation kernel for each time interval (see Section 5.2). The calculated drift and diffusion values modify the referenced initial value and update the state of the system according to the SDE solver characteristics.

We implemented the Euler-Maruyama (EM) [83] scheme to solve SDEs as the current *Ode* simulation strategy is explicitly based upon calculating the value at the next timestep. The EM is an explicit, constant timestep solver derived from the forward-Euler and can thus be implemented in a straightforward manner; it is detailed in Appendix B.2.

We have implemented this solver within the native-code backend, again generating custom, model-specific, LLVM bitcode to simulate each model. This provides efficient, optimised, stochastic simulation with the same advantages demonstrated in Chapter 5.

6.1.1.3 Benchmarks and Discussion

We ran several benchmarks to test the performance and correctness of the SDE implementation within *Ode* using the Euler-Maruyama solver alongside a deterministic ODE representation using the forward-Euler for reference. As mentioned, the test model for our benchmarks was a voltage-clamped Markov-formulation implementation of the *Ks*-channel within the Decker model [39] containing 1000 channels. The method used to develop the *Ode* model is based on the SDE ion channel modelling formulation introduced in Section 2.2.4.1 for a sample 3-state channel system and presented in detail in [37, 99]. The model state diagram and *Ion* DSL representation (see Section 6.2) used to generate the model code is provided in Appendix D.5.2. The initial model code was altered to take the absolute of values when generating the diffusion coefficients

[1]Not to be confused with y' representing $\frac{dy}{dt}$.

t_{start} (ms)	t_{stop} (ms)	h (ms)	Output Period (ms)
0	3000	0.5 / 0.001	5

Table 6.1: Simulation parameters used to simulate the Decker I_{Ks} model at multiple voltages for the implementation benchmarks. The timestep h was decreased to 0.001ms for the performance benchmarks in Fig. 6.4.

to ensure that the channel propensities do not go negative and result in imaginary numbers, as described in [110]. The ODE reference model was created similarly using the formulation described in Section 2.2.5.1.

The model was voltage-clamped and simulated over several fixed voltages that cover the range seen during an AP: -87, -30, 0, and 30 mV. The simulations were performed with the parameters listed in Table 6.1 over 3000ms, with the initial conditions unchanged from the original publication [38, 39]. For each clamped voltage value 100 simulation runs were performed and the current, I_{Ks}, recorded. From these results we calculated the mean of the simulated I_{Ks} trajectories, this was possible as each stochastic simulation recorded the system state at the same time interval.

We proceeded to manually create the same model in low-level, optimised C, again using the Euler-Maruyama solver in order to benchmark the performance of the implementation. The C model was compiled with the same optimisations as *Ode*, and both model representations were simulated 5 times to calculate the mean performance. The simulations were performed using the same parameters listed in Table 6.1, although the timestep was reduced to increase the simulation time.

Fig. 6.3 depicts the results obtained from simulating the *Ks*-Channel model using SDE and ODE representations, with the SDE results closely matching their ODE equivalent. As the voltage increases in each successive plot, modelling AP depolarisation, the current flowing through the ion channel increases from random fluctuations about 0, i.e. closed, to larger values, modelling a higher proportion of channels reaching an open state. At the higher voltages the noise observed by the stochastic function of the 1000 channels in the model becomes less relevant and the results closely match the deterministic representation provided by the ODE solution.

Fig. 6.4 compares the mean performance of 5 SDE simulations within our implementation against an optimised C equivalent for the same fixed voltages. The ODE and SDE solvers use the same fixed timestep, yet the ODE simulations complete far more quickly than both SDE simulations. At the same time, both *Ode* and C SDE simulations exhibit similar performance,

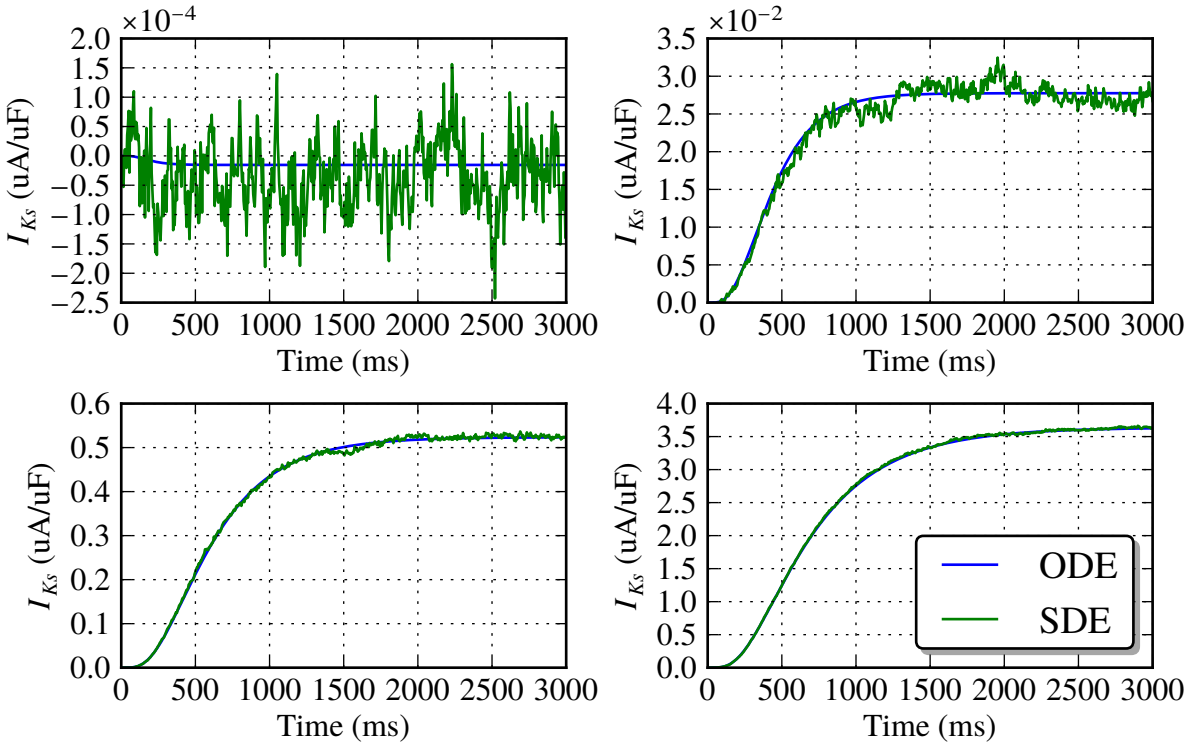


Figure 6.3: Plots generated from taking the mean of 100 stochastic trajectories generated by simulating the Decker I_{Ks} model at several voltages using SDEs within *Ode*. They are compared to a reference ODE simulation. Starting from the top-left in a clockwise order, V_m is clamped to -87, -30, 30, and 0 mV. At low voltages the channel is closed for the large part, with the stochastic behaviour of a small number of channels visible. At larger voltages the channel opens and the noise becomes comparatively smaller, tending towards the deterministic representation.

with *Ode* fractionally faster in each case. We can ascribe each of these results to the same causes. First, both SDE solvers rely on many complex FP functions, such as reciprocal roots, that are not required for the ODE solver. Second, the SDE solvers require and utilise the same source of normally-distributed random numbers[2], a complex computation that dominates the simulation process and effectively normalises any advantages the *Ode* code-generation process has over C. The results demonstrate that *Ode* simulates SDEs as efficiently as low-level C whilst providing direct support for SDE constructs and facilitating high-level modelling abstractions as described in Chapter 4.

6.1.2 Discrete Stochastic Modelling

We model discrete stochastic processes using reaction definitions that may be simulated using the stochastic simulation algorithm (SSA) [59]. The SSA itself was described in Section 2.2.3 and is a mechanism for obtaining an accurate simulation trajectory of a jump-Markov process,

[2]22 normally distributed random numbers are required per timestep for the Decker I_{Ks} model.

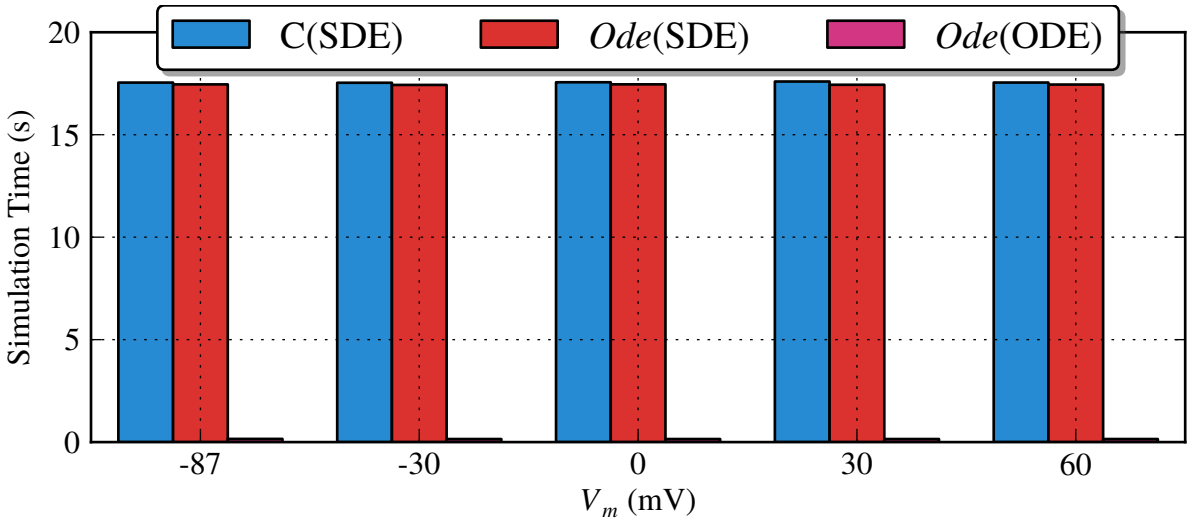


Figure 6.4: Comparison of the mean performance of a single SDE simulation using *Ode* and optimised C, alongside an ODE implementation. As expected, the *Ode* ODE implementation is far more efficient that both SDE implementations. The *Ode* SDE solver is slightly more efficient than its C equivalent.

such as a Markov-formulation ion channel system. We provide SSA-style reactions constructs within the *Ode* front-end and a high-performance SSA implementation within the backend.

6.1.2.1 Syntax & Semantics

Listing 6.2 provides an example of the *Ode* SSA-reaction syntax to capture the SSA representation of the 3-state channel model provided in Section 2.2. The syntax includes a new `reaction` construct to define an SSA-reaction.

The (chemical) reaction is defined on the RHS of the `reaction` construct; this uses a simple notation to easily define uni- and bi-molecular reactions. Each element within the reaction references an initial value and can be prefixed with an integer to represent the stoichiometric coefficient.

As seen in Listing 6.2, each reaction has a `rate` component specified within the element. Traditionally this is a static constant within SSA implementations, often for implementation simplicity and performance [59, 60]. However, by allowing the rate to be any numerical *Ode* expression the reaction propensities may be varied on a per-timestep basis. This is required for certain classes of models, such as ion channel models with voltage-dependent propensities, as seen with the HH52 K channel model in Section 2.2.3. Thus we can create complex, discrete, stochastic models without altering the backend simulation semantics. The grammar for SSA-reactions in *Ode* is provided in Fig. 6.5.

Listing 6.2 SSA usage in *Ode*. This example SSA system models the 3-state reaction system from Section 2.2.3.1. The model code structure, including initial values and computations, is similar to the SDE example in Listing 6.1. Pairs of SSA-reactions are defined to represent bi-directional reactions between channel states. The reaction `rate` may be any *Ode* expression.

```
component getCurrent(V, a1, b1, a2, b2) {
    /* Setup initial values */
    init X1 = 1000
    init X2 = 0
    init X3 = 0
    /* Setup state transitions using SSA reactions */
    reaction {rate : a1} = X1 -> X2
    reaction {rate : b1} = X2 -> X1
    reaction {rate : a2} = X2 -> X3
    reaction {rate : b2} = X3 -> X2
    /* Calculate channel current */
    val current = 36.0 * (X3/1000) * (V-87.0)
    return current
}
```

$$exprStmt \leftarrow componentDef \mid valueDef \mid odeDef \mid sdeDef \mid ssaDef$$
$$ssaDef \leftarrow \texttt{reaction}\ \{\ \texttt{rate}\ :\ E\ \}\ =\ ssaVal\ [+ssaVal]\ ->\ ssaVal_1 + \ldots + ssaVal_n$$
$$ssaVal \leftarrow [num\ .]\ id_{init}$$

Figure 6.5: *Ode* language grammar extensions to support SSA reactions, as in Fig. 3.2.

6.1.2.2 Implementation Details

Frontend

The `reaction` construct is added to the *Core* and *CoreFlat* IRs, and the conversion semantics from *Ode* to *Core* to *CoreFlat* and finally simulation are provided in Appendices A.3 and C.1. The construct was added without altering the core modelling semantics, utilising conversion rules similar to those already present for ODEs and SDEs. Again the immutable simulation-kernel remains, with SSAs joining SDEs and ODEs as a simulation operation that alters the model state via implementation of the direct-SSA.

We introduce several new type and unit rules concerning SSA reactions, illustrated in Appendix A.4. Type rules for the SSA construct are quite simple; we ensure that the rate expression evaluates to a float and use previously defined rules to ensure all referenced initial values are also floats. We do not introduce units to the reaction construct due to time limitations. We could utilise *mols* to annotate initial values when modelling chemical reactions, however the SSA is also used to model many other discrete stochastic systems, e.g. ion channel state changes in this

Chapter or regulatory networks. Thus, for now, we keep the construct dimensionless.

Backend

To implement the SSA construct we alter the *Ode* backend implementations, including the reference interpreter and the LLVM-based native-code compiler. We provide an implementation of the direct-SSA, requiring both a stochastically-chosen timestep and reaction per iteration (see Section 2.2.3).

`Reaction` constructs within a model are converted into simulation operations within the *CoreFlat* representation, these are processed following evaluation of the simulation-kernel upon each, stochastically-chosen, time interval. The triggered reaction is also stochastically-chosen and may reference values computed within the simulation kernel via the configurable and computed `rate` expression. The `initial` values that comprise the chosen reaction are updated according to the chosen reaction's type, be it uni-molecular or bi-molecular, the reaction stoichiometry, etc.

We implemented the direct-SSA using our existing explicit solver infrastructure, details of the algorithm are provided in Appendix B.3. The implementation is more complex than the continuous solvers and reuses the efficient random-number functions implemented for SDEs. It generates model-specific code to iterate over the reactions to determine and then trigger the chosen reaction. This is accomplished in an efficient manner via generated assembly code structured in a manner to reduce CPU branches. The impact of this programmatically-generated, model-specific, reaction selection code can be seen in the benchmarks in the following subsection.

We have implemented the solver within the native-code backend, again generating custom, model-specific, LLVM bitcode to simulate each model. This provides efficient, discrete stochastic simulation with the same advantages shown in Chapter 5 that may be used for high-performance stochastic ion channel simulation.

6.1.2.3 Benchmarks and Discussion

We performed several benchmarks to test the efficiency and correctness of the direct-SSA implementation within *Ode*, alongside a deterministic ODE representation using the forward-Euler for reference. We used the same test model as for our SDE benchmarks in Section 6.1.1.3, that is, a voltage-clamped Markov-formulation implementation of the Decker I_{Ks} model with a population of 1000 channels.

The SSA model was created from the state transition graph given in the original paper by Decker *et al.* [38, 39]. This is the same technique introduced in Section 2.2.3.1 to provide an SSA-based formulation for a sample 3-state channel system. The model state diagram and *Ion* DSL representation (see Section 6.2) used to generate the model code is provided in Appendix D.5.2.

The simulation methodology follows that used within the SDE benchmarks in Section 6.1.1.3. Firstly the model was voltage-clamped and simulated over several fixed voltages that cover the range seen during an AP: -87, -30, 0, and 30 mV. The simulations were performed with the same simulation parameters and initial conditions as those in Section 6.1.1.3. For each clamped voltage value 100 simulation runs were performed and the current, I_{Ks}, recorded. From these results we calculated the mean of the simulated I_{Ks} trajectories. As the SSA executes at discrete, stochastic, time intervals, this required taking the mean of the system state at the time of the first reaction after the desired output period. In this case the output period was every 5ms and due to the fast reactions in the system the next following reaction was never more than 0.05ms after. We do not consider this small difference to affect the results and conclusions within the context of our benchmarks.

We proceeded to manually create the same model in low-level, optimised C, again using an efficient implementation of the direct-SSA to benchmark the implementation performance. The C model was compiled with the same optimisations as *Ode*, and both model representations were simulated 5 times to calculate the mean performance. These performance simulations were performed using the same parameters and initial conditions as in the simulations above.

Fig. 6.6 illustrates the results obtained from simulating the *Ks*-Channel using the SSA and ODEs within *Ode*. The results are similar to the SDE-based simulations in Section 6.1.1.3, with the SSA matching the deterministic simulation. Again, as the voltage increases, the current flowing through the collection of channels increases, modelling the higher proportion of channels in an open state. Similarly, as the current increases, the stochastic behaviour becomes less relevant and the results become more similar to the ODE representation.

We note that the SSA-derived results appear to more closely follow the deterministic results at the resting voltage compared to the SDE equivalent in Fig. 6.3. As explained in Section 2.2.3 the SSA provides an accurate representation of the discrete channel changes. Whereas the Euler-Maruyama SDE solver is known to exhibit high variance [37, 110], and requires taking

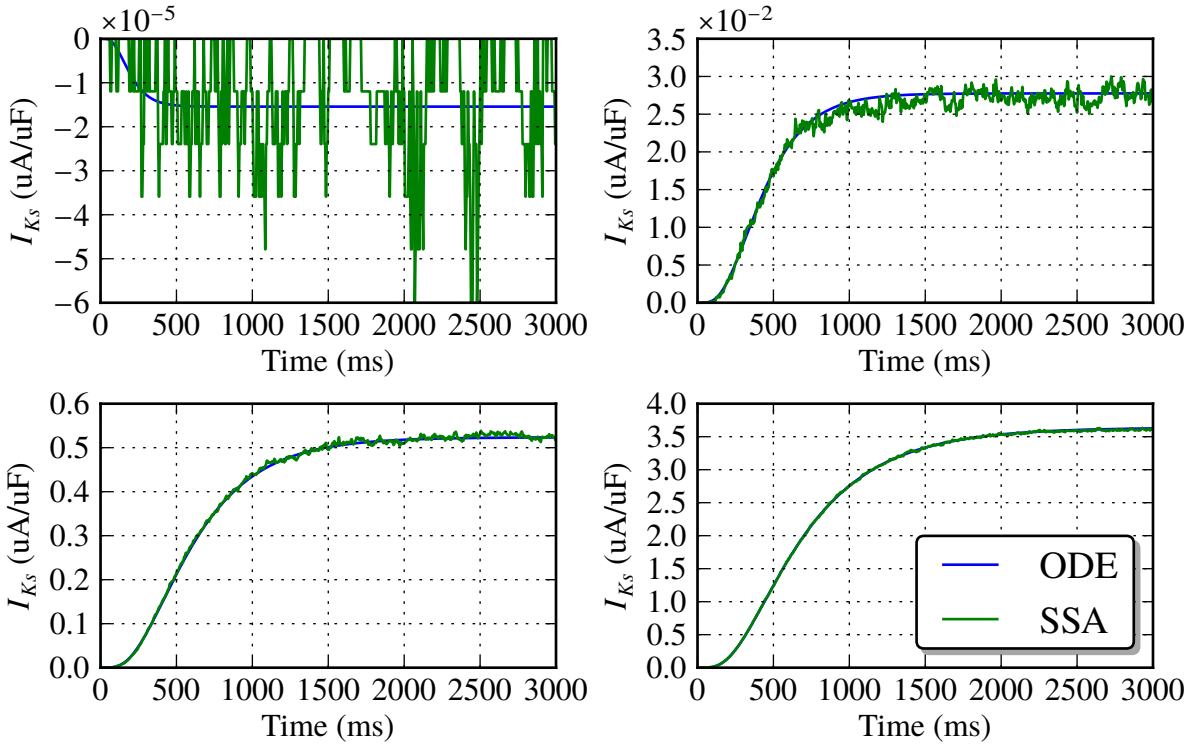


Figure 6.6: Plots generated from taking the mean of 100 discrete-time stochastic trajectories generated by simulating the Decker I_{Ks} model at several voltages using SSA-reactions within *Ode*. They are plotted against a reference continuous-time ODE simulation. Starting from the top-left in a clockwise order, V_m is clamped to -87, -30, 30, and 0 mV. The results are similar the continuous ones in Fig. 6.3, with the stochastic behaviour becoming less apparent at higher voltages as a larger population of channels are open.

the absolute value of rate calculations to avoid creating imaginary values. Furthermore, as can be noted when V_m is at the resting potential, I_{Ks} for the SDE trajectory goes positive, whereas the deterministic simulation and SDE trajectory stay negative. Alternate SDE solvers may be used that, for instance, reflect or project the state values at the boundaries, and exhibit several properties shown by the SSA and ODE results in this domain. We have started development of an alternate projecting-SDE solver in *Ode* with our collaborators for this purpose [37].

Fig. 6.7 compares the mean performance of 5 simulations within our SSA implementation against an optimised C equivalent for the same fixed voltages. It can be seen with both systems that as the voltage increases the time taken to complete a simulation drastically increases. This is simply because as the propensities are increased at higher voltages for the *open* states, more discrete reactions will be triggered at smaller timesteps. Comparing with the SDE results in Fig. 6.4, we see that the SSA performance is highly influenced by the reaction propensities (and, as shown in the following section, the number of channels), whereas, as expected, the SDE performance remains approximately constant across the voltages. Most importantly, it should be

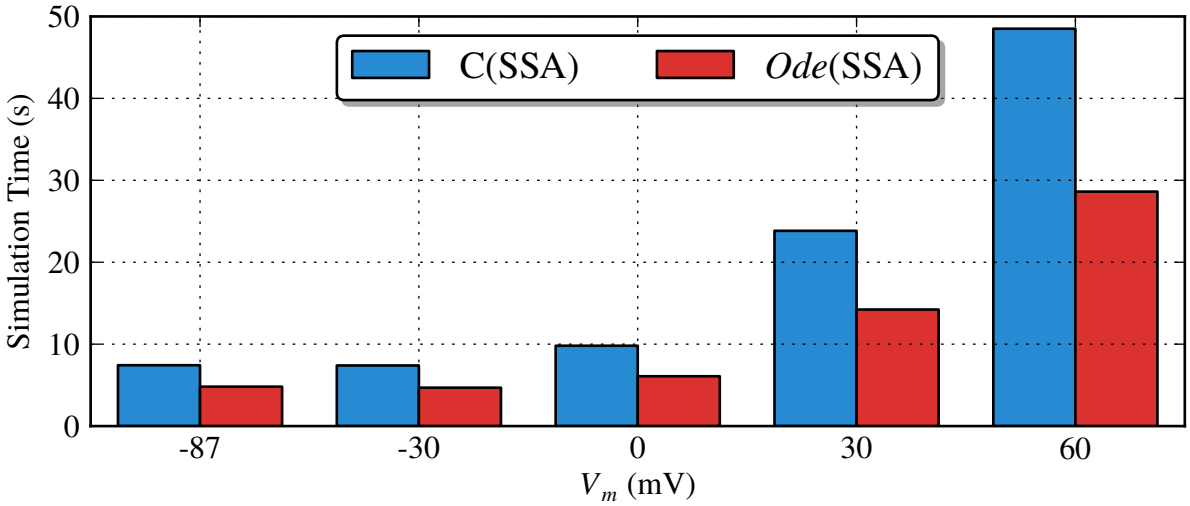


Figure 6.7: Comparison of the mean performance of a single direct-SSA simulation using *Ode* and optimised C. As the voltage increases the CPU time required to complete the simulation increases, due to a higher number of channels in the open state. For each simulation, the *Ode* implementation is faster than C, with the gap widening as the simulation becomes more complex.

noted that each *Ode* simulation completes significantly faster that its C equivalent, from between 35% to 40% faster with the performance gap widening in relation to simulation complexity. We believe this can be attributed to the low-level code used trigger reactions within the *Ode* SSA solver. As described in Section 6.1.2.2, we choose and trigger the next reaction through inlined code generated at compile-time specific to the system size, i.e. number of species. This is an extremely positive result as SSA performance is a limiting factor in its usage and much work has been undertaken in attempting to optimise the direct-SSA, as discussed in Section 2.2.3 and in [60, 91, 92].

6.1.3 Hybrid Stochastic Modelling

A hybrid approach is often required to simulate systems derived from multiple representations, such as cardiac cellular models containing Markov-formulation ion channels. In such systems, multiple approximations taken from chemical kinetic theory may be used to simulate a channel's state, from SSAs, SDEs and ODEs, as described in Section 2.2.6. Yet the membrane voltage of the integrative cell would be modelled as an ODE utilising currents derived from these channels, as detailed in Section 2.1.

These stochastic extensions to the *Ode* DSL represent our initial research within this area and currently allow only manual, modeller-defined, static partitioning of models for hybrid

simulation. Eventually we would like the system to automatically partition the reactions in an ion channel model dynamically as simulation proceeds. We implement two forms of hybrid modelling: continuous stochastic combined with continuous deterministic systems; and discrete stochastic combined with continuous deterministic systems. Furthermore all the stochastic elements within a mixed system are treated as *grouped*, comprising of a single logical system approximated by kinetic theory, i.e. the conformational states of a single ion channel. This is sufficient to allow modelling and simulation of a cardiac electrophysiological model comprising a continuous deterministic membrane voltage equation, multiple ODE-based ion channels, and a single stochastically modelled ion channel, using either SDEs or SSA-reactions.

Having manually partitioned the system into stochastic/deterministic and continuous/discrete approximations simulation can proceed. *Operator splitting* is used to control the hybrid simulation process [111]. Each scheme proceeds independently and updates its state with reference to the last known state of parameters from the other partitions (see Fig. 2.10 and Appendix B.4).

6.1.3.1 Continuous Stochastic & Continuous Deterministic Systems (SDEs & ODEs)

For simulating continuous hybrid-stochastic systems, e.g. a cardiac model using an ODE for the membrane voltage and SDEs for an ion channel, this process is simpler as both systems can use the same timestep. Fig. 6.8 depicts the simulation process, where we utilise an explicit forward-Euler for solving the ODEs and an explicit Euler-Maruyama for SDEs that may be integrated and solved simultaneously with fixed timestep. This is described further in Appendix B.4.1.

6.1.3.2 Discrete Stochastic & Continuous Deterministic Systems (SSA & ODEs)

There are certain issues to consider when simulating hybrid discrete and continuous time systems, e.g. a cardiac model using an ODE for the membrane voltage and the SSA for an ion channel, primarily concerning timesteps and state synchronisation. Our approach is based on work in [14, 15, 102] to simulate successfully several cardiac electrophysiological models.

Fig. 6.9 depicts this staggered simulation approach, where both discrete and continuous partitions are modelled independently. First, the stochastic state changes within the ion channel are modelled discretely, with a stochastic timestep that is much smaller than the timestep of the ODE used to simulate the cell membrane voltage. As such, discrete simulation may proceed up to the next, known, continuous time interval. This is controlled by the constant ODE timestep h. The

continuous system is then simulated to this interval, at which point the independent simulations are paused and the model states are combined and updated. The process is described further in Appendix B.4.2.

This hybrid simulation approach is straightforward and efficient; however the splitting of state may introduce errors. For instance, state changes caused by the discrete simulation will not be utilised when solving the continuous model until the next timestep, as discussed in Section 2.2.6. Another concern is the use of real numbers in the continuous system, as opposed to integers used in the discrete approaches to represent the same value. This was investigated with respect to cardiac models in [14, 102], where it was advised to apply nearest-rounding to represent whole channel populations derived from real channel proportions. Finally, as discrete events occur at stochastic time intervals it can be difficult to synchronise the state of the various continuous solvers. As a result we currently consider only fixed timestep solvers for the continuous subsystem and resynchronise once the discrete simulation passes the next scheduled continuous-subsystem time interval. An adaptive ODE solver, such as CVODE, could be used but would have to be configured to stop simulation at regular intervals in order to simulate the discrete subsystem.

6.1.3.3 Implementation Details

Implementing hybrid modelling required no changes to the *Ode* syntax and only peripheral changes to the model semantics; including verification of the hybrid model with regards to the initial values; and limiting stochastic elements to a single submodel within a hybrid system. The system infers the correct simulation method to use through the type and combination of simulation operations in the model. Hybrid simulation is restricted to certain solvers, i.e. the forward-Euler and the Euler-Maruyama for ODEs and SDEs respectively.

The bulk of the implementation work revolved around the creation of new solvers within the compiler backend. The semantics remain unchanged, with ODE/SDE/SSA simulation operations updating the state values with respect to the simulation kernel upon every timestep. The primary difference is there may now be multiple simulations underway utilising differing timesteps in nested cooperation.

We provided hybrid solvers for the native-code backend only, reusing and modifying the existing solvers implemented for ODEs, SDEs and the direct-SSA in Sections 5.3, 6.1.1 and 6.1.2. Hence the solvers utilise the same custom-code generation approach for simulating hybrid-

stochastic models — generating model-specific solver code to approximate the state change for the continuous deterministic subsystem using the forward-Euler and to trigger the next reaction for the discrete stochastic subsystem using the direct-SSA. Furthermore the solvers apply the same optimisations, at the *Ode* and LLVM levels, to expressions used to define ODEs/SDEs and to those used to calculate SSA reaction propensities.

6.1.3.4 Discussion

We have extended and implemented several forms of hybrid-stochastic simulation in *Ode*. Although limited in scope, they enable simulation of electrophysiological models containing stochastic elements. The hybrid-solvers also act to validate and form potential use-cases for the stochastic extensions to the core DSL described in Sections 6.1.1 and 6.1.2.

The hybrid-solvers were implemented within the DSL backend and have been tested with sample models. As they are based on our existing solvers they share the same potential for highly-efficient simulation through the use of custom, model-specific code-generation and optimisations within our native code compiler. The hybrid-stochastic solvers are used to simulate a cardiac model with a stochastic ion channel in Section 6.3; they demonstrate the flexibility of our DSL modelling approach and the strength of our high-performance simulation implementation.

6.1.4 Discussion

This section extended the *Ode* DSL from continuous deterministic modelling and simulation to stochastic systems; in particular modelling continuous stochastic systems via SDEs and discrete stochastic systems via the SSA. We introduced stochastic constructs to the *Ode* DSL frontend and described their syntax and semantics. We further updated our custom-code generation native-code backend to enable simulation of stochastic systems using well-known solvers.

This work was extended to allow hybrid-stochastic simulation, including support for continuous stochastic/deterministic systems and discrete stochastic/continuous deterministic systems. These particular hybrid forms allow simulation of integrative electrophysiological models with stochastic components. This was implemented by modifying the existing solvers within the native-code backend, allowing highly-efficient hybrid-stochastic simulation of cardiac models.

We provided several simulation results and benchmarks of the *Ode* SDE and SSA implementations using a 17-state, Markov-formulation channel model. The stochastic representations were

simulated over a range of biologically sensible voltages and compared against an ODE-based representation. In both cases the results were similar to the deterministic representation; we saw that as the voltage increased, the current increased due to the channel opening and the stochastic behaviour decreased. The performance results were encouraging, demonstrating a slight improvement for SDEs and a significant 40% increase for the direct-SSA over highly-optimised C-based counterparts. These results demonstrated that *Ode* can efficiently simulate stochastic models whilst providing software abstractions for model reuse and composition.

Developing complex stochastic models, particularly those derived from biochemical reactions, is a manual, error-prone process. This is particularly true when deriving stochastic/deterministic representations of an ion channel component, as we discovered during creation of the Decker I_{Ks} stochastic models. In the next section we introduce a DSL, termed *Ion*, to declaratively describe ion channels and easily generate such simulation-ready representations. This will be used within the subsequent sections to generate an cellular electrophysiological model containing a stochastic ion channel; providing an in-depth application of our hybrid-solvers and forming an investigation into the stochastic behaviour of ion channels within integrative models.

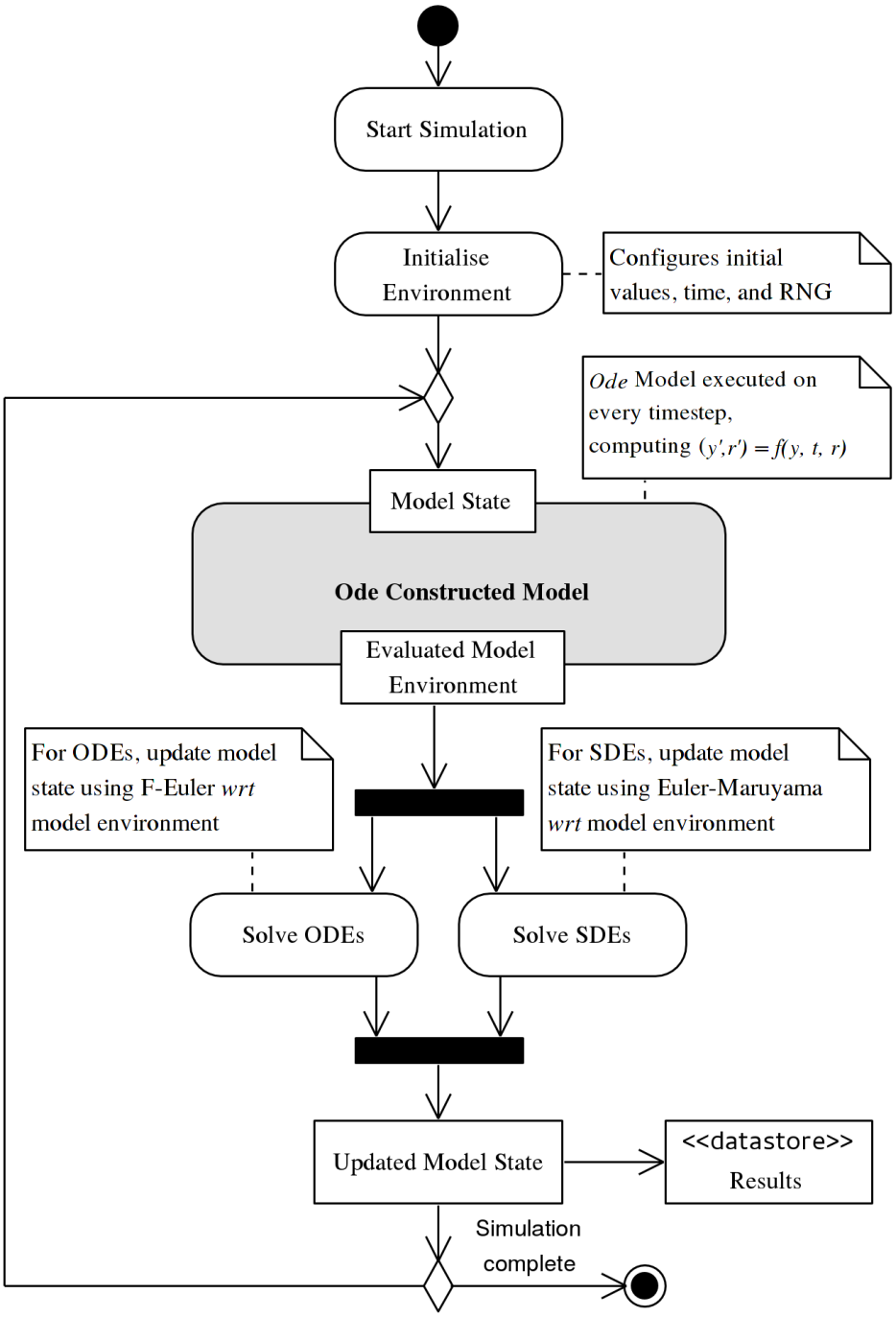


Figure 6.8: Simulation of a hybrid model in *Ode* using ODEs and SDEs. On starting a simulation the model environment is created containing the initial values $y(0)$. The constructed simulation-kernel is evaluated against this model state on every timestep, calculating values required by the solvers, e.g. y'. The ODE and SDE solvers use this evaluated environment to update the model state.

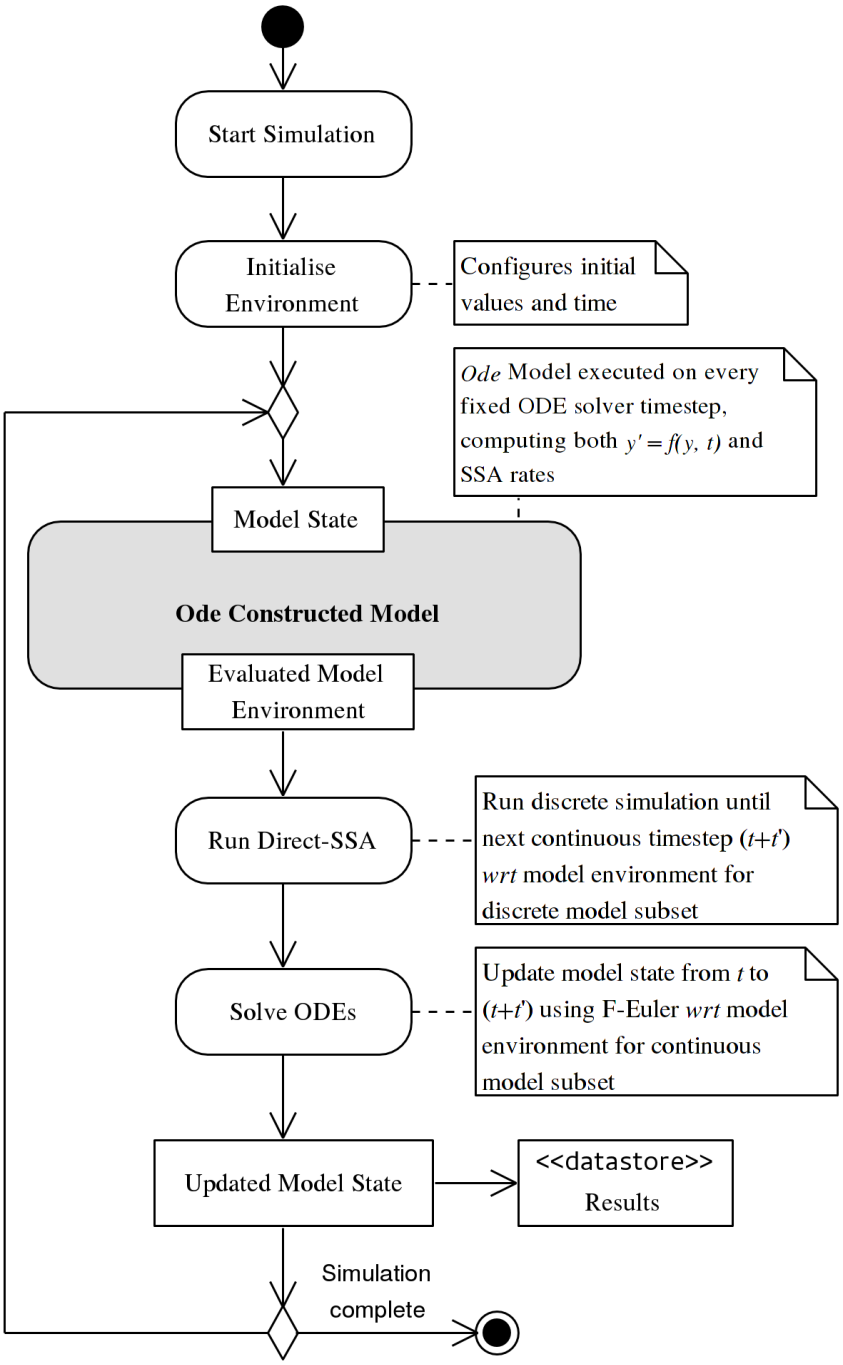


Figure 6.9: Simulation of a hybrid model in *Ode* using ODEs and the direct-SSA, similar to Fig. 6.8. On starting a simulation the model environment is created. The constructed simulation-kernel is evaluated against this model state on every timestep, calculating variables required by the solvers, i.e. y' and reaction rates. The direct-SSA is run up to the next continuous timestep using this evaluated environment, iteratively updating the state of the discrete partition. The ODE solver is subsequently run to the same timestep to update the continuous partition. As discussed in [14, 111], the timestep is adjusted such that influence of each subsystem during independent simulation is small.

6.2 *Ion* DSL

This section introduces a biological DSL, termed *Ion*, for declaratively describing ion channels for simulation using multiple deterministic and stochastic methods as part of an *Ode*-based model. *Ion* is specific to the small, yet important, biological domain of ion channel modelling. It is a high-level, application-specific language that builds upon the more general, and lower-level mathematical *Ode* modelling platform; demonstrating the adaptation of *Ode* to differing biological modelling domains.

Section 2.1.4.2 described a Markov-based approach to model ion channel function that models the conformational state changes that occur during channel activation. By calculating the population of channels in an 'open' state, we may determine the ionic current at an individual channel level. Section 2.2 described and demonstrated how this Markov-formulation ion channel may be modelled and simulated using several mechanisms through reference to biochemical reactions [67]. The current calculated by an ODE, SDE, or SSA representation of the channel may be utilised further within the HH-style voltage equation of a cardiac electrophysiological model (itself using an ODE — see Section 2.1) to perform integrative simulations. However, as seen with the Decker I_{Ks} model in the previous section, modelling and integrating a Markov-formulation channel is a time-consuming and error-prone manual process, particularly when using continuous approximations.

Alternatively we propose the *Ion* external biological markup language/DSL, specifically to describe ion channels for integration into electrophysiological cell models. This has been developed in conjunction with our collaborators with whom we implemented our stochastic and hybrid-stochastic algorithms [37]. The DSL enables the declarative description of a Markov-formulation ion channel; i.e. in terms of the number of channel gating states, activation probabilities, etc., such that we may calculate Eq. (2.5).

This high-level model may be verified and used to generate multiple representations of the channel current, using ODEs, SDEs or SSA reactions, via a source transform into an *Ode* module. This module exhibits a standardised and typed interface, for simulation and/or integration within a cell model in a hybrid manner. Fig. 6.10 depicts this translation process from an *Ion* model to an *Ode* source module.

Introducing such a DSL enables the easier development, modelling and refinement of ion channel models in a verified, declarative manner by modellers; whilst facilitating usage within the modular and efficient environment offered by the lower-level *Ode* DSL. It demonstrates the use of multiple levels of DSLs, developed in conjunction with one another, to expand the domain model; and may be used in conjunction with the modular patterns discussed in Chapter 4 to abstract and reuse channel models, to the benefit of the modelling process.

An ion channel modelling DSL would allow modellers to experiment and capture the behaviour of ion channels using a Markov-formulation. Conversion of such systems into the discrete *Ode* `reaction` syntax is trivial, however conversion into continuous representations, and in particular SDEs, is highly cumbersome. One approach considered was to extend the discrete *Ode* `reaction` syntax for internal conversion into the multiple simulation representations. However, internal implicit conversion is a trait we wish to avoid within the model development process. Furthermore the conversion of a Markov-formulation ion channel model into an SDE representation (see Section 2.2.4.1) is specific and not transferable to other domains.

Instead an ion channel modelling DSL allows the collection of all domain-specific knowledge at this level; with the *Ode* DSL remaining an independent, lower-level language for describing the ODEs, SDEs, and SSA-reactions used within biological mathematical modelling. Specific ion channel modelling features can be added to *Ion* and verified in a domain-specific manner. For instance, ion channel models have a strict structure consisting solely of reversible reactions within a single graph, *Ion* can and does test for this. In a similar vein *Ion* is able to generate the voltage calculation automatically. The DSL can be extended to support further modelling features, such as verification and expansion of the repeated sub-units that often comprise an ion channel into a larger state system. This would not be possible in *Ode* without greatly altering the DSL and introducing ion channel specific modelling features.

We believe this to be the first DSL for modelling ion channels. When combined with the high-level model development environment and efficient hybrid-stochastic solvers offered by *Ode*, it will enable modellers to capture easily detailed ion channel behaviour based on new experimental data and rapidly perform custom investigations. We describe the syntax and semantics of the *Ion* DSL, the mathematics involved in translating to a simulation-ready *Ode* module and several implementation details. Multiple simulation representations generated from

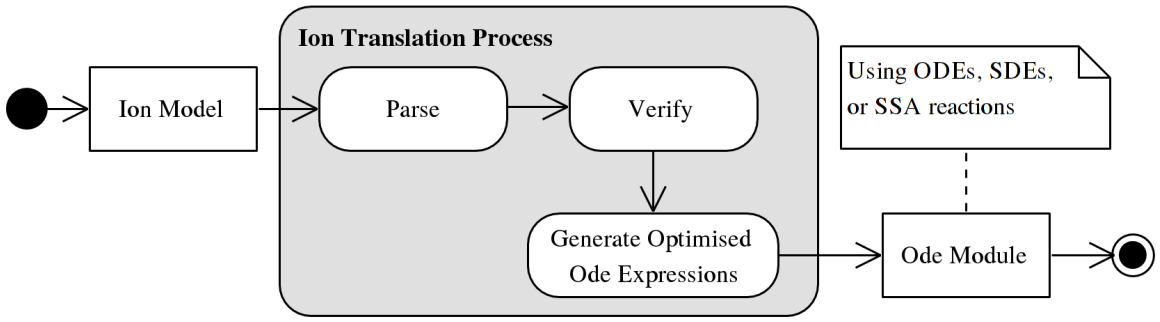


Figure 6.10: Translation process of the *Ion* implementation. An *Ion* channel model is parsed by the implementation, undergoes several verification checks, and is converted into an optimised *Ode* model using the approximations detailed in Section 2.2.

an *Ion* representation of the HH52 *K* channel used in Section 6.3 are provided in Appendix D.5.1.

6.2.1 Syntax & Semantics

Ion has a textual interface, however the design does not preclude a future GUI-based tool to ease use by domain experts. The syntax for declaratively describing the ion channel properties is based on the JavaScript object notation (JSON) [34]. Listing 6.3 presents an example *Ion* model based upon the 3-state channel system provided in Section 2.2, the syntax is as follows. A model consists of a channel name, used to identify the final module when translated into *Ode*, and several nested elements. These elements can be a single value, a list, or a map of key/value pairings, and are described below:

`sim_type` the simulation method chosen to model the ion channel behaviour, can be `ode`, `sde`, or `ssa`;

`density` an integer indicating the number of this particular ion channel per unit-area on the cell membrane;

`equilibrium_potential` the ion channel resting voltage;

`channel_conductance` maximum conductance of the channels when occupying an open-state;

`initial_states` a mapping from the system states to the proportion of channels within that state when starting a simulation;

Listing 6.3 An *Ion* encoding of the 3-state channel model first described in Eq. (2.7). This particular model uses an `sde` representation and sets several channel parameters, such as density, E_l, and g_l. The initial populations of the states in the system are declared and the open states identified. Finally, the transition list represents the bi-directional reactions between channel states.

```
channel IonModule {
  sim_type : sde,
  density : 1000,
  equilibrium_potential : -87,
  channel_conductance : 36,
  initial_states: { X1 : 1.0, X2 : 0, X3 : 0 }
  additional_inputs: {a1, b1, a2, b2}
  open_states : {X3},
  transitions : {
    {transition: X1 <-> X2, f_rate: a1, r_rate : b1},
    {transition: X2 <-> X3, f_rate: a2, r_rate : b2}
  }
}
```

`additional_inputs` an optional list of identifiers that may be used within transition rate expressions;

`open_states` a list of states where the channel is open, i.e. changed ions are flowing across the membrane;

`transitions` a nested list of transition elements that model the state changes within the ion channel. Each **`transition`** element describes a reversible transition between two states with a forward and reverse rate, `f_rate` and `r_rate` respectively. The rates may be any valid *Ode* expression, enabling varying transition rates that are determined on each timestep during simulation.

The various structures required to generate the chosen model approximation are described in detail using the same sample from Eq. (2.7) in Section 2.2, e.g. the stoichiometric matrix, Wiener vectors, etc. These are used during transformation to generate *Ode* computations and simulation operations that model the ion channel operationally according to the chosen simulation mechanism, ODEs, SDEs, or SSA reactions; hence the model is governed by *Ode* simulation semantics.

This code is generated within a standalone *Ode* module that exports a predefined interface, shown in Fig. 6.11. This consists of several cell-level constant parameters and a single component that accepts the membrane and resting voltage and returns the ion channel current. The ion channel current returned by the component is determined by implementing Eq. (2.5). The component

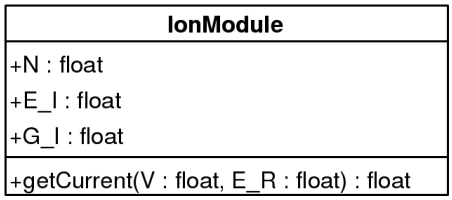


Figure 6.11: Structure of the *Ion*-generated module interface. The number of channels (N), equilibrium potential (E_I) and channel conductance (g_I) are public values. The `getCurrent()` component encodes the channel model and returns the channel current I_I.

accepts additional arguments that are defined within the `additional_inputs` element in the *Ion* model.

Calculation of the model initial/state values differs depending on if a discrete or continuous representation of the ion channel is selected. For a discrete representation, the state values are integers that represent the total number of channels per density in a particular state. For a continuous representation, the state values are reals that represent the proportion of total channels in a particular state.

6.2.2 Implementation Details

Frontend

As *Ion* is a source-to-source translation based system, the final semantics are defined by the previously described *Ode* simulation semantics. Translation semantics are implicitly defined to describe the conversion of an *Ion*-based ion channel declaration into a simulation-ready *Ode* model. This is achieved through the mathematics involved in converting a model into an ODE, SDE, or SSA representation, as described in Section 2.2.

An ion channel model in *Ion* undergoes several verification passes prior to translation into *Ode*. For instance, the system ensures that the initial proportions of channels in each state either sum to 1 or represent valid channel populations with respect to the density value (depending on if a continuous or discrete representation was chosen). The system checks that all states have been correctly defined and that the transitions form a correct state-transition graph comprising a single component with no self-loops. Any verification errors will terminate the process, otherwise will result in a syntactically-valid *Ode* model. This model will be semantically-valid with regards to the generated simulation constructs, however user-defined *Ode* expressions used to calculate reaction propensities in *Ion* may require further verification and correction within *Ode*. The

model consistency checks, combined with the strong type-system and verification features present within *Ode*, provide comprehensive assurances regarding model correctness prior to simulation.

A converted *Ion* model conforms to the *Ode* type system. This includes the use of *Ode* compatible statements to represent transition rates within the *Ion* model; these are checked to evaluate to a float. The units-of-measure system is not utilised in the translated output and is left for future work.

Backend

An *Ion* channel model is converted into an *Ode* module through the `ion` source-to-source compiler. This tool parses the ion channel models contained within an *Ion* file, processes each one and outputs a single *Ode* file that contains translated modules for each channel. Both the *Ode* and *Ion* DSLs are implemented in Haskell and their implementations share source code.

Although the native *Ode* backend is highly optimised (see Chapter 5), during code generation from the *Ion* model we perform several model-specific optimisations to generate efficient lower-level *Ode* code. For instance, model propensities and rates are calculated only once and cached using temporary values in the model, avoiding costed repeated computations within the generated simulation expressions. Additionally, an internal AST was created within the *Ion* implementation to represent symbolically the required simulation expressions. These expressions are optimised to simplify many common mathematical operations prior to generating the *Ode*-compatible code.

6.2.3 Discussion

This section introduced a custom DSL frontend for declaratively modelling ion channels using a Markov-formulation that captures the conformational changes that occur during channel activation. A model in this DSL may be translated into several *Ode*-compatible representations, and imported into integrative cardiac electrophysiological cellular models for simulation using the *Ode* hybrid solvers. We presented the syntax and basic semantics for the *Ion* DSL, including the mathematical details that underpin the transformation into *Ode*-compatible representations suitable for simulation. Several key implementation and usage details were discussed.

Ion demonstrates the benefits of DSLs for enabling new modelling scenarios and possibilities. Modellers may rapidly create and easily alter Markov-formulation ion channel models, confident that they may be converted into simulation-ready representations through an automated, error-free

and verified process.

This work also showcases the *Ode* DSL, presenting a stable, flexible and powerful compilation target for use within the biological modelling domain. The development of the *Ion* DSL was greatly eased by the lower-level modelling environment presented by *Ode*, we may use this knowledge to generate further *Ode*-backed DSLs for other biological application domains. Translated models obtain the benefits of the *Ode* DSL, such as model abstraction mechanisms and an efficient hybrid-stochastic simulation environment, yet may be created in a declarative manner within a more suitable DSL. It validates our hypothesis into how the use of DSLs as a part of software engineering may benefit the biological modelling community.

In the following section we use the *Ion* DSL to generate multiple representations of an ion channel model for simulation within an electrophysiological cell model.

6.3 Simulation Study — Stochastic Hodgkin-Huxley Model

In this chapter we have introduced several stochastic modelling constructs to *Ode* that were demonstrated through the simulation of a single ion channel. Following this we developed the *Ion* DSL to model Markov-formulation ion channels via multiple representations. In this section we demonstrate combined usage of these features through a case study to model and simulate cardiac electrophysiological models. The *Ion* DSL is used to generate multiple representations of an ion channel model for simulation within an electrophysiological cell model utilising the previously created hybrid-stochastic solvers. This will utilise the deterministic and stochastic DSL mechanisms to create reusable, modular models and provide data to test the efficiency of the native-code simulation implementation.

6.3.1 Aim

As in previous simulation studies, we have focussed on generating and simulating electrophysiological models. We plan to investigate stochastic ion channel behaviour specified by Markov-derived channel gating equations, using the Hodgkin-Huxley AP model (HH52) [69] as a basis with a parameterised potassium (K) channel. Multiple representations of the K channel used in this model will be created that utilise continuous deterministic (the default), continuous stochastic, and discrete stochastic modelling approaches. These will be applied to the parameterised HH52 cell model to create custom models for simulation.

The K channel will be specified within the *Ion* DSL, providing feedback regarding the utility and correctness of the DSL design and implementation. The *Ion* representation is used to generate multiple modular variants of the K channel that may be inserted into the integrative HH52 cell model. This will exhibit and test the various new simulation constructs within *Ode*; demonstrating several approaches to modelling and simulation facilitated by the DSL. The resulting family of models will be simulated to test the correctness and efficiency of our implementation and the results compared.

This work will present a comprehensive overview of our stochastic implementation and the classes of models that may now be created. This tests both the efficiency of the hybrid-stochastic solvers and the suitability of the *Ion* DSL for enabling the rapid generation of simulation-capable

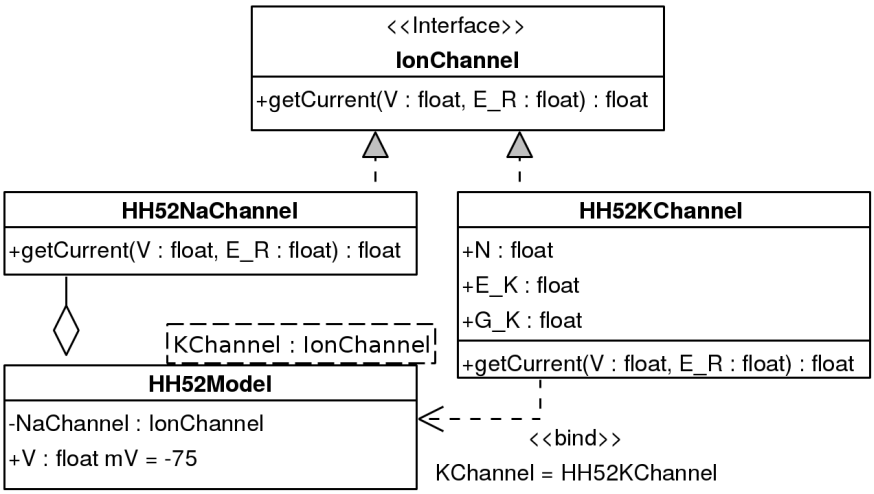


Figure 6.12: Modular structure of the HH52 model used for simulation. The base `HH52Model` module imports a `HH52NaChannel` module representing I_{Na}, and is parameterised on I_K. Several versions of `KChannel` are provided that implement I_K: the original HH52 version and *Ion*-generated variants using ODEs, SDEs, or SSA-reactions.

ion channels; whilst providing a brief investigation into the stochastic behaviour of ion channels within an electrophysiological model.

6.3.2 Models and Methodology

HH52 Cell Model

The HH52 *Ode* model used for these simulations reuses the cardiac modular structure and design patterns developed in Section 4.4. This includes the `IonChannel` interface used to specify compatible ion channel models, however here we use a non-units-annotated interface as the *Ion* DSL does not currently support units-of-measure. The UML diagram in Fig. 6.12 demonstrates this model structure, using a notation described in Appendix D.2. As seen in the figure, the HH52 cell model statically imports the *Na* channel module and is parameterised on its *K* channel, this can be provided by any channel module that implements the `IonChannel` interface. Listing 6.4 presents a segment of the HH52 cell model in *Ode* demonstrating this structure, the full model is not provided but is similar to the parameterised HH52 model in Appendix D.4.2. The model uses the original published parameters and initial conditions for the cell membrane and *Na* channel (see Figs. 2.5 and 2.6).

Listing 6.4 HH52 cell code extract demonstrating use of the parameterised `KChannel` module within the cellular model, following the modular structure described in Fig. 6.12. I_K is used with the remaining HH52 channel currents to calculate the transmembrane voltage.

```
/* Main HH52 Cell Model ************************************************/
module HH52Model(KChannel) {
  import CardiacElec.HH52.HH52NaChannel as NaChannel
  // Cell parameters and i_L currents omitted here ...

  // Channels currents - from modules
  val i_Na = NaChannel.get_current(V, E_R)
  val i_K = KChannel.get_current(V, E_R)
  // Calculate voltage from sum of currents
  ode { initVal : V, deltaVal : dV } = -(-i_Stim + i_Na + i_K + i_L) / Cm
}
```

HH52 *K* channel Model

The HH52 *K* channel was modelled using the *Ion* DSL, based on the Markov-formulation channel state diagram shown in Eq. (2.4). The *Ion*-encoded model is presented in Listing 6.5, capturing the channel state diagram and parameters using the *Ion* syntax described in the previous section. Most channel parameters are unchanged from the original data listed in Fig. 2.6, however the *n* gate is replaced with 5 states in Markov-formulation (see Eq. (2.4)), whose initial conditions are taken from [14].

Running the `ion` translator transforms the *Ion* HH52 *K* channel into an *Ode* module that models its behaviour utilising ODEs, SDEs, or SSA-reactions. The output generated in each form from the *Ion* model in Listing 6.5 is shown in Appendix D.5.1. The equations generated by this translation process were described in Section 6.2, and effectively implement the SSA, SDE, and ODE ion channel modelling representations first described in Section 2.2. *Ion* automatically performs this transformation on the channel state system encoded by the model and outputs an *Ode* representation using the requested formulation. The translated *Ode* module exports a signature that enable its use as a generic module argument to the HH52 cell model depicted in Fig. 6.12 and Listing 6.4, i.e. it implements the `IonChannel` interface, facilitating the generation of a family of comparable models.

Hybrid Simulation

The benchmarking and analysis tools mentioned in Section 5.3.2 are used to script the *Ode* console to programmatically construct the models, compile, and then execute the models. This

Listing 6.5 An *Ion* source representation of the HH52 *K* channel in Markov-formulation, as discussed in Section 2.1.4. The channel parameters are taken from the original model, listed in Fig. 2.6, and the channel state reactions from the structure presented in Eq. (2.4). The initial conditions are taken from [14]. The `sim_type` parameter determines the modelling formulation to use in the converted *Ode* output, examples are shown in Appendix D.5.1.

```
channel HH52KChannel {
  sim_type : sde,
  density : 1000,
  equilibrium_potential : -87,
  channel_conductance : 36,
  initial_states: { C1 : 0.532, C2 : 0.256, C3 : 0.123, C4 : 0.059, O : 0.030 }
  additional_inputs: {alpha, beta}
  open_states : {O},
  transitions : {
    {transition: C1 <-> C2, f_rate: 4*alpha, r_rate : 1*beta},
    {transition: C2 <-> C3, f_rate: 3*alpha, r_rate : 2*beta},
    {transition: C3 <-> C4, f_rate: 2*alpha, r_rate : 3*beta},
    {transition: C4 <-> O, f_rate: 1*alpha, r_rate : 4*beta}
  }
}
```

Listing 6.6 Scripted commands entered from *Ode* console to generate custom HH52 models from the HH52 cellular model and *Ion*-derived HH52 *K* channel module components at compile-time for performing the hybrid simulations in this study.

```
// import HH52 model components
import CardiacElec.HH52.HH52KChannel
import CardiacElec.HH52.HH52Model
// import Ion-Generated K-Channel representations
import IonModels.HH52KChannel.ODEKChannel
import IonModels.HH52KChannel.SSEKChannel
import IonModels.HH52KChannel.SSAKChannel
// construct original models and custom models
module HH52Original  = HH52Model(HH52KChannel)
module HH52_ODEKChan = HH52Model(ODEKChannel)
module HH52_SDEKChan = HH52Model(SDEKChannel)
module HH52_SSAKChan = HH52Model(SSAKChannel)
```

simulation-time construction of custom models is performed through the *Ode* console interface, the commands used to compose the HH52 cell model with multiple *K* channel combinations are provided in Listing 6.6.

Simulation uses the hybrid-stochastic solvers within the native-code compiler detailed in Section 6.1.3. The hybrid simulations were performed with the parameters listed in Table 6.2, using the Euler-Maruyama solver for the SDE representation, the direct-SSA to simulate the discrete reaction representation, and the forward-Euler solver for the ODEs in the system. As mentioned, most initial conditions for the HH52 model were unchanged from the original data provided in Fig. 2.6, however the *n* gate of the *K* channel was replaced with 5 state variables using the initial conditions provided in the table. Multiple simulations were performed, each

Parameter	Value
t_{start} (ms)	0
t_{stop} (ms)	6000
h (ms)	0.01
Output Period (ms)	0.2
$C_1(0)$	0.532
$C_2(0)$	0.256
$C_3(0)$	0.123
$C_4(0)$	0.059
$O(0)$	0.030

Table 6.2: Simulation parameters used to perform hybrid simulation of the HH52 model, where the K channel is in Markov-formulation using either an ODE, SDE, or SSA-reaction representation. The initial conditions for the K channel are under the dotted line and are taken from [14].

running for 6000ms with an AP-generating stimulus occurring every 60ms.

6.3.3 Results

Fig. 6.13 plots a single simulation of the combined model using an SDE-based K channel with an increasing channel population, N. The figure includes a deterministic (ODE) simulation plot as a reference. At low channel populations, the AP is extremely irregular and frequent, occurring both as expected during the stimulus and at other intervals. As the channel population increases, this effect decreases and the stochastic simulation becomes closer to the results traditionally obtained from the original ODE-based model.

We do not attempt to derive biologically meaningful results from these simulations, rather we intend to demonstrate the types of hybrid-simulations that may be performed using the *Ion* and Ode DSLs. However they do demonstrate the potential effects on the AP caused by the stochastic behaviour of ion channels at low populations. Such low populations can occur as a result of drug block and may prove useful when examining ion channel mutation, as discussed in Section 2.2.1.

Fig. 6.14 plots a similar single simulation of the combined model using an SSA-based K channel with an increasing channel population. As with the SDE simulation, at low channel numbers the APs occur irregularly and frequently, occasionally occurring at the expected interval. Increasing the channel population causes the stochastic results to trend closer towards the traditional, deterministic AP results.

Fig. 6.15 plots the mean of 100 AP trajectories using the original HH-formulation K channel

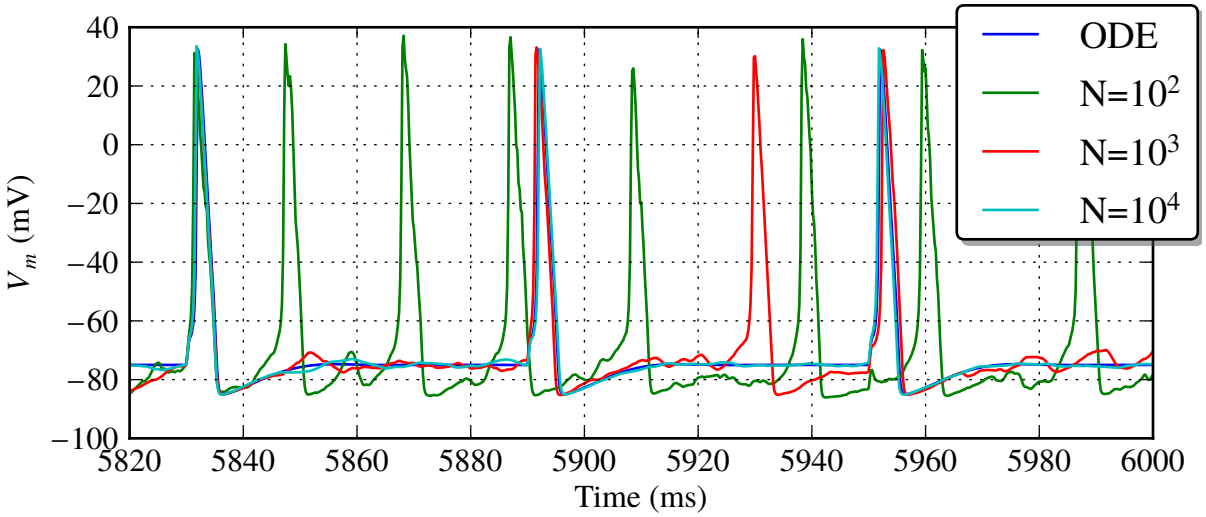


Figure 6.13: AP plots from a single simulation run of the hybrid HH52 model containing a K channel that uses SDEs. A highly irregular and rapid AP is generated when the population, N, is low. As N increases the AP becomes more regular and correlates with the reference AP.

and the *Ion* generated Markov-formulation K channel representations with an increasing channel population. The original HH- and *Ion*-derived Markov-formulation ODE models share the same deterministic AP, thus demonstrating the correctness of *Ion* conversion for this representation. Both stochastic representations share similar APs, and as the number of channels rises the stochastic results become closer and eventually indistinguishable from the deterministic ones. Taking the mean of 100 trajectories presents a different view of the system compared to the single simulation run plots in Figs. 6.13 and 6.14. The beginnings of a regular AP can be seen when N=100 for the stochastic models, and when N=1000 it is clearly visible; this is in contrast to the irregular, frequent APs seen during a single simulation run.

Table 6.3 presents information regarding the performance of the *Ode* hybrid solvers for a single simulation of the hybrid HH52 model using the differing K channel representations with increasing channel populations. We can compare these timings as all solvers are *driven* by the same forward-Euler ODE solver using the same timestep to calculate the membrane voltage. As expected the deterministic representations perform a simulation run faster that the stochastic representations. Looking at the ODE-based channel models, the original HH-formulation model is slightly more efficient than the Markov-formulation model, as it uses a single state value to model the channel's open probability compared to the 5 required to model the individual channel's states.

As expected from the results presented in Section 6.1.1.3, hybrid-simulation using SDEs is

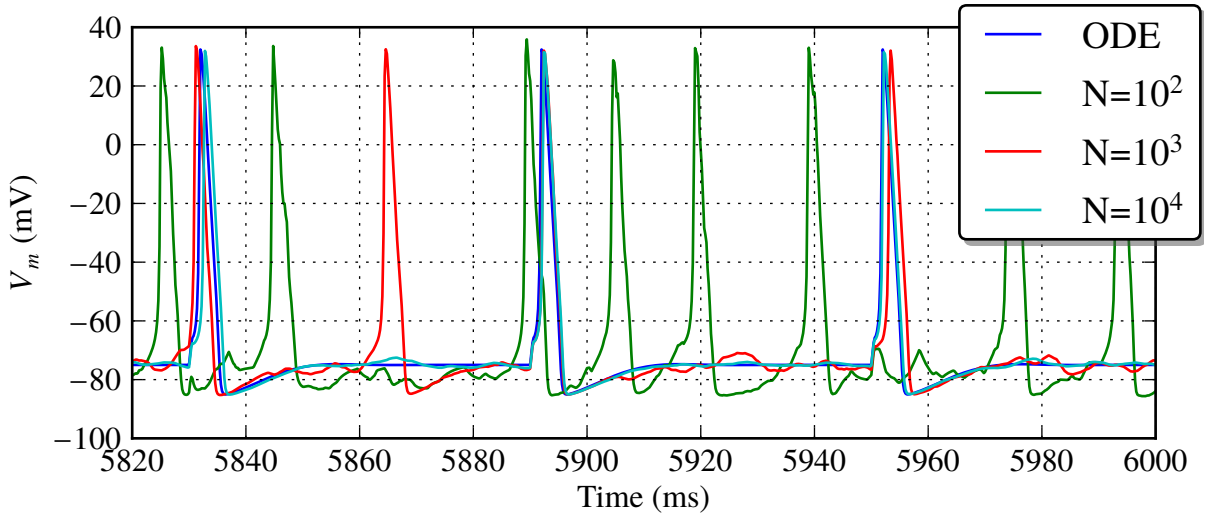


Figure 6.14: AP plots from a single simulation run of the hybrid HH52 model containing a K channel that uses SSA-reactions. The results are similar to those in Fig. 6.13, when the population number, N, is low, the AP is highly irregular and rapid. The AP becomes more regular and the stochastic behaviour diminishes as N is increased.

***K* channel representation**	**Time(s)**		
HH-formulation using ODEs	0.100		
Markov-formulation using ODEs	0.110		
	N=10^2	**N=10^3**	**N=10^4**
Markov-formulation using SDEs	0.279	0.276	0.278
Markov-formulation using SSA-reactions	0.162	0.312	1.650

Table 6.3: Simulation performance of the hybrid-stochastic solvers when simulating the HH52 model containing differing K channel representations. Simulations were performed at several channel populations, N, and are compared against reference ODE implementations.

slower, due to the increased complexity of the model and random number calculations. However, performance remains constant as the number of channels increases. Finally, hybrid-simulation using the SSA is noticeably faster that the SDE equivalent at low channel populations, yet as the channel population increases to more realistic values the simulation time increases greatly. This is due to the computational requirements of the SSA algorithm, with the timestep decreasing exponentially in relation to the sum of reaction propensities in the system (see Appendix B.3). Whilst not tested against C implementations, these hybrid solvers reuse our previously examined ODE, SDE and SSA systems and thus we are confident that they remain competitive and efficient.

6.3.4 Discussion

In this section we utilised the recently-added *Ode* stochastic constructs alongside the *Ion* DSL to create and perform single-cell electrophysiological simulations. The hybrid-stochastic solvers enable simulation of single-cell models controlled by an ODE that contains ion channel subcomponents modelled using multiple (stochastic) representations.

This study demonstrated the ease with which modellers can create ion channel models for inclusion within larger electrophysiological models, e.g. a cardiac model, to elucidate their behaviour within an integrated environment. The *Ion* DSL has been shown to facilitate the rapid creation of models that may be compiled to multiple *Ode* modules in an error-free manner. The availability of multiple modelling constructs integrates well with the module system, modules representing the same ionic currents may export the same interface whilst utilising deterministic or stochastic methods internally. This will provide a rapid means to experiment with differing model representations as needed.

We demonstrated these features using variations of the well-known HH52 model that contained multiple representations of the potassium ion channel. These representations were automatically derived from a Markov-formulation of the channel states encoded in the *Ion* DSL, using ODEs, SDEs, or SSA-reactions. The results presented a high-level look into the effects caused by the stochastic behaviour of ion channels on the AP of a complete electrophysiological model. Irregularities were demonstrated at low channel levels, and a similar approach may be used with cardiac electrophysiological models to investigate cardiac behaviour in such situations. We detailed the performance of our hybrid-stochastic solvers, demonstrating that the multiple modelling and highly-efficient simulation systems provide modellers with many options according to their particular requirements.

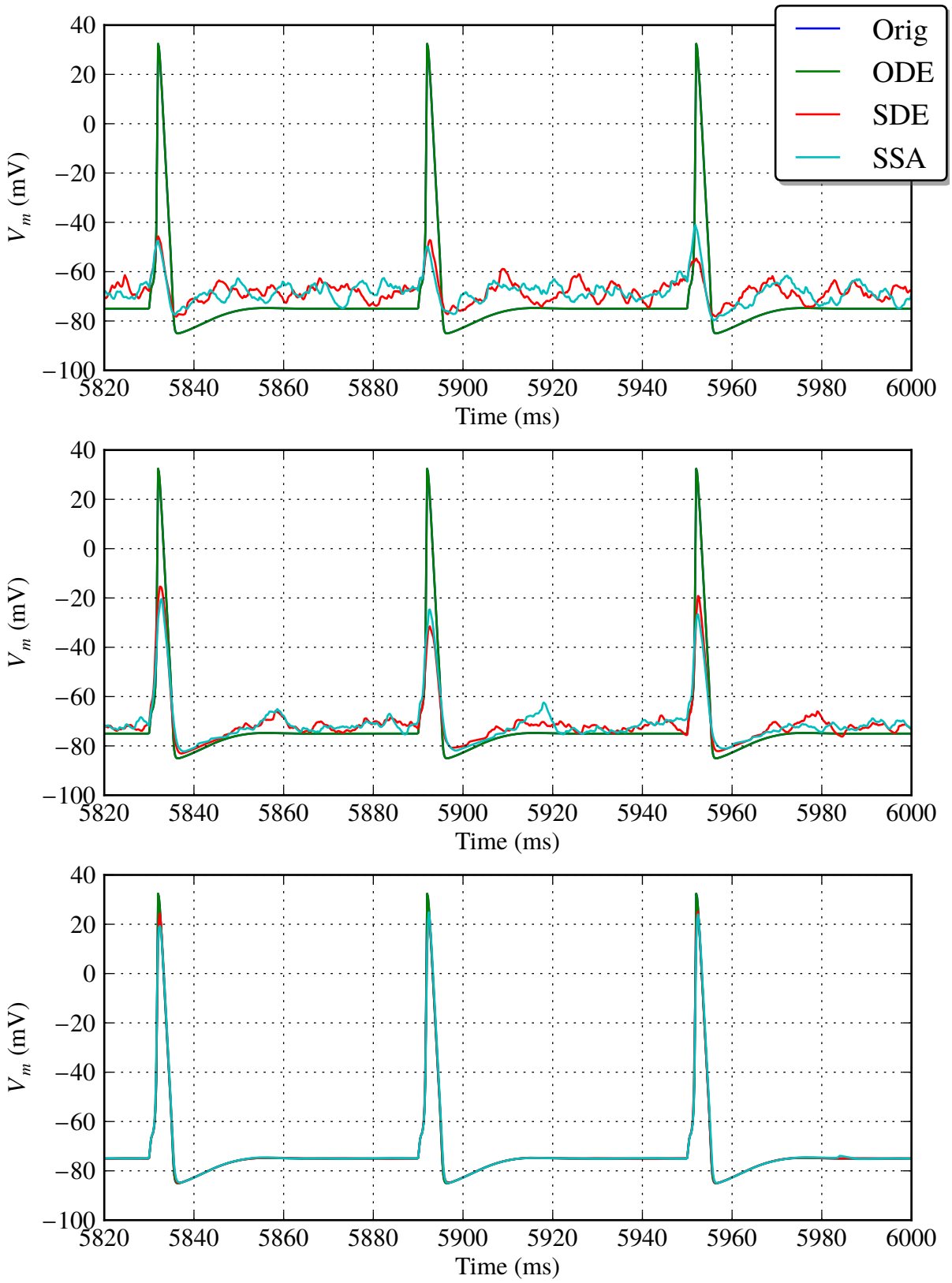


Figure 6.15: AP plots generated from the mean of 100 simulation runs of hybrid HH52 models containing Markov-formulation K channels that utilise ODEs, SDEs, or SSAs. The original ODE-based HH-formulation K channel is provided for reference. The number of K channels, N, in the models, was increased for each set of simulations, starting with N=100 in the top plot, to N=1000, and finally N=10000 in the bottom plot. A visible AP can be seen when N is low, and as N increases the stochastic APs becomes closer to their deterministic equivalents.

6.4 Discussion

In this chapter we have investigated the extension of the *Ode* DSL with stochastic constructs to model biochemical reactions systems. A particular focus was made on enabling easy and efficient stochastic modelling of ion channels within electrophysiological models.

This research began with the introduction of SDEs in *Ode* to modelling continuous stochastic systems. Their syntax and semantics were discussed, along with implementation details regarding native-code compilation. The implementation was tested using a complex ion channel model and demonstrated efficient simulation with performance similar to that achieved by C equivalents. The work was further extended with the introduction of the direct-SSA to *Ode* for modelling discrete stochastic reaction systems. Again we discussed the syntax, semantics and implementation details; and tested the native-code implementation using a discrete representation of the same model. The results indicated extremely efficient SSA simulation, over 40% faster than highly-optimised C-based equivalents. These stochastic additions were utilised to develop several hybrid-stochastic solvers that enable native-code simulation of continuous deterministic systems with continuous stochastic or discrete stochastic sub-components. These particular compositions were chosen to support modelling cellular electrophysiological models with stochastic ion channel components.

To aid development and reduce errors when manually generating ion channel models, we developed the *Ion* DSL to capture declaratively Markov-formulation ion channel function for transformation into an *Ode* module. This enables the rapid creation of models and generation of multiple *Ode*-compatible representations for simulation in whole-cell electrophysiological models.

The stochastic simulation constructs and *Ion* DSL were utilised within a case study involving the development of HH52 models containing stochastic and discrete representations of the *K* channel. Simulation of these models demonstrated at a high-level the impact of stochastic ion channel behaviour caused by low population within a complete, integrative electrophysiological model. This work could be extended in conjunction with modellers to investigate more complex cardiac models with a view to elucidating the several cardiac disorders believed to be caused by low channel populations [19, 26, 64]. The study also served to demonstrate the correct implementation of the hybrid solvers and to test their efficiency within our compilation framework.

We believe we have demonstrated the ease with which ion channel models can be created, whilst providing a flexible simulation mechanism to, for instance, enable investigation of stochastic phenomena or model newly observed experimental behaviour. The modeller benefits from the high-level declarative *Ion* DSL, the powerful abstractions within *Ode*, and gains access to a highly-efficient stochastic simulation environment. Such work advances our hypothesis of using software engineering methodologies to advance model development, including the creation of DSLs and the use of modular development to structure stochastic cardiac models [68, 129]. Furthermore it demonstrates the stability and utility of the *Ode* DSL as a compilation target.

Introducing stochastic constructs into the DSL provides a flexible system for modelling biological reaction systems from multiple perspectives. It provides a mechanism for more accurately modelling ion channels at an individual level and investigating their stochastic behaviour; this may be beneficial when channel populations are reduced. The multiple modelling constructs provided integrate well with the abstraction facilities provided by the *Ode* module system, allowing several representations of the same biological system using differing modelling mechanisms to be created and replaced as needed. This relied on our previously described patterns and processes for developing structured models (see Chapter 4). At the same time the *Ode* backend facilitates efficient stochastic simulations, highly important as stochastic simulation is computationally intensive. The stochastic extensions presented in this chapter complete our current research, and is something we intend to return to and expand upon as part of future research, as detailed in Section 7.2.

Chapter 7

Conclusion

Computational methods have long been used to simulate and generate approximate solutions to biological mathematical models. However the languages and simulation environments used tend to be complex and result in the creation of specialised and *ad hoc* systems that require extensive computing knowledge and often exhibit poor performance. Within this thesis we have described our research and results from investigating the use of DSLs and other software engineering concepts to aid the construction of well-designed, reusable, and extensible biological models that may be simulated efficiently.

We have investigated the benefits gleaned from utilising DSLs, including the *Ode* and *Ion* high-level DSLs; the embedded-DSLs present in the module system, the units-of-measure system, and the scriptable console interface. We have researched and developed the primary *Ode* DSL for creating and simulating mathematical biological models, particularly those derived from chemical reactions. A specific focus was on cardiac electrophysiological models governed by ODEs, SDEs and SSA-reactions. The resulting language is a high-level, typed language that supports programmable unit-checking and conversion and is amenable to efficient model simulation, as demonstrated with our optimising implementation and benchmarks against CellML and C in Chapter 5. The core modelling features and language flexibility and simplicity were demonstrated through several examples and case studies.

The module system enables model abstraction, reuse, and safe composition; aiding the creation of collaboratively-developed module repositories and eventual modelling communities. It enables the construction of large-scale models, promoting software engineering concepts such as encapsulation and aggregation that encourage reusable modular model development.

Parameterised modules allow the construction of reusable generic components that can be safely specialised with multiple alternate implementations. They have many biological modelling uses, e.g. altering the implementation of an ion channel or modifying experimental and simulation parameters. The module system was demonstrated with the development and simulation of a suite of cardiac models parameterised by their ion channels. This was accomplished using patterns and structures developed for cardiac model construction.

Ode models are simulated through an optimising JIT backend that generates efficient, specialised native-code for the combined model and solver. The use of run-time code generation for the multiple solvers resulted in extremely efficient numerical simulations using several modelling approaches. It represents novel research for DSL implementations and in the application of compiler technology to the biological modelling domain. This encompasses research into static, domain-specific model optimisations, auto-vectorisation, and run-time code generation — all of which may be extended in the future.

Implementing stochastic constructs enabled modelling of non-deterministic phenomena that underpin biological systems from continuous and discrete perspectives based on the theory of chemical kinetics. This research demonstrated the flexibility of the modelling core, enabling a single DSL to capture models using multiple, hybrid representations. Stochastic simulation is computationally demanding, yet our approach allowed for highly-efficient native-code simulation. Easy creation and efficient simulation will allow modellers to investigate the stochastic effects present within biological models much more readily.

From a biological perspective this research aims to lower the barrier required to develop reusable (biological) mathematical models. We have undertaken the initial development of a modular framework of cardiac models parameterised by ion channels with a typed interface that promotes reuse. We further developed a declarative DSL, *Ion*, that operates at a high-level in allowing modellers to directly capture ion channel state function that may then be translated into multiple mathematical modelling representations. The modular framework was reused when generating ion channel model modules for performing investigations into the stochastic behaviour of ion channels within electrophysiological cells. These stochastic models were expected, and demonstrated, to produce variations on the AP; such stochastic simulations may help to elucidate the causes behind cardiac cycle irregularities.

7.1 Discussion and Related Work

Having completed our research and obtained several meaningful results, we now take a critical look at the work and discuss our assumptions in relation to our results. As mentioned in Section 1.2, our research spanned three major themes, covering investigations into the effective development of mathematical models through software engineering practices; the need for high-performance simulation of complex, potentially stochastic, models; and the application of this research to cardiac and biological modelling in general.

Section 1.1 presented the case for the application of software engineering to the domain of mathematical cardiac modelling; the domain is currently encumbered with low-level, *ad hoc* models that eschew best practice and inhibit model understanding and reuse. This makes it difficult, if not impossible, to successfully repeat research and reproduce results. We suggested that the introduction of processes used to develop software would alleviate this issue. We chose DSLs as the basis for this work, since many modellers lack the necessary experience to develop complex models computationally.

The research into modelling DSLs and software engineering encompassed work on model reuse, verification and testing, deployment, and development design patterns and architectures. It combined aspects from the *science* of programming, e.g. type and units systems and code generation, with *practical* engineering experience, e.g. modular design patterns and the separation of models from experimental and simulation protocols to ensure repeatable results, to form an implicit domain model for mathematical biological modelling [68, 129].

We further investigated model simulation implementations and obtained highly-efficient model simulation through careful DSL design and the tight integration of the model definition and model-specific solver code. Our research resulted in a DSL for modelling biological systems using multiple modelling mechanisms that provides many abstractions for reusable, collaborative, model development, yet may be simulated efficiently.

This research was used to model ion channel function in cardiac electrophysiological systems, resulting in techniques and interfaces to structure such models, the creation of cardiac model repositories, and extensions/further DSLs for modelling ion channels stochastically based on chemical kinetics. The sections below discuss these research areas further and place the research

into context regarding related work and influences within each area. We discuss this in a joint manner within this section rather than in each chapter as we feel our research is comprised of several smaller, interesting, and novel ideas from software engineering applied to this domain rather than one overriding aspect. Hence it is only when taken as a whole that the impact of the work can be better analysed.

7.1.1 DSLs

The *Ode* language demonstrated the benefits of using specific, lightweight languages to capture models and aid development. We believe it provides a convenient environment for modellers compared to *ad hoc* model and simulation development in either high-level languages that sacrifice performance or low-level languages that can result in a difficult development environment. The type-system enables a development experience similar to that provided by using a scripting language (i.e. Python or MATLAB), whilst enabling model verification and optimised code-generation. The units system extends this verification, providing annotations to unit-check model expressions in a form modellers would be familiar with. It is difficult to determine objectively the utility and ease-of-use of a language, however we believe the DSL provides an effective model development environment, as evidenced by the models developed during our research.

Related Work

The initial impetus to develop a DSL was to provide an alternative to general languages, such as MATLAB and C/C++, currently being used to generate complex and inextensible models. In contrast *Ode* provides first-class modelling constructs, techniques to aid model verification, and more efficient simulation, whilst supporting high-level structural abstraction facilities similar to such languages.

These abstractions are derived from the DSL's roots in functional programming concepts, and the internal IR is based on the simply-typed lambda calculus [7] (as the datatype in Appendix A.3 shows). We extended the lambda calculus with primitive constructs used to describe mathematical models, i.e. initial values, ODEs, SDEs, and SSA-reactions; representing a novel extension and application of this language to encode the expressions calculated on each simulation timestep.

ML-derived functional languages, in particular OCaml and Haskell, served as the inspiration

for many syntactic and semantic aspects of the language[1]. The type system is based on the same Damas-Hindley-Milner inference algorithm that underlies such languages, and the units-of-measure system is based on similar extensions to ML [137] and F# [78, 79, 131]. However, we did not build the DSL on top of these languages as we required control of the compilation process to implement model-specific optimisations and a custom simulation backend. This combination of functional language concepts along with our own features has resulted in a novel DSL suited for modelling mathematical systems; where our own contributions include the language simplicity and ease-of-use, the model verification features, the ability to script simulations, and the efficient simulation system.

A major area of related research is existing modelling DSLs, including SBML [73] and in particular, CellML [93] (see Section 2.3.1.2). We believe *Ode* is far more suited to the model development process than these model curation and markup DSLs. Specifically, as CellML was designed to curate existing, validated models we feel it is impracticable as a model development language, providing limited computational power. In contrast *Ode* facilitates the abstraction and parameterisation of expressions, enabling reuse of model components. Models may be structured naturally in *Ode* using encapsulation and containment, perhaps to express underlying biological structure, using scoping rules familiar to modellers; rather than requiring explicit grouping as in CellML that in turn decreases model cohesion. The type and units-of-measure systems in *Ode* also promote model verification. As discussed within this thesis, *Ode* provides further benefits, such as highly-efficient simulation performance and a scripting interface for configuring repeatable and reproducible simulations. As a result we believe is far more suited to model development.

7.1.2 Modularity and Reuse

We investigated the use of modules as a means to structure and encapsulate biological models, The module system enables the construction of complex models from the composition of smaller components that encapsulate submodels, values and expressions through a typed interface. The programmable module implementation provides features including abstraction, generics, and aggregation, enabling the development of reusable and extensible model frameworks such as the cardiac framework in Section 4.4. During this development we identified several patterns

[1] *Ode* itself is implemented in Haskell.

for structuring cardiac models that may form an eventual pattern language applicable to the domain [4, 16, 129].

Modular development is common in software engineering, resulting in systems with low-coupling and high-cohesion; however it requires upfront investment and design that may not be demonstrated by model developers. We believe the gains are worth the additional effort, modellers may construct models on a piecewise basis to investigate differing biological phenomena and create reusable, collaboratively-developed model repositories. We hope that the cardiac modular framework discussed in this thesis will serve as an example for collaboratively developing and structuring models, whilst providing insightful results in itself.

Related Work

As discussed in Section 4.1, code is commonly well structured in general-purpose programming languages, either using classes in OO languages or modules in imperative and functional languages. Good code structure enables the construction of extensible, large-scale programs from reusable components that may be developed independently. We are influenced by modular languages such as Modula-3 [18], and similar to our work on DSLs investigated features provided by functional languages to structure programs. The Standard ML and OCaml module systems inspired the *Ode* module system, supporting static and parameterised modules (i.e. functors) over the base language. These systems implement modules as a form of the simply-typed lambda calculus over expressions, providing a programming language in itself. The method was published in [89] and served as the basis for our approach. We believed such a powerful module system would prove highly useful to structure and reuse biological models, as was demonstrated in Chapter 4, and it represents one of our core contributions to the modelling domain.

Several biological modelling DSLs have also investigated and included module systems as a means to structure models that capture the logical grouping of biological structures. For instance LBS [114] has been used to describe complex biochemical reaction systems in a safe and modular manner. However, to the best of our knowledge we have not seen the advanced modularity present in functional languages and in *Ode* applied to the mathematical biological modelling domain.

The CellML module system was described in Section 2.3.1.2. Modularity in the DSL is significantly limited, requiring the static importing of modules that must be manually linked whilst taking care to match the correct names of values in the process, decreasing model cohesion.

SBML Level 3 [73] also provides support for modules in a similar, static, vein to CellML. Unfortunately, like CellML, modules are an optional feature unsupported by most development tools and their respective modelling communities. In contrast *Ode* provides integrated support for the programmable substitution of model components at compile-time, and has a type-system that aids the construction of collaborative, interface-driven, modular model development.

7.1.3 Performance

The efficient simulation performance observed validates the many decisions made within the DSL design and the backend implementation. The DSL was inspired by the C++ ethos of providing high-level abstractions that do not inhibit performance. For instance the pure numerical core enables many constant optimisations and the abstraction features provided by the module system are all resolved at compile-time.

The creation of custom, model-specific simulation code enables tight integration and optimisation between the model and specified solver during compilation. Direct code-generation is a novel approach to model simulation that can be extended to perform simulation-time tracing and recompilation to obtain further optimisations with respect to model state [3, 10].

As Clang compiles via LLVM, many of the same optimisations, including our *vecmath* optimisation, could also be applied to models compiled-via-C. However direct compilation against LLVM allows the generation of greater semantic information regarding (the immutability of) model calculations. This in turn enables more aggressive optimisations in LLVM than are possible when performing compilation-via-C. For instance we can directly state the memory access patterns of values, and have access to low-level CPU features such as the FPU state and vector registers. The benefits of this approach were demonstrated in Sections 5.3 and 5.4, where models in *Ode* were simulated more efficiently than their CellML-derived C counterparts when both were compiled via LLVM with the same optimisations.

The *CoreFlat* IR has proven to be a successful low-level model/simulation representation that has been expanded to support several simulation mechanisms, facilitates optimisations, and is suitable for execution on a wide variety of hardware platforms, e.g. GPUs. We believe many improvements and further optimisations are possible, including to the auto-vectorisation mechanism.

Related Work

Our work on efficient simulation centred around the use of LLVM to generate a native-code representation of the model for simulation, with LLVM providing an extensive range of low-level optimisations. We considered developing the DSL on top of an existing language, an approach taken by several modelling DSLs [52, 97]. However such an approach is limited by the semantics and performance characteristics of the underlying language, and would not have provided sufficient access to the code-generation backend to enable efficient simulation.

Several modelling languages, including CellML, rely on compilation-via-C to provide efficient model simulation. This approach was also not considered, as by coding against LLVM directly we were able to provide greater semantic metadata regarding the model to maximise optimisations. Our approach was justified with the performance results in Sections 5.3.2 and 5.4.5 favourably comparing *Ode* against CellML-derived C models. The use of LLVM to generate model code is becoming more common, for instance in [125] to simulate SBML chemical reaction models. A further contribution to enabling efficient simulation was the custom-generation of model-specific solvers during compilation. These solvers, for instance, take into account the fixed size of the system and/or generate custom direct-SSA reaction selection logic. These are the initial benefits accrued by a code-generation approach to simulation, and we are presently unaware of any other biological modelling languages using such an approach.

Several model-specific optimisations were developed and implemented at the *Ode* and LLVM levels during compilation. We focussed specifically on optimisations that increase CPU throughput whilst only negligibly altering the simulation results, e.g. floating point optimisations and auto-vectorisation of transcendental functions. A orthogonal approach is to optimise and simplify the model itself, for instance in [29] the author replaces transcendental functions with look-up tables, potentially altering the model behaviour in the process.

7.1.4 Stochastics

The DSL semantics were designed with extension in mind; new simulation operations can be implemented that modify model state values with respect to the evaluated simulation-kernel upon every simulation timestep. This was demonstrated with the stochastic DSL extensions that enable modellers to capture stochastics effects within continuous and discrete biological models

easily through the high-level DSL syntax. Implementation of these extensions was shown to enable highly-efficient stochastic simulation via specialised code-generation, further validating the system backend. Further stochastic techniques could be implemented, for particular efficiency or domain concerns, e.g. tau-leaping simulation [58, 60, 91] or alternate SDE solvers more suited for simulating stochastic ion channel behaviour [37].

We included limited support for the modelling and simulation of hybrid-stochastic systems, enabling large scale systems to be modelled using ODEs whilst containing stochastic subcomponents. This enabled efficient simulation of stochastic ion channels within a (cardiac) electrophysiological cell, and was demonstrated using a variety of continuous and discrete channel representations. The accuracy and errors introduced by our hybrid-schemes were not discussed; several issues are detailed in [14, 15] and the results do follow expected trajectories.

Related Work

Extending *Ode* to support stochastic modelling and simulation resulted in several contributions. Many SDE-based models we investigated used general-purpose languages, such as MATLAB, and suffered from issues we have previously mentioned regarding their use, e.g. lack of extensibility and modularity. We extended *Ode* to provide first-class support for SDE definitions, allowing modellers to benefit from the DSL features to develop complex stochastic cardiac models in a reusable and extensible fashion. We are unaware of any biological modelling DSLs that provide specific support for continuous stochastic models.

Similarly stochastic discrete models are often created in general-purpose languages such as C/C++, as efficient simulation is necessary, and suffer from similar drawbacks. Several DSLs and simulation environments have been developed for discrete models, for instance SBML and LBS, however they do not support the range of high-level abstractions provided by *Ode*. Significant research has also been undertaken into the efficient simulation of stochastic discrete systems. For instance, approximate simulation algorithms, such as τ-leaping [60], or simulation algorithms specific to certain biological contexts [118]. Recent advances have also included using LLVM to compile SBML models [125], and GPU-based model simulation [40, 82, 92]. However we have yet to see elsewhere the use of model-specific solver code-generation, and believe our approach could provide highly-efficient GPU simulation through use of the optimised *CoreFlat* backend described in Chapter 4.

We provided support for the hybrid simulation of statically partitioned models that contain a stochastic subcomponent. This was implemented specifically for modelling cardiac cells with a stochastic ion channel as demonstrated in Section 6.3. Previously such cardiac models would have been created manually using MATLAB in an *ad hoc* manner [14, 15, 110]. In contrast our approach allows for the verified, modular, construction of such electrophysiological hybrid models that may be simulated significantly more efficiently. Several optimised environments have been developed to simulate biochemical reaction systems in a hybrid fashion using multiple representations, i.e. ODEs, SDEs, or SSA-reactions, such as Hy3S [72]. Research has also focussed on the automatic and dynamic partitioning of hybrid systems during simulation into more suitable representations in response to system state changes [111]. However we are unaware of any hybrid systems using code-generation to create combined model and simulation code, and believe our DSL could be extended to provide automated partitioning backed by JIT-compilation for highly-efficient hybrid simulation.

7.1.5 Summary

Our research has demonstrated the successful application of software engineering to advance the modelling process and enable modellers to develop complex biological models in an extensible, reusable manner. This research was spearheaded by the introduction of several DSLs, but also encompassed work on modularity, code generation, and type and units systems. Many avenues exist by which this research may be furthered, from both computational and biological modelling aspects, these are discussed in the following section.

7.2 Future Work and Extensions

We produced several novel pieces of research and obtained meaningful results through our investigation into the use of software engineering, including DSLs, to facilitate efficient biological modelling and simulation. Areas where we may continue our work include extending the *Ode* and *Ion* DSLs; improving the simulation implementation and numerical schemes; and research to improve modeller ease-of-use and automation. A few initial thoughts are presented below.

7.2.1 *Ode* Changes & Improvements

Several improvements can be made to the *Ode* DSL, and the code-base would benefit from extensive cleaning and refactoring. We are satisfied with the simple syntax of the DSL, having used it successfully with collaborators to develop several complex models. However areas for improvement exist with regards to the semantics and general usage.

7.2.1.1 Abstraction Mechanisms

We noticed that several of the aggregation and abstraction mechanisms in the DSL, i.e. modules, components and records, have overlapping uses. Throughout the thesis we have identified areas and patterns where each could be used, however we would like to investigate combining records and modules into a single heterogeneous structure for data encapsulation. Previous research exists on this topic, including first-class modules in OCaml [90, 132] and alternate record systems [17, 88], and would greatly simplify the DSL. This could be taken further by extending the record system into a lightweight object system.

7.2.1.2 Types and Ontologies

Structural subtyping is used within the DSL to check modules for compatibility via their exposed type interface, increasing the potential for module composability. Sometimes we may wish to treat two modules as incompatible even though they exhibit the same interface, but the current type-system is too restricted for this. For instance, in Section 4.4 a single interface is specified for all ion channels, however this allows substituting *Na* channels where *K* channels would be expected — a semantic rather than type error.

Several solutions to this issue exist. One would be to ensure the interfaces do not overlap, e.g.

by using differing attribute names. Another would be to extend the type system to provide type constructions, allowing users to create a new type for each channel such that incorrect channel usages become type errors. An initial implementation of this was developed, however it soon became apparent that type constructors complicate the model development process and are far removed from the biological model semantics.

We now believe introducing ontologies into the DSL [33, 42, 52, 93] would be a better solution, enabling modellers to create and attach semantic metadata to model elements that have a known and agreed upon biological meaning. Ontologies would form a semantically-aware, domain-specific type system for which we could develop an automated ontology checker within the implementation. Similar work in this area includes Type Providers in the F# programming language [131].

Ontologies are naturally understood by modellers and would not greatly complicate model code. An agreed upon domain ontology would aid functional curation and model comparisons. It would facilitate the undertaking of phenomenological studies, as in Section 4.4, by linking data and parameters across models. We have previously implemented an ontology system for cardiac models within PyCML [52], and several ontologies exist for cardiac electrophysiological models[2]. They would allow use to introduce semantic descriptions to models, ensuring semantically-valid model construction and increasing the safety of model compositions.

7.2.1.3 User Interface & Interaction

The intended users of the modelling system are domain experts and modellers rather than software engineers. However at present the system is difficult to compile, distribute and use effectively. The current implementation runs only on 64-bit Linux systems whilst many modellers utilise the Microsoft Windows platform. Work is required to remove platform- and architecture-specific code however 3rd-party libraries, LLVM in particular, are hard to utilise on Windows.

A further concern is the extremely terse feedback provided to modellers in the event of a compilation or run-time error, currently consisting of an error code and debugging information. Ideally this will be expanded to provide comprehensive detailed information regarding syntactical and semantic errors, including line and column numbers to locate the error. Such feedback is vital to increase uptake and ease of use by modellers.

[2] `http://bioportal.bioontology.org/ontologies/EP`

Finally, the system user-interface is text-based, for both model development and when configuring simulations. This enables scripting experimental scenarios, as used within our simulation studies, and is more familiar to traditional software developers. We have provided a basic tutorial for using the system in Appendix D.1. However in the long-term we feel that GUI-based modelling tools will greatly aid model development, particularly for the *Ion* DSL. Tooling in general facilitates the software development process, and we will investigate developing an IDE to aid text-based creation, development and debugging of models, perhaps based upon the Xtext DSL environment for the Eclipse IDE[3].

7.2.1.4 Optimisations

The *Ode* implementation itself has seen few optimisations, for instance many internal structures exhibit linear, and even quadratic time-complexity with respect to model size. This becomes an issue when working with larger models, such as GPB10 (see Section 2.1.3), that can require over 20s to validate, optimise, and compile. Optimisations across the *Ode* code-base would enable the system to provide faster feedback during the model development process and would remove a large bottleneck when performing multiple simulation runs.

7.2.1.5 Proofs

We presented a basic implementation of ML-style type inference [119] to the *Ode* modelling language, demonstrating its use in aiding model verification. This was further extended to support units-of-measure checking based on existing work in [78, 79, 131] and a typed module system inspired by [89]. The type rules for these systems are provided in Appendix A. However we have not included proofs of type-safety and correctness [119]. These systems are basic implementations of an existing algorithm, and as such we do not believe it was necessary to provide such proofs simply to demonstrate the benefits of type systems to the biological modelling domain from a software engineering perspective.

Similarly we have provided big-step operational semantics for the translation of *Ode* models into low-level IRs, and of model evaluation in the *CoreFlat* IR. Again we did not provide proofs regarding evaluation correctness, however the same evaluation rules are implemented directly within the interpreter and act as an operational reference. The results obtained from running

[3] `http://www.eclipse.org/Xtext/`

models in the interpreter have been compared successfully against the native-code compiler and against alternate simulation platforms (see Chapter 5), acting as a form of implementation verification.

We may consider providing proofs of subject-reduction (i.e. preservation) and progress for the type-system using the Coq proof assistant `http://coq.inria.fr`. This may also be extended to provide a strong progress proof for model evaluation at the *CoreFlat* IR level.

7.2.2 Parallel and Heterogeneous Computational Simulation

To further improve simulation efficiency we intend to investigate the parallel simulation of models on increasingly common multi-core and cluster systems. Keeping with our theme of a modeller-accessible language, this would require research into methods for performing parallel simulation in an automated fashion rather than introducing specific constructs into the DSL. However our emphasis on increasingly static, optimised simulation kernels conflicts with the synchronisation and sharing overheads required for parallel simulation on shared memory systems. An alternative approach would be to support the automatic execution of independent simulation instances across multiple shared memory and cluster computing configurations. As described in Section 2.4, such cluster machines are comprised of similar CPU architectures to desktops, and hence our work on maximising performance on desktop-class CPUs can be translated over. Multiple simulation trajectories are required when approximating the chemical master equation (CME) probability distribution or performing parameter sweeps [80] and is a trivially parallelisable operation. Support infrastructure would be needed to control such simulations, whereby simulation results could be compared automatically to each other and to expected results using functional curation [32].

Additionally, our work in utilising LLVM and vectorisation paves the way for GPU simulation, as GPUs are essentially clusters of numerical vector co-processors. Research will be conducted into producing GPU compatible code from the *CoreFlat* IR, where we believe an efficient mapping is possible. This may be accomplished by producing OpenCL compute kernels that will enable heterogeneous simulation on CPUs and GPUs, representing a novel approach to model simulation. This may lead towards real-time simulation [8].

7.2.3 Dynamic Hybrid Partitioning

Currently our hybrid modelling support is restricted to a static, modeller-defined, partitioning of the system prior to simulation; we may extend this to support dynamic hybrid-stochastic systems. This would enable biochemical reaction systems to switch their particular modelling representation from discrete stochastic (SSA), to continuous stochastic (SDEs), to continuous deterministic (ODEs). This would be based on modeller-directed thresholds and the current model state in order to balance simulation speed and accuracy (see Section 2.2.6).

Each reaction may be annotated to specify the approximation scheme used when partitioning, these annotations may include the current reaction propensities and species populations to enable dynamic repartitioning during simulation. This would enable models to capture the effects of population changes and simulate them appropriately with regard to the impact of their stochastic behaviour, e.g. ion channel numbers in a cardiac cell. It would integrate well with our code-generation approach, the implementation may specialise and regenerate model-simulation code in response to tracing/profiling model state changes during simulation; hence always executing an optimised simulation. Dynamic repartitioning would be an extensive research undertaking, as tracing/bookkeeping overheads and repartitioning will add complexity.

7.2.4 Model Parameters

There are several features we would like to include for modifying and specifying model parameters, particularly with regards to experimentally-derived data. Firstly *parameter sequences* enable capturing a range of numerical values that a parameter may take over the course of multiple simulations, ideal for performing parameter sweeps and sensitivity analysis. The *Ode* implementation could then be updated to perform automatic parallel simulation across the sequences on multi-core and cluster systems, as described in Section 7.2.2. The example below demonstrates a potential use and syntax, identifying several possible values for the `NumChannels` parameter and a sequence of values for `ClampedVoltage` parameter,

```
1 module Model(K) {
2   val NumChannels = [100, 1000, 10000] // 3 explicit values
3   val ClampedVoltage = [-60, -30, ... 60] // sequence of 5 values
4   val i_K = K.getCurrent(ClampedVoltage, E_R, NumChannels)
5 }
```

Furthermore we would like to support *parameter estimation* and fitting utilising a successive approximation method, this could be done within the *Ode* run-time, or by integrating advanced parameter estimation tools such as Nimrod [1]. Finally, *parameter ranges* allow modellers to configure allowable tolerances for values during simulation, similar to *code contracts* [100]. Whilst a simple addition they are important for stochastic simulations that may become unbounded and parameter sweeps that may cause instability. Native code-generation would enable the injection of efficient range checks within the simulation kernel. The example below demonstrates the potential syntax, clamping the `C` and `O` initial values within the 0 and 1 range (e.g. as required for ion channel state proportions),

```
module KChannel {
  component getCurrent(ClampedVoltage, E_R, NumChannels) {
    // state values clamped with ranges
    init C { >= 0 and <= 1 } = 0.6
    init O { >= 0 and <= 1 } = 0.4
    /* ... current calculation here ... */
    return I
  }
}
```

7.2.5 Multi-scale models and Chaste Integration

Single cell cardiac-models are used within multi-scale tissue and whole organs models to determine the voltage at nodes on a spatial mesh. Such simulations are extremely computationally expensive and require the use of large-scale simulation environments such as Chaste [120]. We plan to integrate the *Ode* implementation into Chaste. This would allow modellers to develop a single-cell model using *Ode*, taking advantage of the modelling abstractions and efficient simulation offered, and utilise it within multi-scale systems in Chaste.

Work is already progressing in this area, using the *Ode* object-file backend and FFI (see Appendix C.4) to allow run-time linking into pre-compiled *Ode* models from within Chaste. This *Ode* backend generates thread-safe, allocation-free code that facilitates parallel simulation on multi-core and cluster machines supported by Chaste. This integration will allow modellers to rapidly develop single-cell models in *Ode* whose behaviour can be observed at the tissue and whole-organ level, enabling efficient investigation into a wide variety of cardiac electrophysiological properties. The integration is not limited to cardiac modelling, but may also be used within Chaste's cellular framework. Here *Ode* could be used to describe cell-cycle models

and sub-cellular reaction networks, such as the Tyson-Novak cell cycle model or Delta-Notch signalling pathway [28, 108, 135].

Furthermore we are interested in integration with the recently released FieldML specification as part of the Physiome Project[4] [23] for curating hierarchical mathematical models that span multiple scales. *Ode* could be used for developing spatially-homogeneous models within this framework.

7.2.6 Output and Analysis

Section 3.4 and Appendix A.6 detail the minimal data-format used when saving simulation results, consisting of a single array of all state values taken at regular snapshot intervals t_i. We would like to allow modellers to configure which values are included within the simulation results and adopt a more general data-format, preferably one that enables collation of multiple related results into a single output file, e.g. when performing multiple stochastic simulation runs.

We have developed a framework in Python to analyse results (see Section 5.3.2). Currently this is focussed on cardiac models however it could be extended into a more general analysis framework. We are interested in integrating this framework and the DSL with recent work on using SED-ML[5] [84] to describe experimental protocols and Chaste to perform *functional curation* of models [32]: that is to enable the construction of simulations with reproducible results that may be compared against experimental data and previous domain models (see Section 2.3.4.1).

7.2.7 Potential Biological Research and Use

We have focussed our research on the application of software engineering processes to enable modellers to efficiently and effectively conduct research and simulation of cardiac electrophysiological models. The following list details some of the scientific questions and research that this work will enable from a biological modelling perspective. Taken as a whole, this biological research enables a greater understanding of cardiac function through modelling, facilitates conduction of *in silico* experiments into pathologies or pharmacological agents, and furthers our domain's goal of predictive medicine:

[4] `http://physiomeproject.org/software/fieldml/`
[5] `http://sed-ml.org/`

- simulation of complex models containing more detailed representations of ion channel states and their effect on the AP;
- investigation into cardiac cycle disorders caused by ion channel irregularities and mutations from a stochastic perspective, such as Long-QT syndrome [20, 26];
- research into the robustness of a model to ion channel variations and fluctuations due to stochastic modelling;
- a larger range of hybrid-stochastic models to be designed and their behaviour investigated, building upon the results in Section 6.3;
- the possible inclusion of single cell, potentially hybrid-stochastic, models within whole-organ models;
- investigation into individual ion channel states, such as those affected by drug blockage or mutations, on the AP.

We hope that the primary users of this work will be computational biological modellers who presently use general-purpose languages such as MATLAB to develop model implementations and wish to perform *in silico* experimentation. We envisage the work being used by modellers developing new models based on newly-obtained experimental data and the reuse of existing model subsystems. Within Section 1.1 we discussed several issues that currently hinder biological model development. These included the use of complex general-purpose languages or primarily curation-based languages such as CellML for model development. We noted that this approach did not provide features that aid the development of biological mathematical models in a systematic, extensible, and verifiable fashion.

These shortcomings and needs influenced several features within our work — for instance, the DSL provides several modelling abstractions that encourage succinct models that can be composed, reused and extended in a verifiable manner. The module system enables the generic substitution of model components, aiding the construction of reusable modules that mirror the encapsulation and compartmentalisation seen in biological models. It enables the creation of multiple representations of a modular component, aiding model development making use of multiple experimental datasets as in [105, 138]. These modules may be placed within repositories that further encourage collaborative model development.

Modellers may utilise the work to simulate complex models using the highly-efficient simulation implementation. This helps decrease the turn-around time when comparing results to existing models and data during experimental model development. Furthermore, efficient simulation is essential when performing multiple simulation runs, for instance when performing sensitivity analysis or stochastic simulations. Stochastic models, both continuous and discrete, may be developed to succinctly model certain biochemical phenomena, such ion channel behaviour at low population numbers. They utilise the *Ode* modelling abstractions and may be simulated efficiently within hybrid modelling scenarios specifically suited to the cardiac domain. Stochastic modelling is not supported by other cell-level modelling DSLs, such as CellML, and can be cumbersome to develop in general-purpose languages; hence the introduction of the *Ion* DSL and direct support for stochastic constructs within the *Ode* DSL.

Our primary use case was to investigate and aid the development of cardiac electrophysiological models from a software engineering perspective. However we may also look at the application of the DSL to other biological domains, as it may be used to model other systems that can be captured as the interaction of biochemical reactions over time, such as gene regulatory networks and signalling pathways [28, 135] in a stochastic or deterministic fashion. The DSL may also be used to capture highly-complex, ODE-based, cell cycle models [103, 108]. Such complex models could be expressed succinctly within *Ode* and would benefit from the modelling and simulation features provided. However, promoting usage and uptake of our DSL and processes within both our own cardiac domain and the larger biological modelling community is an open and hard question. We believe documentation, examples and tutorials (as in Appendix D.1), and GUI tools would help our efforts.

7.3 Summary

In this chapter we have provided an overview of our research with respect to our aims and hypothesis, the motivation for doing so and the scientific merits. We have critically discussed our work, within the context of related research in the field, and have emphasised the benefits of our approach. Finally we described our current and future plans within this area.

This thesis as a whole detailed our research and development into the application of software engineering methodologies, including the use of DSLs, to the process of developing biological models, with a focus on cardiac electrophysiological models. As discussed in Chapter 1 we had noted several issues with computational biological model development, and wished to aid modellers in performing investigations into cardiac cell properties efficiently via multiple schemes.

We developed a primary DSL providing several advanced features that make it suitable for high-level, collaborative development of reusable, extensible biological mathematical models. The DSL features were used to demonstrate the large-scale construction of modular cardiac models through the application of software engineering methods, processes and patterns. The implementation exhibits highly-efficient simulation through the creation of custom simulation code specialised to the particular model whilst allowing high-level abstractions. The addition of stochastic constructs enables the easy creation and efficient simulation of models that capture the stochastic behaviour underpinning biological and biochemical reaction systems. This was demonstrated through the creation of Markov-formulation ion channel models and a hybrid-stochastic scheme that enables simulation of an integrative cardiac cell model with stochastic components.

We believe this work includes several novel pieces of research and development, from computational and biological modelling aspects. It provides a flexible environment with many capabilities for modellers to develop and simulate complex models, integrate experimental data and investigate biological phenomena. We identified areas for future work and research, and with our collaborators plan to further extend the DSL, create more complex model frameworks and identify further design patterns for structuring biological models.

Appendix A

Ode Language Details

This appendix collects several supplementary and low-level details regarding the *Ode* DSL, including the interactive console commands, built-in functions, types and unit definitions and rules, and translation semantics. Additionally, Fig. A.1 provides a general overview of the *Ode* DSL implementation for reference.

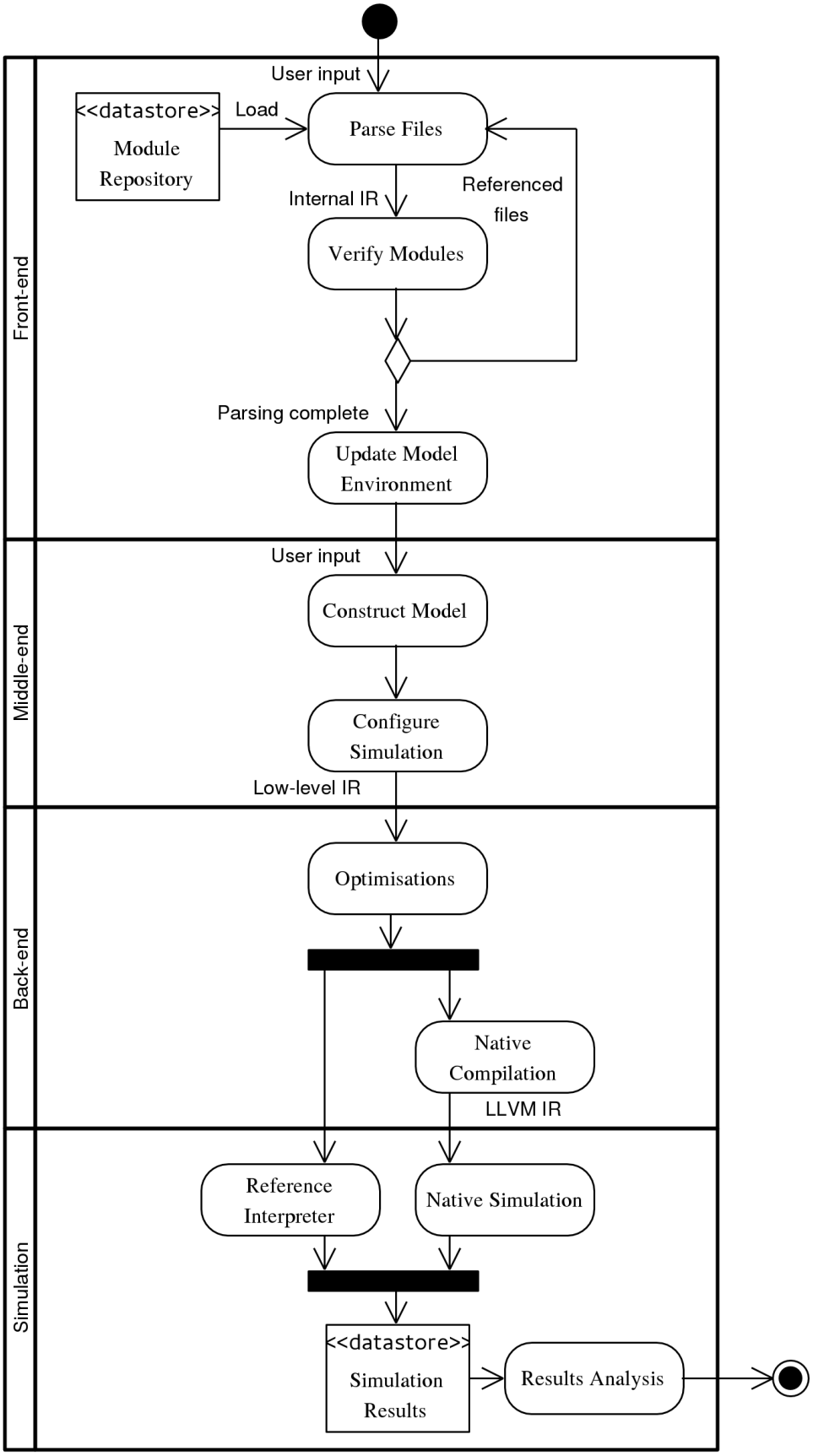


Figure A.1: General system overview of the *Ode* DSL implementation, illustrating major stages in the implementation front-, middle- and back-ends required to convert model source-code into native code for executing a simulation run.

A.1 Console Commands

The following table lists the built-in commands accepted by the *Ode* interactive console for loading, configuring and simulating models. They share a similar format, taking commands from and writing results to the standard input and output streams respectively. They effectively form the external API to the simulation engine that may be scripted to reproduce simulation results and execute batch runs,

Command	Arguments	Description
`:simulate`	m: *String*	Simulate model identified by module `m`
`:addRepo`	f: *String*	Add a directory path to the module repository
`:delRepo`	f: *String*	Delete a directory path from the module repository
`:startTime`	t: *Float*	Initial simulation time t_{start}
`:stopTime`	t: *Float*	Final simulation time t_{stop}
`:timestep`	h: *Float*	Simulation timestep h (represents minimum timestep during adaptive simulation)
`:disableUnits`	*None*	Toggle units-of-measure checking
`:timeUnit`	u: *String*	Set time, t, unit <μs, ms, s, min, hr>
`:maxTimestep`	t: *Float*	Max. timestep, h_{max}, during adaptive simulation
`:maxNumSteps`	i: *Int*	Max. number of adaptive steps to take from current to next time, $t_n \rightarrow t_{n+1}$
`:relError`	e: *Float*	Relative error during adaptive simulation
`:absError`	e: *Float*	Absolute error during adaptive simulation
`:modelType`	m: *String*	Model type when performing adaptive simulation <stiff, nonstiff>
`:period`	m: *Float*	Interval period, t_i, to save model simulation state
`:startOutput`	m: *Float*	Time, t_{out}, from when simulation state is saved
`:output`	s: *String*	Filename for saving simulation results
`:odeSolver`	s: *String*	Set ODE Solver <euler, rk4, adaptive>
`:sdeSolver`	s: *String*	Set SDE Solver <em, projem>
`:backend`	s: *String*	Simulation backend <interpreter, jitcompiler, aotcompiler, objectfile>
`:exeOutput`	f: *String*	Filename for compiled simulation executable
`:linker`	s: *String*	Linkage type when compiling <dynamic, static>
`:disableExecute`	*None*	Toggle auto-execution of simulation
`:mathModel`	s: *String*	Set math model <strict, fast>
`:mathLib`	s: *String*	Set math library <gnu, amd, intel>
`:vecMath`	*None*	Toggle *vecmath* optimisation
`:optimise`	*None*	Toggle *Ode* and LLVM optimisations
`:shortCircuit`	*None*	Toggle short-circuiting of boolean operators
`:powerExpan`	*None*	Toggle expansion of `pow()` calls
`:clear`	*None*	Clear module environment and reset to defaults
`:help`	*None*	Print list of available commands
`:quit`	*None*	Exit system

A.2 Operators and Standard Library

A.2.1 Built-in Operators

The following table lists the basic operators in the DSL, they observe expected operator precedence rules,

Operators	Description
`+ - * / %`	Numerical operators
`== != > >= < <=`	Relational operators
`&& || ! and or not`	Logical operators
`-`	Negation

A.2.2 Built-in Terms and Functions

The following table lists the built-in language terms,

Terms	Type	Description
`time`	`Float u`	The current simulation time, t_{cur}, in the (user-defined) time unit `u`
`wiener`	`Float`	A fresh realisation of a Wiener value at t_{cur}, a dimensionless, normally-distributed random number
`()`	`Unit`	A Unit value (i.e. none/void)

The following table indicates the built-in mathematical operations, implemented via the C mathematical library[1]. The operations are unit-aware, following the unit-inference rules in Appendix A.5, and include extended unit-checked operations,

[1] A reference may be found at `http://www.gnu.org/software/libc/manual/html_node/Mathematics.html`

Mathematical Functions	Type	Description
`sin cos tan`	`Float → Float`	Trigonometric Functions
`asin acos atan`	`Float → Float`	Inverse Trigonometric Functions
`sinh cosh tanh`	`Float → Float`	Hyperbolic Functions
`asinh acosh atanh`	`Float → Float`	Inverse Hyperbolic Functions
`exp exp2 exp10`	`Float → Float`	Exponents
`log log2 log10 logb`	`Float → Float`	Logarithms
`expm1 log1p`	`Float → Float`	Specialised Exponents and Logarithms
`pow`	`(Float, Float) → Float`	Power Function
`sqrt cbrt`	`Float → Float`	Root Functions
`erf erfc`	`Float → Float`	Error Functions
`gamma lgamma`	`Float → Float`	Gamma Functions
`abs floor ceil round`	`Float u → Float u`	Unit-checked Rounding Functions
`upow`	`(Float u1, Int) → Float u2`	Unit-checked Power Function
`uroot`	`(Float u1, Int) → Float u2`	Unit-checked Root Function

A.2.3 Predefined Types and Units

The following table details the built-in types, used in the type-inference rules specified in Appendix A.5,

Types	Notes
Float	Numerical value, implemented as double-precision FPs
Boolean	Boolean True or False value
Unit	Unit type for the empty Unit value `()`
Function	Function mapping from values of input types to values of output types
Tuple	Aggregate heterogeneous structure of values in fixed order
Record	Aggregate heterogeneous structure of values with named identifiers

The following table lists the built-in SI base units, implicit conversion functions are created for all metric prefixes of these units. For some dimensions, e.g. T, we predefine several base units to ease modeller use,

Base Units	Dimension	Quantity
`m`	L	Length
`cm`	L	Length
`kg`	M	Mass
`s`	T	Time
`min`	T	Time
`hr`	T	Time
`A`	I	Electric Current
`K`	Θ	Thermodynamic Temperature
`mol`	N	Amount Of Substance
`cd`	J	Luminous Intensity

The following table lists the built-in derived units, these particular units are included to aid the unit-checking of cardiac electrophysiological models.

Symbol	Name	Base Units	Quantity
`V`	volt	$kg.m^2.A^{-1}.s^{-3}$	Voltage
`mV`	millivolt	$g.m^2.A^{-1}.s^{-3}$	Voltage
`S`	siemens	$m^{-2}.kg^{-1}.s^3.A^2$	Conductance
`mS`	millisiemens	$m^{-2}.kg^{-1}.s^3.mA.A$	Conductance
`F`	farad	$m^{-2}.kg^{-1}.s^4.A^2$	Capacitance
`uF`	microfarad	$m^{-2}.kg^{-1}.s^4.mA^2$	Capacitance
`uA_area`	ampere	$uA.cm^{-2}$	Current per surface area

Listing A.1 Simplified Haskell datatype illustrating the *Core* IR used to encode *Ode* models.

```
data Id = Id | RecordId

data Expr =
    Var Id (Maybe Id)
    | App Id Expr
    | Abs Id Expr
    | Let [Id] (Expr b) (Expr b)
    | Literal
    | Op Expr
    | If Expr Expr Expr
    | Tuple [Expr]
    | Record (Map Id Expr)
    | Ode Id Expr
    | Sde Id Expr Expr
    | Rre [(Int, Id)] [(Int, Id)] Expr
    | UnitCast Unit

data Literal = Num Double Unit | Boolean Bool | Time | Wiener | Unit

data Op = BasicOp | MathOp

data BasicOp =
      Add | Sub | Mul | Div | Mod
    | LT | LE |GT | GE | EQ | NEQ
    | And | Or | Not | Neg

data MathOp =
      Sin | Cos | Tan
    | ASin | ACos | ATan
    | Exp | Log
    | Pow | Sqrt | Cbrt
    | UPow Int | URoot Int | ...
```

A.3 Intermediate Representations and Datatypes

A.3.1 *Core* Syntax

Listing A.1 presents a simplified form (for space and complexity reasons) of the term language used to represent *Core* IR expressions as a Haskell datatype. The expressions form an extended subset of the lambda-calculus [7].

A.3.2 *Ode* to *Core* Translation Semantics

Fig. A.2 presents semantics for translating a subset of the main *Ode* expression language into *Core* IR terms. We describe a meaningful subset of the *Ode* language that covers the main expressions and keeps the semantics short and simple. The semantics are given in a big-step operational style [119] using a syntax provided in [117]. Within the translation rules, $a_1, a_2, \ldots,$

E-Val	**Trans**⟦ `val` $x = a_1$; a_2 ⟧ = `Let` $x\ a_1\ a_2$
E-Var	**Trans**⟦ x ⟧ = `Var` x
E-Comp	**Trans**⟦ `component` $f(x_1, \ldots, x_n)\ a$ ⟧ = `Let` f (`Abs` y (`Let` $[x_1, \ldots, x_n]\ y\ a$))
E-CompCall	**Trans**⟦ $f(a_1, \ldots, a_n)$ ⟧ = `Let` t (`Tuple` $[a_1, \ldots, a_n]$) (`App` $f\ t$)
E-BuiltIns	**Trans**⟦ $a_1 * a_2$ ⟧ = `Op Mul` (`Tuple` $[a_1, a_2]$)
	Trans⟦ $a_1 \parallel a_2$ ⟧ = `Op Or` (`Tuple` $[a_1, a_2]$)
	Trans⟦ `sin` a ⟧ = `Op Sin` a
	...
E-Literals	**Trans**⟦ `1` ⟧ = `Number` 1
	Trans⟦ `True` ⟧ = `Bool` *True*
	...
	Trans⟦ () ⟧ = `Unit`
	Trans⟦ `time` ⟧ = `Time`
	Trans⟦ `wiener` ⟧ = `Wiener`
E-Piecewise	**Trans**⟦ `piecewise`$\{a_{C1} : a_{T1}, \ldots, a_{Cn} : a_{Tn}, \texttt{default} : a_{\text{def}}\}$ ⟧ = `If` $a_{C1}\ a_{T1}$ (`If` ... `If` $a_{Cn}\ a_{Tn}\ a_{\text{def}}$)
E-Tuple	**Trans**⟦ $(a_1, \ldots, a_n)$ ⟧ = `Tuple` $[a_1, \ldots, a_n]$
E-Record	**Trans**⟦ $\{x_1 : a_1, \ldots, x_n : a_n\}$ ⟧ = `Record` $\{x_1 : a_1, \ldots, x_n : a_n\}$
E-ODE	**Trans**⟦ `ode` $\{\texttt{initVal} : x_{\text{init}}\} = a_{\text{delta}}$ ⟧ = `Ode` $x_{\text{init}}\ a_{\text{delta}}$
E-SDE	**Trans**⟦ `sde` $\{\texttt{initVal} : x_{\text{init}}, \texttt{diffusion} : a_{\text{diff}}\} = a_{\text{delta}}$ ⟧ = `Sde` $x_{\text{init}}\ a_{\text{diff}}\ a_{\text{delta}}$
E-Reaction	**Trans**⟦ `reaction` $\{\texttt{rate} : a_{\text{rate}}\} = int_1\ x_1, \ldots, int_n\ x_n -> int_1\ x_1, \ldots, int_m\ x_m$ ⟧ = `RRE` $[(int_1, x_1), \ldots, (int_n, x_n)]\ [(int_1, x_1), \ldots, (int_m, x_m)]\ a_{rate}$

Figure A.2: Translation semantics defining the conversion of a subset of *Ode* expressions into *Core* IR terms.

are arbitrary evaluated expressions, $x_1, x_2, \ldots$ are identifiers, $y_1, y_2, \ldots$ are *fresh*, unique identifiers, and **Trans**⟦ e ⟧ is the translation function that maps from the set of *Ode* syntactical expressions to *Core* terms.

A.4 Type Constraint Rules

The type constraint rules for expressions are provided on the following pages. The rules follow a syntax given in [119] where $\Gamma \vdash t : T \mid C$ reads as 'the term t has type T under the assumptions of the type environment Γ whenever the type constraints C are satisfied'. Additionally we use the notation X to represent a *fresh* type variable unique over the process.

In general only a single type of constraint is available in the type system, defining an equality constraint between two types/type variables, using the syntax $T_1 = T_2$. However a minor addition enables subtyping on record fields, where a record with additional fields R_2 is said to be a subtype of R_1 if the fields of R_1 are a subset of R_2 with the same types, denoted as $R_1 >: R_2$. This is used to collate and constrain all usages of a record's fields within a model.

[1] Also applies to `Sub`, `Mod`, `Mul`, `Div` (seen in rules on next page).
[2] Also applies to `LT`, `LE`, `GE`, `EQ`, `NEQ`.
[3] Also applies to `Or`.
[4] Also applies to all math operations of type `Float → Float`.
[5] Also applies to all math operations of type `(Float, Float) → Float`.

T-Literals

$$\Gamma \vdash \texttt{Number} : \mathit{Float} \mid \emptyset \qquad \Gamma \vdash \texttt{Boolean} : \mathit{Bool} \mid \emptyset \qquad \Gamma \vdash \texttt{Unit} : \mathit{Unit} \mid \emptyset$$

$$\Gamma \vdash \texttt{Time} : \mathit{Float} \mid \emptyset \qquad \Gamma \vdash \texttt{Wiener} : \mathit{Float} \mid \emptyset$$

T-Let

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2 \mid C_2 \quad C' = C_1 \cup C_2}{\Gamma \vdash \texttt{Let}\ x\ t_1\ t_2 : T_2 \mid C'}$$

T-Let2 (Tuple Unpacking)

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \quad \Gamma, x_1 : X_1, \ldots, x_n : X_n \vdash t_2 : T_2 \mid C_2 \quad C' = C_1 \cup C_2 \cup \{T_1 = \mathit{Tuple}(X_1, \ldots, X_n)\}}{\Gamma \vdash \texttt{Let}\ (x_1, \ldots, x_n)\ t_1\ t_2 : T_2 \mid C'}$$

T-Var

$$\frac{x : T \in \Gamma}{\Gamma \vdash \texttt{Var}\ x : T \mid C}$$

T-Var2 (Record Selection)

$$\frac{\mathrm{x} : T \in \Gamma \qquad C = \{\mathit{Record}(y : X) >: T\}}{\Gamma \vdash \texttt{Var}\ x\ y : X \mid C}$$

T-Abs

$$\frac{\Gamma, x : X \vdash t_{body} : T_{body} \mid C}{\Gamma \vdash \texttt{Abs}\ x\ t_{body} : X \to T_{body} \mid C}$$

T-App

$$\frac{\Gamma \vdash t : T \mid C \qquad x : T_f \in \Gamma \quad C' = C \cup \{T_f = T \to X\}}{\Gamma \vdash \texttt{App}\ x\ t : X \mid C'}$$

T-If

$$\frac{\Gamma \vdash t_C : T_C \mid C_C \quad \Gamma \vdash t_T : T_T \mid C_T \qquad \Gamma \vdash t_F : T_F \mid C_F \quad C' = C_C \cup C_T \cup C_F \cup \{T_C = \mathit{Bool}; T_T = T_F\}}{\Gamma \vdash \texttt{If}\ t_C\ t_T\ t_F : T_F \mid C'}$$

T-Tuple

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \ldots \Gamma \vdash t_n : T_n \mid C_n \quad C' = \bigcup_{i=1}^{n} C_i}{\Gamma \vdash \texttt{Tuple}\ (t_1, \ldots, t_n) : \mathit{Tuple}(T_1 \ldots T_n) \mid C'}$$

T-Record

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \ldots \Gamma \vdash t_n : T_n \mid C_n \quad C' = \bigcup_{i=1}^{n} C_i}{\Gamma \vdash \texttt{Record}\ (t_1, \ldots, t_n) : \mathit{Record}(T_1, \ldots, T_n) \mid C'}$$

T-BasicOps

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \qquad \Gamma \vdash t_2 : T_2 \mid C_2 \qquad C' = C_1 \cup C_2 \cup \{T_1, T_2 = Float\}}{\Gamma \vdash \texttt{Add}^1\ t_1\ t_2 : Float \mid C'} \qquad \frac{\Gamma \vdash t_1 : T_1 \mid C_1 \qquad \Gamma \vdash t_2 : T_2 \mid C_2 \qquad C' = C_1 \cup C_2 \cup \{T_1, T_2 = Float\}}{\Gamma \vdash \texttt{GT}^2\ t_1\ t_2 : Bool \mid C'}$$

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \qquad \Gamma \vdash t_2 : T_2 \mid C_2 \qquad C' = C_1 \cup C_2 \cup \{T_1, T_2 = Bool\}}{\Gamma \vdash \texttt{And}^3\ t_1\ t_2 : Bool \mid C'}$$

T-MathOps

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \qquad C' = C_1 \cup \{T_1 = Float\}}{\Gamma \vdash \texttt{Sin}^4\ t_1 : Float \mid C'} \qquad \frac{\Gamma \vdash t_1 : T_1 \mid C_1 \qquad \Gamma \vdash t_2 : T_2 \mid C_2 \qquad C' = C_1 \cup C_2 \cup \{T_1, T_2 = Float\}}{\Gamma \vdash \texttt{Pow}^5\ t_1\ t_2 : Float \mid C'}$$

T-ODE

$$\frac{x_{init} : T_{init} \in \Gamma \qquad \Gamma \vdash t_{delta} : T_{delta} \mid C_{delta} \qquad C' = C_{delta} \cup \{T_{init}, T_{delta} = Float\}}{\Gamma \vdash \texttt{ODE}\ x_{init}\ t_{delta} : T_{delta} \mid C'}$$

T-SDE

$$\frac{x_{init} : T_{init} \in \Gamma \qquad \Gamma \vdash t_{delta} : T_{delta} \mid C_{delta} \qquad \Gamma \vdash t_{wiener} : T_{wiener} \mid C_{wiener} \qquad C' = C_{delta} \cup C_{wiener} \cup \{T_{init}, T_{delta}, T_{wiener} = Float\}}{\Gamma \vdash \texttt{SDE}\ x_{init}\ t_{delta}\ t_{wiener} : T_{delta} \mid C'}$$

T-RRE

$$\frac{x_{x1} : T_{x1}, \ldots, x_{xn} : T_{xn} \in \Gamma \qquad x_{y1} : T_{y1}, \ldots, x_{ym} : T_{ym} \in \Gamma \qquad \Gamma \vdash t_{rate} : T_{rate} \mid C_{rate} \qquad C' = C_{rate} \cup \{T_{x1}, \ldots, T_{xn}, T_{y1}, \ldots, T_{ym}, T_{rate} = Float\}}{\Gamma \vdash \texttt{RRE}\ (x_{x1}, \ldots, x_{xn})\ (x_{y1}, \ldots, x_{ym})\ t_{rate} : Unit \mid C'}$$

A.5 Unit Constraint Rules

Unit rules are used in conjunction with the previous type inference rules (see Section 3.3.1.1 and Appendix A.4) to infer the units within expressions. Within the implementation the type and units rules are applied simultaneously, we present them separately for clarity. Expressions without a units rule are unit independent, and are covered by the type inference rules alone; for expressions with both rules, the units rules operate on *Float* values.

The basic unit rules are provided on the following page. The rules utilise a similar notation and form to the type inference rules, where $\diamond$ represents a dimensionless unit. We note that unit rules are not created for the SSA-reaction construction (see Section 6.1.2.2), and that the unit for the `time` term is user-configurable form the console (see Appendix A.1). Whereas for the type rules we utilised a single equality constraint, we introduce several constraint types for unit-inference, where a unit u represents a vector of base units with integer powers, i.e. a base/derived unit, or a unit-variable,

U-Eq $u_1\ u_2$ — similar to the type-equality rule, this constraint declares that 2 units are in the same dimension and equal to each other,

U-SameDim $u_1\ u_2$ — this constraint declares that two units are compatible in that they are the same dimension, but may be of differing units,

U-Sum $u_1\ u_2\ u_3$ — this constraint declares that the powers of two units u_1, u_2 sum to a third unit u_3, e.g. when representing the result from the multiplication of two unit-annotated values.

U-Mul e $u_1\ u_2$ — this constraint declares that the power of a unit u_1 multiplied by the integer e forms u_2, e.g. when representing the result from the raising a unit-annotated value to a power.

[1] Also applies to `Sub`, `Mod` (seen in rules on next page).
[2] Also applies to `LT`, `LE`, `GE`, `EQ`, `NEQ`.
[3] Also applies to `FAbs`, `Floor`, `Ceil`.
[4] Also applies to all dimensionless mathematical operations, e.g. `Cos`, `ATan`, `Exp`, etc.

U-Literals

$$\Gamma \vdash \texttt{Number} : \diamond \mid \emptyset \qquad \Gamma \vdash \texttt{Number}\ u : U_u \mid \emptyset \qquad \Gamma \vdash \texttt{Time} : U_{time} \mid \emptyset$$

$$\Gamma \vdash \texttt{Wiener} : \diamond \mid \emptyset$$

U-BasicOps

$$\frac{\Gamma \vdash t_1 : U_1 \mid C_1 \qquad \Gamma \vdash t_2 : U_2 \mid C_2 \qquad C' = C_1 \cup C_2 \cup \{\textbf{U-Eq}\ U_1\ U_2\}}{\Gamma \vdash \texttt{Add}^1\ t_1\ t_2 : U_2 \mid C'} \qquad \frac{\Gamma \vdash t_1 : U_1 \mid C_1 \qquad \Gamma \vdash t_2 : U_2 \mid C_2 \qquad C' = C_1 \cup C_2 \cup \{\textbf{U-Sum}\ U_1\ U_2\ X_3\}}{\Gamma \vdash \texttt{Mul}\ t_1\ t_2 : X_3 \mid C'}$$

$$\frac{\Gamma \vdash t_1 : U_1 \mid C_1 \qquad \Gamma \vdash t_2 : U_2 \mid C_2 \qquad C' = C_1 \cup C_2 \cup \{\textbf{U-Sum}\ X_3\ U_2\ U_1\}}{\Gamma \vdash \texttt{Div}\ t_1\ t_2 : X_3 \mid C'} \qquad \frac{\Gamma \vdash t_1 : U_1 \mid C_1 \qquad \Gamma \vdash t_2 : U_2 \mid C_2 \qquad C' = C_1 \cup C_2 \cup \{\textbf{U-Eq}\ U_1\ U_2\}}{\Gamma \vdash \texttt{GT}^2\ t_1\ t_2 : \mathit{Bool} \mid C'}$$

U-MathOps

$$\frac{\Gamma \vdash t : U \mid C}{\Gamma \vdash \texttt{Round}^3\ t : U \mid C} \qquad \frac{\Gamma \vdash t : U \mid C \qquad C' = C \cup \{\textbf{U-Mul}\ e\ U\ X\}}{\Gamma \vdash \texttt{UPow}\ e\ t : X \mid C'} \qquad \frac{\Gamma \vdash t : U \mid C \qquad C' = C \cup \{\textbf{U-Mul}\ e\ X\ U\}}{\Gamma \vdash \texttt{URoot}\ e\ t : X \mid C'}$$

$$\frac{\Gamma \vdash t : U \mid C \qquad C' = C \cup \{\textbf{U-Eq}\ U\ \diamond\}}{\Gamma \vdash \texttt{Sin}^4\ t : \diamond \mid C'}$$

U-UnitCast

$$\frac{\Gamma \vdash t_1 : U_1 \mid C_1 \qquad C' = C_1 \cup \{\textbf{U-SameDim}\ U_1\ U_2\}}{\Gamma \vdash \texttt{UnitCast}\ t_1\ U_2 : U_2 \mid C'}$$

U-ODE

$$\frac{x_{init} : U_{init} \in \Gamma \qquad \Gamma \vdash t_{delta} : U_{delta} \mid C_{delta} \qquad C' = C_{delta} \cup \{\textbf{U-Sum}\ U_{delta}\ U_{time}\ U_{init}\}}{\Gamma \vdash \texttt{ODE}\ x_{init}\ t_{delta} : U_{delta} \mid C'}$$

U-SDE

$$\frac{x_{init} : U_{init} \in \Gamma \qquad \Gamma \vdash t_{delta} : U_{delta} \mid C_{delta} \qquad C' = C_{delta} \cup \{\textbf{U-Sum}\ U_{delta}\ U_{time}\ U_{init}\}}{\Gamma \vdash \texttt{SDE}\ x_{init}\ t_{delta}\ t_{wiener} : U_{delta} \mid C'}$$

A.6 Output File Format

At periodic intervals when simulating a model the system saves the current time t_i and the approximate solution $y(t_i)$ to an output file. This enables results to be analysed off-line with a variety of software packages and distributed to multiple users for independent research. We achieve this using a simple binary format.

The file header consists of a single integer n that indicates the number of stateful values, i.e. size of the system y. This is followed by the results data as an array of doubles stored in row-major order with a column length of $n + 1$, where each row specifies the current time t_i and approximate solution $y(t_i)$ at that point. This data can be read sequentially on a row-by-row basis, or *memory-mapped* directly for full analysis. Readers and writers have been created for C, MATLAB, and Python, in the latter case forming the base for a benchmarking and analysis framework. We may investigate alternate formats in the future, such as NumPy NPY format[2], NetCDF[3] or HDF5[4].

Element	Type	Description
`n`	`uint64_t`	Number of system state values
`data`	`[][n+1]`	Solution array, containing the time and system state at regular intervals t_i

[2] `http://docs.scipy.org`
[3] `http://www.unidata.ucar.edu/software/netcdf/`
[4] `http://www.hdfgroup.org/`

Listing A.2 Simplified Haskell datatype used to represent modules and the module language.

```
data ModuleCmd = ModDef ModName Module | ModImport ModName

data Module =
  - Literal/Standalone Module
  LiteralMod ModuleBody
  - Parameterised Module
  | FunctorMod ModArgs ModuleBody
  - Parameterised Module Application
  | AppMod ModName [Module]
  - Value Reference to a module
  | VarMod ModName
  - Partially evaluated module application
  | RefMod ModName TempModuleBody LocalModEnv
```

A.7 Module System Datatypes and Evaluation Semantics

Listing A.2 presents a simplified (for space and complexity reasons) Haskell datatype representing the module system term language. The table below describes the main terms, they are similar to the expression terms in the lambda-calculus but operate at the module level [7],

Term	Description
`ModDef`	Creates a module within the module environment with the given name
`ModImport`	Imports a module from the repository into the module environment
`LiteralMod`	A standard module definition, containing a list of *Ode* declarations in the `ModuleBody` structure
`FunctorMod`	A parameterised module definition, containing a list of *Ode* declarations and the input module arguments with inferred signature requirements
`AppMod`	Application of a functor with concrete module parameters, evaluation eventually results in a `LiteralMod` standard module
`VarMod`	A variable that points to a module defined elsewhere in the environment
`RefMod`	An intermediate structure that arises from the partial evaluation of functor applications, holding a dummy module implementing the correct signature and a local environment of modules to use when application completes in the backend

The most important function is the application and evaluation of functors within the module environment, resulting in the creation of a new module. Although internally the interpreter only partially evaluates functor applications at this point, creating references to yet-to-be-fully applied modules, held in the `RefMod` structure. This is such that applied modules may be fully inlined within the backend whilst sharing common definitions.

Interpretation of these terms occurs at runtime against the global module environment of

E-ModDef	**Eval**⟦ `ModDef` $x\ a$ ⟧ $\rho = \rho[x \mapsto a]$
E-ModImport	**Eval**⟦ `ModImport` x ⟧ $\rho = \rho[x \mapsto a]$
	(where a is a module retrieved from module repos)
E-LiteralMod	**Eval**⟦ `LiteralMod` m ⟧ ρ = `LiteralMod` m
E-FunctorMod	**Eval**⟦ `FunctorMod` $(x_1, \ldots, x_n)\ n$ ⟧ $\rho = ((x_1, \ldots, x_n), n)$
E-AppMod	**Eval**⟦ `AppMod` $f\ (a_1, \ldots, a_n)$ ⟧ ρ =
	let $((x_1, \ldots, x_n), n) = \rho\, f$ in
	let m' = **PEval**⟦ n ⟧ $\rho\ [x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$ in
	`RefMod` $f\ m' \gamma [x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$
E-VarMod	**Eval**⟦ `VarMod` x ⟧ $\rho = \rho\, x$
E-RefMod	**Eval**⟦ `RefMod` $f\ m'\ \gamma$ ⟧ ρ = `LiteralMod` m'
E-ModBody	**Eval**⟦ n ⟧ $\rho = m$
	(module body evaluated according to Appendix A.3.2)

Figure A.3: Evaluation semantics for the module system, evaluation occurs prior to simulation during model compilation.

imported modules from all repositories. The interpretive semantics are presented in Fig. A.3, where ρ represents the module environment; $a_1, a_2, \ldots$ are arbitrary evaluated modules; $x_1, x_2, \ldots$ and f are module identifiers; m and n are evaluated and unevaluated module bodies respectively; and **Eval**⟦ e ⟧ is the evaluation function that maps from the set of module expressions to module semantics. We also define a second evaluation function, **PEval**⟦ n ⟧, that partially evaluates a module body and returns dummy evaluated module body, m', exposing the correct module signature. Finally, γ represents an module environment used within a `RefMod` to hold locally required modules to complete evaluation of a functor.

A.8 Module System Type Rules

The module type rules are provided below, again following a syntax seen previously for *Ode* expressions. Within the rules, $\Gamma \vdash a : T \mid C$ reads as the module a has a type signature T (taken as the collection of the types of all visible terms within the module body m), under the module type environment Γ whenever the type constraints C are satisfied. Additionally we use the notation X to represent a *fresh* module type variable that is unique over the process. We use the same syntax to represent the module terms as in the evaluation rules above.

The module type system has a single constraint type, used between two functor signatures with the syntax $T1_{in} \to T1_{out} = T2_{in} \to T2_{out}$, seen in the `T-AppMod` rule. This constraint defines a structural subtyping relationship between each pair of input modules, where each module in $T2_{in}$ can be a subtype of its equivalent in $T1_{in}$ (see Section 4.3.1), and an equality relationship between the output modules $T1_{out}$ and $T2_{out}$.

T-ModDef

$$\frac{\Gamma \vdash a : T \mid C \qquad \Gamma' = \Gamma, \mathrm{x} : T}{\Gamma' \vdash \texttt{ModDef}\, x\, a : T \mid C}$$

T-ModImport

$$\frac{x : T \in \Gamma \qquad \Gamma' = \Gamma, \mathrm{y} : T}{\Gamma' \vdash \texttt{ModImport}\, x\, y : T \mid \emptyset}$$

T-LiteralMod

$$\frac{\Gamma \vdash m : T \mid \emptyset}{\Gamma \vdash \texttt{LiteralMod}\, m : T \mid \emptyset}$$

T-FunctorMod

$$\frac{\Gamma, x_1 : X_1, \ldots, x_n : X_n \vdash m : T \mid C}{\Gamma \vdash \texttt{FunctorMod}\, (x_1, \ldots, x_n)\, m : (X_1, \ldots, X_n) \to T \mid C}$$

T-AppMod

$$\frac{m_1 : T_1 \in \Gamma \ldots m_n : T_n \in \Gamma \qquad f : T_f \in \Gamma \qquad C = \{T_f = (T_1, \ldots, T_n) \to X\}}{\Gamma \vdash \texttt{AppMod}\, f\, (m_1, \ldots, m_n) : X \mid C}$$

T-VarMod

$$\frac{x : T \in \Gamma}{\Gamma \vdash \texttt{VarMod}\, x : T \mid C}$$

T-RefMod

$$\frac{\Gamma \vdash m' : T \mid \emptyset}{\Gamma \vdash \texttt{RefMod}\, f\, m' \gamma : T \mid \emptyset}$$

Appendix B

Numerics

This appendix details the numerical systems used by the solvers within the *Ode* implementation. A high-level interface was created within the *Ode* implementation that all solvers must adhere to, allowing substitutability of solvers and easing the introduction of new solvers. The solver implementations utilise custom-code generation of model-specific, low-level bitcode to perform the simulation operations specified in the model in the *CoreFlat* IR. These are calculated with respect to the simulation kernel, which is evaluated upon every timestep to determine derivatives with respect to time for ODEs and SDEs, and the reaction propensities for SSA-reactions (see Section 5.2).

B.1 Ordinary Differential Equations

A differential equation is an equation involving an unknown function and its derivatives. It is called an ordinary differential equation (ODE) if the unknown function depends on only one independent variable, for instance the following unknown function y with independent variable x,

$$\frac{dy}{dx} = 5x + 3.$$

The order of a differential equation is that of the highest derivative appearing in the equation; we only consider 1^{st} order differentials where the independent variable is t, *time*. We utilise y' to represent the first derivative of y with respect to the independent variable under consideration. A differential equation, combined with conditions on the unknown function and its derivatives given at one value of the independent variable, constitutes an initial-value problem [126].

The standard form for a first-order initial-value ODE in the unknown function $y(t)$ is,

$$\begin{aligned} y' &= f(t, y); \\ y(t_0) &= y_0, \end{aligned} \tag{B.1}$$

where the derivative y' appears only on the left side. Many first-order differential equations can be written in standard form by algebraically solving for y' and setting $f(t, y)$ equal to the right side of the resulting equation. The majority of mathematical biological models, including most within our domain and covered in this thesis (see Section 2.1.3), are first-order initial-value problems in standard form.

Complex systems may contain multiple unknown functions and their derivatives. These may be converted into a set of first-order equations dependent on t, for instance,

$$\begin{aligned} w' &= f(t, w, z), \\ z' &= g(t, w, z), \end{aligned}$$

where, in many cases, the derivatives on the left hand side are combined into a single vector $\mathbf{y}$.

The solution to an initial value problem is a function $y(t)$ that solves the differential equation

and satisfies all given subsidiary conditions. This solution may be derived using analytic methods, however with increasing model complexity this approach becomes intractable and we turn to approximate methods, including computational numerical simulation.

B.1.1 Numerical Simulation

Numerical methods generate approximate solutions to initial-value problems at discrete points of the independent variable, i.e. $t_0, t_1, \ldots, t_n$. The difference between any two successive t-values is the step-size h (also termed the timestep) and may be constant or adaptive. The timestep chosen depends on the particular model properties and simulation requirements, and must balance accuracy and computational resources. Smaller values generate a closer approximation whilst requiring more computation; alternatively if h is made too large the approximate solution may not resemble the exact solution at all.

Furthermore, many of the systems we wish to simulate are stiff, whereby certain numerical methods for solving the system become numerically unstable unless the step-size is taken to be extremely small. They include terms that lead to rapid variation in the solution [115]. Stiffness is often observed in large chemical systems containing both slow and fast reactions, as discussed in Section 2.2.6. Different solvers may exhibit differing stability properties that enable the use of larger step-sizes when generating an approximate solution to a stiff system.

The solution at t_n is designated by $y(t_n)$, or simply y_n. From this system state, y_n, and the right-hand side of the equation, $f(t, y)$, a numerical solver may explicitly determine the next value, y_{n+1}, and generate a solution for the system. The method used to determine the next value differs between solvers and affects the computational efficiency, simplicity, accuracy and stability of the scheme. We briefly cover the explicit solvers implemented in *Ode*; a fuller treatment of their definition, behaviour and usage may be found in [115].

B.1.1.1 Forward-Euler

Given a constant step-size h, we define the explicit forward-Euler method as,

$$y_{n+1} = y_n + hy'_n \tag{B.2}$$
$$\text{where } y'_n = f(t_n, y_n).$$

Often the Euler method is not accurate enough, being only of order one, where the order defines the rate of convergence of the approximate solution to the exact solution as h tends to 0 [115]. We often require higher-order methods to generate consistent solutions.

B.1.1.2 4th Order Runge-Kutta

The Runge-Kutta family of solvers are more stable than the forward Euler at higher-orders, allowing the use of larger step-sizes. We implement the explicit 4^{th} Order Runge-Kutta (RK4) with a constant step-size. As can be seen from the following definition, simulation using this solver requires ~4 times the CPU resources of the forward-Euler,

$$
\begin{aligned}
y_{n+1} &= y_n + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \qquad \text{(B.3)}\\
\text{where } k_1 &= hf(t_n, y_n)\\
k_2 &= hf(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1)\\
k_3 &= hf(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2)\\
k_4 &= hf(t_n + h, y_n + k_3).
\end{aligned}
$$

B.1.1.3 Adaptive Solver (CVODE)

An adaptive solver allows h to vary between lower and upper bounds in response to the current system state. This can dramatically improve simulation efficiency whilst introducing a (bounded) error to the approximate solution. It is ideally used when solving stiff systems such as those modelling chemical reaction kinetics.

We implement an adaptive solver in *Ode* using the CVODE package [27]. This is implemented through the object-file backend that links static, CVODE-using, solver code to the model exported by *Ode* FFI (see Appendix C.4). The *Ode* FFI exports functions to set the initial value y_0 and to calculate $y'_n = f(t_n, y_n)$. These are used by the adaptive solver to compute an approximate solution with a varying h that may be recorded at constant intervals t_i.

Simulation parameters for the adaptive solver are set via console commands described in Appendix A.1. These include the maximum and minimum boundaries for h; the absolute and relative error tolerances for the solution in y; the model stiffness; and the number of steps taken internally when iterating from t_i to t_{i+1}.

B.2 Stochastic Differential Equations

We use stochastic differential equations (SDEs) for modelling continuously over the independent variable t where the unknown function, typically termed X, demonstrates stochastic behaviour [83]. An SDE is a differential equation in which one or more terms is a stochastic process, resulting in a solution which is itself a stochastic process. They can be considered similar to an ODE in containing a deterministic element, alongside an additional random white noise term. Several SDE formulations exist, when modelling the Brownian motion that underpins biochemical reactions we may utilise an Itō SDE,

$$\begin{aligned} dX_t &= \mu(t, X_t)\, dt + \sigma(t, X_t)\, dW_t \\ X_0 &= x_0, \end{aligned} \tag{B.4}$$

where μ and σ are termed the drift and diffusion coefficients respectively, and W_t denotes a Wiener process that represents standard Brownian motion [83]. The stochastic process X_t is called a diffusion process and is generally a Markov process. As with ODEs, approximate solutions may be obtained via numerical simulation.

B.2.1 Numerical Simulation

The stochastic extensions to *Ode* in Section 6.1.1 enable the creation of SDEs as in Eq. (B.4) through the construction of `drift`, `diffusion` and `wiener` expressions and terms. The native-code solvers implemented within *Ode* explicitly approximate the next value X_{t+1} from the current state X_t using Eq. (B.4). Currently we have implemented the explicit Euler-Maruyama (EM) scheme to solve SDEs. An alternate method more suited to simulating ion channel states was also developed and will be tested and enabled as part of future work. This scheme derives from the EM and implements projection of X_t to keep the proportion of channels in each state bounded between 0 and 1 during simulation [37, 110].

B.2.1.1 Euler-Maruyama

The explicit Euler-Maruyama (EM) method is a simple generalisation of the forward-Euler method for ODEs (see Appendix B.1.1.1) to SDEs. It generates an approximation to the exact solution X

for Eq. (B.4) by the Markov chain Y simulated over a constant timestep h that partitions $[t_0, t_n]$ into the discrete intervals $[\tau_0, ..., \tau_N]$ [83]. Having set the initial conditions, i.e. $Y_0 = x_0$, we define,

$$Y_{n+1} = Y_n + \mu(\tau_n, Y_n)\Delta t + \sigma(\tau_n, Y_n)\Delta W_n \tag{B.5}$$

$$\text{where } \Delta W_n = W_{\tau_{n+1}} - W_{\tau_n}.$$

The random variables ΔW_n are independent and identically distributed normal random variables with expected value 0 and variance Δt.

B.3 Direct SSA Simulation

The stochastic simulation algorithm (SSA) was discussed in Section 2.2.3 as a exact method for obtaining a trajectory of the system state of a well-stirred solution of chemically reacting species over time. By obtaining multiple trajectories we may generate the probability distribution of this system, rather than directly solve the Chemical Master Equation (CME) that becomes impractical for larger systems (see Section 2.2). As discussed in Section 2.2.3.1 we may model the state changes of a discrete population of Markov-formulation ion channels using chemical reactions, and use the SSA to simulate their behaviour over time.

Using the same notation as in that section, we assume a system of $\mathbf{N}$ chemical species, $\{S_1, \ldots, S_N\}$ that interact through $\mathbf{M}$ reactions, $\{R_1, \ldots, R_M\}$. We consider that the system is confined to a constant volume Ω and is in thermal equilibrium at some constant temperature. We define $X_i(t)$ as the number of molecules of a species S_i in the system at time t. Finally we attempt through simulation to determine the state vector $\mathbf{X}(t) \equiv (X_1(t), \ldots, X_N(t))$ from some initial state $X(t_0) = \mathbf{x}_0$ at time t_0.

Changes in the system state over time are caused by chemical reactions, where each reaction R_j has two properties. The first is a state-change vector $\mathbf{v}_j \equiv (v_{1j}, \ldots, v_{Nj})$, where v_{ij} is the change in the molecular population of S_i caused by one R_j reaction. That is, the occurrence of one R_j reaction will result in a system state change from $\mathbf{x}$ to $\mathbf{x} + \mathbf{v}_j$. The second property is the reaction propensity function a_j, defined as,

$a_j(\mathbf{x})\, dt \triangleq$ the probability, given $\mathbf{X}(t) = \mathbf{x}$, that one R_j reaction will occur somewhere inside Ω in the next infinitesimal time interval $[t, t + dt)$.

The propensity function for a particular reaction R_j is determined by the reaction type (unimolecular or bimolecular), the number of molecules of the species concerned (e.g. x_i, x_k) and a reaction constant c_j, as discussed in [59].

The direct-SSA enables us to simulate the time-evolution of this system and construct an exact realisation of the process $\mathbf{X}(t)$ from some initial state $\mathbf{X}(t_0) = \mathbf{x}_0$ in a discrete, stochastic manner. This is a Monte Carlo procedure that iteratively updates the system by stochastically determining the next reaction R_j to trigger at time τ. The algorithm is briefly described below, where r_1 and r_2 are uniformly distributed random numbers in the unit interval. A more complete

treatment may be found in [59, 60],

1. Set the initial time $t = t_0$ and initial system state $\mathbf{x} = \mathbf{x}_0$

2. At time t with the system in state $\mathbf{x}$, calculate the sum of all reaction propensities $a_0(\mathbf{x})$ as,

$$a_0(\mathbf{x}) = \sum_{j'=1}^{M} a_{j'}(\mathbf{x}),$$

3. Determine the time, τ, to the next reaction as,

$$\tau = \frac{1}{a_0(\mathbf{x})} \ln\left(\frac{1}{r_1}\right),$$

4. Select the index of the next reaction, j, as,

$$j = \text{the smallest integer satisfying } \sum_{j'=1}^{j} a_{j'}(\mathbf{x}) > r_2 a_0(\mathbf{x}),$$

5. Effect the next reaction by updating the state $\mathbf{x}$ according to R_j, i.e. $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{v}_j$, and by increasing the time to the next interval, i.e. $t \leftarrow t + \tau$,

6. Return to Step 2 to repeat the process as necessary.

B.4 Hybrid Simulation

As described in Section 6.1.3, *Ode* supports hybrid simulation of systems containing ODEs and either SSA- or SDE-based subcomponents [111]. The hybrid support is used in Section 6.3 to simulate cardiac models with stochastic ion channels. Hybrid modelling was discussed in Section 2.2.6, and the implemented schemes are straightforward extensions to the individual simulation mechanisms for ODEs/SDEs/SSA previously described in this section.

B.4.1 ODEs & SDEs

For models containing multiple ODEs and SDEs, we utilise the forward-Euler method combined with the Euler-Maruyama to generate an approximate solution $y(t)$ with initial values $y(t_0) = y_0$. We split the system, defining the model state controlled by ODEs as $w(t)$ and that controlled by SDEs as $v(t)$. We calculate $w(t_{n+1})$ and $v(t_{n+1})$ through the respective solvers while using the same step-size h. These independent model states are combined to form the single approximate next state $y(t_{n+1})$ and determine $y(t)$.

B.4.2 ODEs & SSAs

For models containing multiple ODEs and SSA-reactions, we utilise a staggered simulation scheme as implemented in [14, 15, 102]. This employs *operator splitting* to execute independent simulations for the ODEs and SSA-reactions that are synchronised at specified time-intervals. For our cardiac ion channel modelling use-case, the state of the stochastic ion channel advances on a much smaller time-scale than the constant timestep h needed to simulate the ODEs that determine other ionic currents and the membrane voltage. Hence the ODE solver is used to *drive* the simulation, with the SSA running in between the fixed ODE-solver intervals $[t_0, \dots, t_m]$.

We split the system, defining the state controlled by ODEs as $w(t)$ and that controlled by SSAs as $v(t)$. For a given state $y(t_n)$, the SSA is executed for the submodel until t_{n+1}; this provides $v(t_{n+1})$ with respect to a static $w(t_n)$. We then simulate the ODEs utilising the forward Euler to approximate $w(t_{n+1})$, again with respect to the previous $v(t_n)$. At t_{n+1}, $v(t_{n+1})$ and $w(t_{n+1})$ are combined to form the approximate next state $y(t_{n+1})$ and determine $y(t)$.

Appendix C

Implementation Details & Results

This appendix contains several low-level details regarding the *Ode* backend implementation described in Chapter 5 to enable high-performance model simulation. This includes details of the conversion and evaluation semantics for the *CoreFlat* IR, the code-generation strategy via LLVM, a foreign function interface (FFI) for using a model with external systems, e.g adaptive solvers, and details of the *Ode* run-time library used to support simulations. This appendix also provides several tables that contain the raw results data obtained from the implementation simulations and benchmarks.

Listing C.1 Haskell datatype for the *CoreFlat* IR that represents a model prior to simulation.

```
type CoreFlatModel = (InitVals, SimExprs, [SimOp])

-- | Map of initial values
type InitVals = Map Id Double

-- | Simulation Kernel
type SimExprs = OrderedMap Id (Expr, Type)

-- | Computation Expressions
data Expr   = Var
            | Op [Var]
            | If Var SimExprs SimExprs

-- | Atomic, core values
data Var    = VarRef Id
            | TupleRef Id Int
            | Tuple [Var]
            | Num Double | Boolean Bool | Unit | Time | Wiener

-- | Main simulation opersions
data SimOp  = Ode Id Var
            | Sde Id Var Var
            | Rre [(Int, Id)] [(Int, Id)] Var

-- CoreFlat Types
data Type = TFloat | TBool | TUnit | TTuple [Type]
```

C.1 *CoreFlat* Implementation Details

C.1.1 CoreFlat IR Datatype

Listing C.1 presents a simplified datatype that encodes the *CoreFlat* model representation. As described in Section 5.2.2, this is a 3-element tuple (I, E, S) that contains model initial/state values (`InitVals`), a numerical simulation-kernel evaluated on each timestep (`SimExprs`), and simulation operations that model the time evolution of the system (`SimOps`).

C.1.2 Translation Semantics

Fig. C.1 presents translation semantics for the conversion of expressions in the *Core* language into the *CoreFlat* IR. Translation occurs in the *Ode* backend upon a subset of the *Core* IR where the model has already undergone module and component inlining, as described in Section 5.2.2. Initial values are determined through compile-time interpretation of the model AST (see Section 5.1.4), and stored as primitives within the `InitVals` structure.

Within the translation rules, $a_1, a_2, \ldots$, are arbitrary translated expressions; $x_1, x_2, \ldots$, are

E-Let
$\textbf{Trans}[\![\ \texttt{Let}\ x\ a\ e\ ]\!]\ E\ S = \textbf{Trans}[\![e]\!]\ \ E[x \mapsto a]$
$\textbf{Trans}[\![\ \texttt{Let}\ [x_1, \ldots, x_n]\ a\ e\ ]\!]\ E\ S =$
$\quad \textbf{Trans}[\![e]\!]\ \ E[y \mapsto a; x_1 \mapsto (\texttt{TupleRef}\ y\ 1); \ldots; x_n \mapsto (\texttt{TupleRef}\ y\ n)]$

E-Var
$\textbf{Trans}[\![\ \texttt{Var}\ x\ ]\!]\ E\ S = \texttt{VarRef}\ x$

E-Op
$\textbf{Trans}[\![\ \texttt{Op}\ Op\ a\ ]\!]\ E\ S = (\texttt{Op}\ Op\ y)\ \ E[y \mapsto a]$

E-Literals
$\textbf{Trans}[\![\ \texttt{Number}\ 1\ ]\!]\ E\ S = \texttt{Number}\ 1$
$\textbf{Trans}[\![\ \texttt{Bool}\ True\ ]\!]\ E\ S = \texttt{Bool}\ True$
$\ldots$
$\textbf{Trans}[\![\ \texttt{Unit}\ ]\!]\ E\ S = \texttt{Unit}$
$\textbf{Trans}[\![\ \texttt{Time}\ ]\!]\ E\ S = \texttt{Time}$
$\textbf{Trans}[\![\ \texttt{Wiener}\ ]\!]\ E\ S = \texttt{Wiener}$

E-If
$\textbf{Trans}[\![\ \texttt{If}\ a_c\ e_t\ e_f\ ]\!]\ E\ S = \texttt{If}\ y_c\ (\textbf{Trans}[\![e_t]\!])\ (\textbf{Trans}[\![e_f]\!])\ \ E[y_c \mapsto a_c]$

E-Tuple
$\textbf{Trans}[\![\ \texttt{Tuple}\ [a_1, \ldots, a_n]\ ]\!]\ E\ S = \texttt{Tuple}\ [y_1, \ldots, y_n]\ \ E[y_1 \mapsto a_1; \ldots; y_n \mapsto a_n]$

E-Record
$\textbf{Trans}[\![\ \texttt{Record}\ \{x_1 : a_1, \ldots, x_n : a_n\}\ ]\!]\ E\ S = \texttt{Tuple}\ [x_1, \ldots, x_n]\ \ E[x_1 \mapsto a_1; \ldots; x_n \mapsto a_n]$

E-Ode
$\textbf{Trans}[\![\ \texttt{Ode}\ x_{init}\ a_{delta}\ ]\!]\ E\ S = E[y_{delta} \mapsto a_{delta}]\ \ S[\texttt{Ode}\ x_{init}\ y_{delta}]$

E-Sde
$\textbf{Trans}[\![\ \texttt{Sde}\ x_{init}\ a_{drift}\ a_{delta}]\!]\ E\ S =$
$\quad E[y_{delta} \mapsto a_{delta}; y_{drift} \mapsto a_{drift}]\ \ S[\texttt{Sde}\ x_{init}\ y_{drift}\ y_{delta}]$

E-Reaction
$\textbf{Trans}[\![\ \texttt{Rre}\ [(Int, x_1), \ldots, (Int, x_n)]\ [(Int, x_1), \ldots, (Int, x_m)]\ a_{rate}]\!]\ E\ S =$
$\quad E[y_{rate} \mapsto a_{rate}]\ \ S[\texttt{Rre}\ [(Int, x_1), \ldots, (Int, x_n)]\ [(Int, x_1), \ldots, (Int, x_m)]\ y_{rate}]$

Figure C.1: Translation semantics from converting a *Core* model subset into *CoreFlat* terms, including both model expressions in E and simulation operations in S.

identifiers; $y_1, y_2, \ldots$, are *fresh*, unique identifiers; and $\textbf{Trans}[\![\ e\ ]\!]$ is the translation function that maps from the set of *Core* expressions, represented by $e_1, e_2, \ldots$, to *CoreFlat* terms. E represents computational expressions from the simulation-kernel and S a list of simulation operations, i.e. ODEs, SDEs and SSA-reactions, within the *CoreFlat* model representation, modelling the `SimExprs` and `SimOps` structures seen in Listing C.1.

C.1.3 Evaluation Simulation Semantics

Fig. C.2 presents interpretative semantics for evaluating the simulation-kernel of a *CoreFlat* model, represented by the `SimExprs` structure in Listing C.1.

Within the evaluation rules, ρ represents the run-time environment; $a_1, a_2, \ldots$, are arbitrary evaluated expressions; $x_1, x_2, \ldots$ are identifiers; and $\textbf{Eval}[\![\ e\ ]\!]$ is the evaluation function that maps from the set of syntactical expressions, represented by $e_1, e_2, \ldots$, to semantic operations. As with the above translation semantics, E represents the current list of named expressions from the

E-Let	$\textbf{Eval}[\![\ E\{x : a\}\ ]\!]\ \rho = \rho[x \mapsto a]$
E-Var	$\textbf{Eval}[\![\ \texttt{VarRef}\ x\ ]\!]\ \rho = \rho\ x$
E-Op	$\textbf{Eval}[\![\ \texttt{Op Mul}\ [a, b]\ ]\!]\ \rho = a \times b$
	$\textbf{Eval}[\![\ \texttt{Op Or}\ [a, b]\ ]\!]\ \rho = a \vee b$
	$\textbf{Eval}[\![\ \texttt{Op Sin}\ a\ ]\!]\ \rho = \sin a$
	…
E-Literals	$\textbf{Eval}[\![\ \texttt{Number}\ 1\ ]\!]\ \rho = 1$
	$\textbf{Eval}[\![\ \texttt{Bool}\ \mathit{True}\ ]\!]\ \rho = \text{True}$
	…
	$\textbf{Eval}[\![\ \texttt{Unit}\ ]\!]\ \rho = ()$
	$\textbf{Eval}[\![\ \texttt{Time}\ ]\!]\ \rho = t_{cur}$
	$\textbf{Eval}[\![\ \texttt{Wiener}\ ]\!]\ \rho = N(0, 1)$
E-If	$\textbf{Eval}[\![\ \texttt{If}\ x\ E_t\ E_f\ ]\!]\ \rho = \text{let } \rho' = \rho \text{ in}$
	$\quad \text{if } \rho\ x = \text{True then } \textbf{Eval}[\![E_t]\!]\rho' \text{ else } \textbf{Eval}[\![E_f]\!]\rho'$
E-Tuple	$\textbf{Eval}[\![\ \texttt{Tuple}\ [a_1, \ldots, a_n]\ ]\!]\ \rho = \{1 : a_1, \ldots, n : a_n\}$
E-TupleRef	$\textbf{Eval}[\![\ \texttt{TupleRef}\ x\ n\ ]\!]\ \rho = (\rho\ x).n$

Figure C.2: Interpretative simulation semantics for the *CoreFlat* simulation-kernel.

simulation-kernel that we iterate over in order.

The *Ode* interpreter (see Section 5.2.3) implements these semantics — on each simulation timestep, the time of which is represented by t_{cur}, the run-time environment ρ is populated with the current model state and the simulation-kernel is evaluated. Model simulation operations, held in `SimOps` in Listing C.1, are then performed with respect to the newly-evaluated environment ρ. These operations update the model state, stored in `InitVals`, according to the simulation-specified numerical solvers, as described in Appendix B.

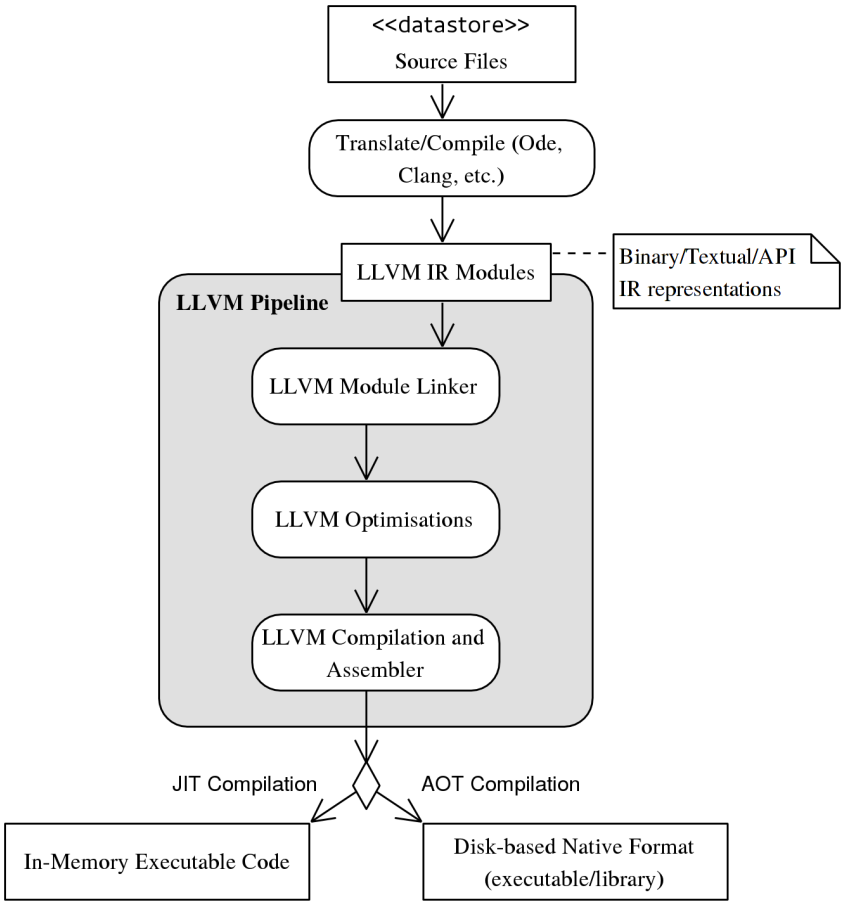


Figure C.3: The LLVM compilation pipeline. Source-files in several languages are translated into LLVM bitcode modules, e.g. when using Clang compiler for C/C++ code. LLVM modules are linked together, undergo many advanced low-level optimisations, and are finally converted to native machine-code, either in memory or to disk.

C.2 Code Generation

C.2.1 LLVM Introduction

To perform code-generation the compiler utilises the LLVM compiler framework via a typed intermediate assembly format termed *LLVM bitcode* [86]. The general structure and processes involved in using the LLVM system is depicted in Fig. C.3. Listing C.2 presents a fragment of the bitcode language that illustrates its basic structure, as seen it resembles a generic CPU assembly syntax.

LLVM presents a low-level, typed, 'virtual machine' that may be used in preference to generating native assembly/machine-code directly. A range of system-level tools exist for the LLVM platform, such as optimisers and linkers, forming a complete code-generation infrastructure.

LLVM's computational model consists of a CPU-level virtual machine with an infinite

Listing C.2 Sample of the LLVM assembly syntax. This module performs a increment operation using the defined `inc` function, prints 'hello world', and successfully terminates.

```
; global definitions within the module
target triple = "x86_64-redhat-linux-gnu"
declare i32 @puts(i8*)
@.str = private unnamed_addr constant [13 x i8]
  c"hello world\0A\00", align 1

; main function
; allocates a variable, stores 1+1, prints "hello world", then exits
define i32 @main() nounwind uwtable {
entryBB:
  %0 = alloca double, align 8
  store double 0.0, double* %0, align 8
  %1 = call double @inc(double 1.0)
  store double %1, double* %0, align 8
  %2 = call i32 (i8*)*
    @puts(i8* getelementptr inbounds ([13 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

; function to increment a double
define double @inc(double %in) readnone nounwind {
entryBB:
  %res = fadd double 1.0, %in
  ret double %res
}
```

supply of single-assignment registers, known as static single assignment (SSA[1]) form [3, 87]. Primitive mathematical operations can be performed on values in these registers, with the results stored in new, immutable registers. Mutability is provided through a parallel load-store memory architecture whose values may be overwritten. Aggregate structures, including structures, arrays, and vectors, can be constructed from the basic values. LLVM provides a strong, static type-system used to check bitcode correctness and aid optimisations.

As is common in compiler IRs, LLVM instructions are ordered sequentially within basic-blocks (BB) that have a single, well-defined, entry- and exit-point. Basic-blocks are collected into functions that in turn comprise an LLVM module. LLVM modules may be created in-memory through an API or stored on disk in binary/textual forms, as demonstrated in Listing C.2. These modules can be linked and compiled to native code for direct execution (JIT compilation) or serialised to disk (AOT compilation).

LLVM bitcode is optimised during compilation, making use of extensive static-code analysis including dataflow analysis. This data underlies many LLVM optimisations, and can be used to

[1]Not to be confused with the stochastic simulation algorithm (SSA) defined elsewhere within this thesis.

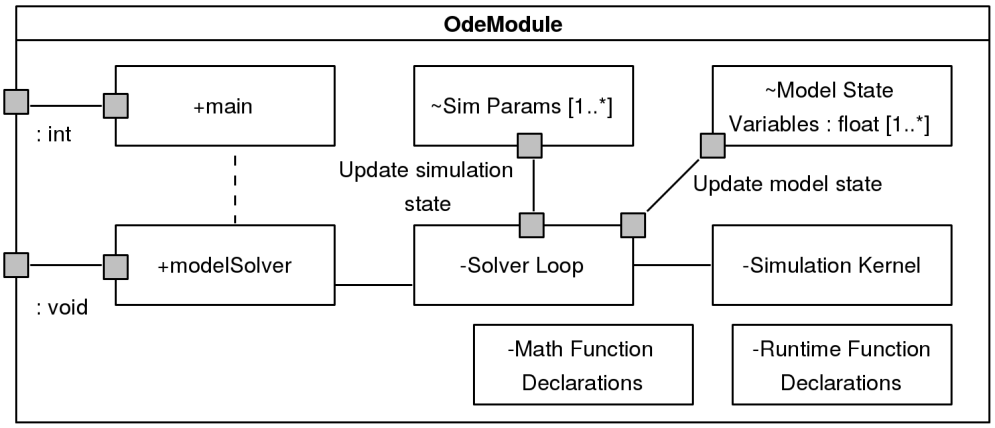


Figure C.4: Structure of the generated LLVM module representing the *Ode* model-simulation code. This comprises of a `modelSolver` function that contains the custom solver loop and simulation-kernel, alongside constant simulation parameters and model state variables.

develop domain-specific optimisations that can in turn be applied to our generated model code.

LLVM is widely used in industry, including the popular Clang C/C++ compiler[2] and several OpenGL/OpenCL implementations. However it is not a general solution that enables high-performance code-generation in all situations. It is ideally suited to statically-typed languages with semantics similar to C. To obtain suitable results a language implementation must be carefully designed to generate as efficient bitcode as possible.

C.2.2 Bitcode Structure

We now detail the low-level structure of the generated bitcode representing a simulation. This is generated from a *CoreFlat* model comprising several elements — the initial values, the numerical simulation-kernel and the simulation operations (see Section 5.2). These are combined with the chosen solver algorithm and the simulation parameters/configuration. This must all be expressed in an optimal form within the LLVM module, Fig. C.4 illustrates this structure using a UML component diagram.

The generated module has a single entry-point, termed `modelSolver`, within which all model simulation code is generated as sequential code. The implementation generates all initial values as scalar, double values globally within the module to aid efficiency. Similarly the implementation generates all static simulation parameters as read-only primitives at the global level.

The simulation kernel, which processes the model computational code with respect to the

[2] `http://clang.llvm.org/`

current model state and time, is generated as an independent 'blob' of bitcode with a specified interface for interaction with the specific solver. The solvers themselves implement a common interface to enable easy integration of alternate solvers for specific models and use-cases. The solver implements the selected simulation algorithm and is custom-generated at compile-time with respect to the model and simulation parameters, and is tightly coupled to the model computation code. It embeds the model code, executes the simulation operations and controls the simulation according to the defined simulation parameters.

C.2.3 Low-level Model Simulation

On starting a simulation control passes to the `modelSolver` function which has three distinct phases: setting up the simulation; running the specific model-simulation code for the simulation; and cleanly shutting down the simulation. The flow of control through these subsystems is depicted using the UML sequence diagram in Fig. C.5; this is further expanded in Fig. C.6 presenting an annotated callgraph of the *Solver Loop*.

During simulation setup the runtime library initialises the solver environment, involving auxiliary tasks such as seeding the random number generator and allocating handles and buffers for file output. The global initial values are assigned their the pre-calculated values (see Section 5.1.4) and this initial model simulation state is written to disk using the associated library functions.

Control then passes to one of the pluggable model solver backends to start the simulation. Solvers are responsible for running the *Solver Loop* that updates the time and other simulation parameters. A solver will then evaluate the simulation-kernel code segment with respect to the desired time and state values. The solver computes the simulation operations, i.e. the ODEs within the model. This utilises the time and state values and just evaluated simulation-kernel environment with the chosen solver algorithm to determine and update the system state. Finally it saves the simulation state if required, updates the simulation parameters and determines if further simulation is required.

Upon finishing simulation the runtime library shuts down the simulation environment, freeing all resources, and exits the top-level function, terminating the simulation.

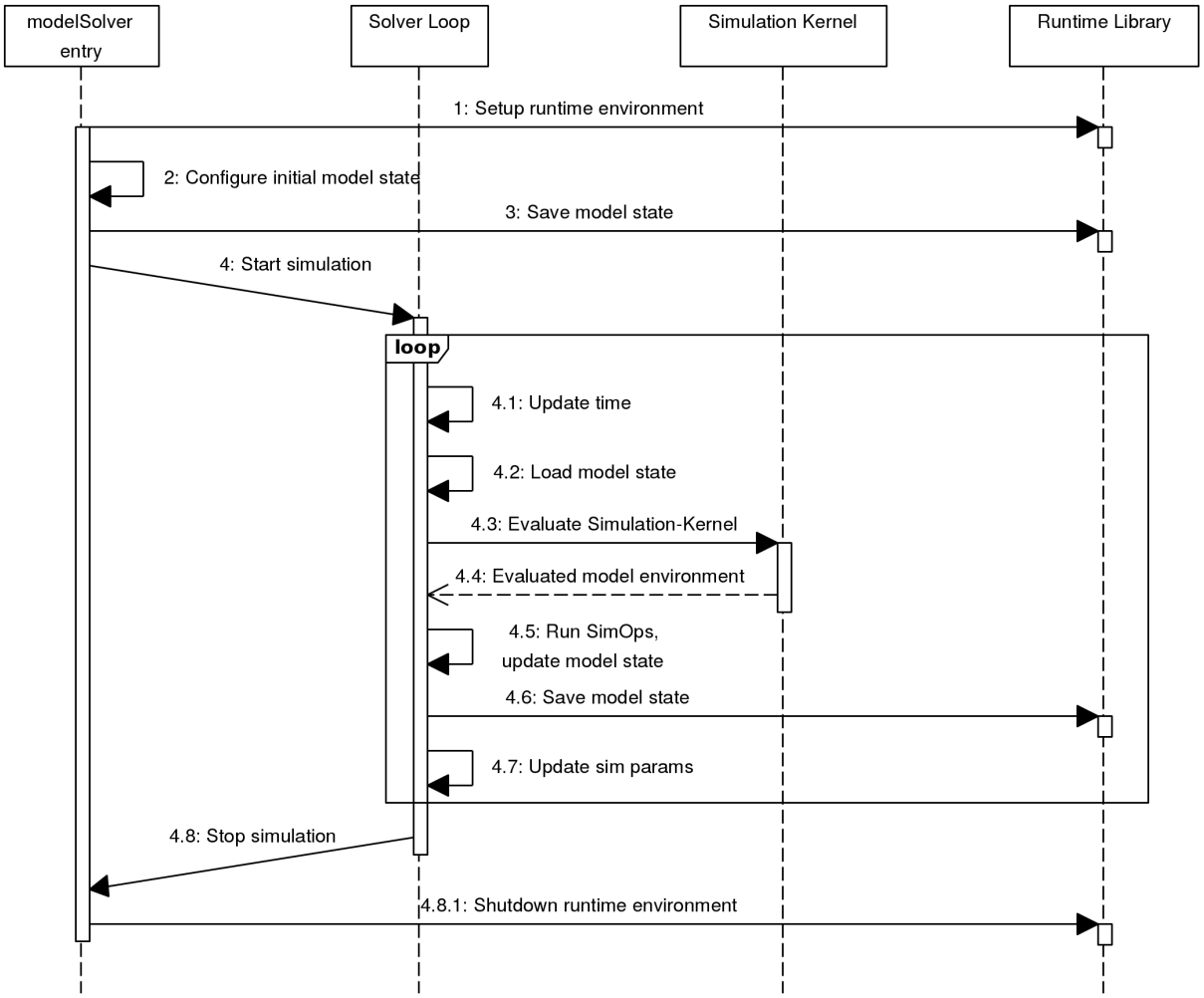


Figure C.5: UML sequence diagram indicating low-level execution of an *Ode* generated simulation, using the module component defined in Fig. C.4. The `solverLoop` block is responsible for running the particular solver, calling into the *Simulation Kernel* that evaluates the *Ode*-generated model code on each simulation timestep.

C.2.4 Model Bitcode Generation

We now detail the process of generating efficient LLVM bitcode from the low-level *CoreFlat* IR in a form amenable to optimisation and high-performance simulation. This uses the basic module structure described in Appendix C.2.2, where the combined model-simulation code is comprised of several sections. Initial values have already been evaluated during compilation and are stored as doubles within the bitcode. Simulation parameters are both encoded as static constants globally within the module and directly embedded within the solver where required.

Solvers are generated during model compilation via custom code-generation of LLVM-bitcode, resulting in model-specific solver loops, e.g. with regards to state size and simulation parameters. To ease code-generation we developed several macros that simplify the creation of common

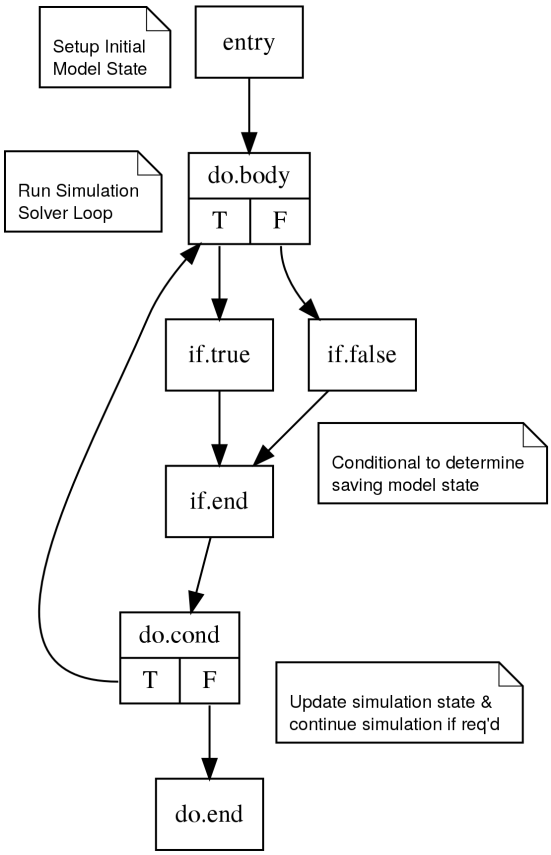


Figure C.6: Annotated callgraph illustrating the control-flow within the `modelSolver` function, depicting the simulation loop and saving the model state. This represents an alternate view of the control-flow to Fig. C.5, and was generated from the LLVM model code in Listing C.4.

high-level structures such as if-statements and while-loops in LLVM bitcode. Model simulation operations, e.g. ODEs/SDEs/RREs, are processed within this solver loop to implement the desired solver algorithm. Again this is performed by code-generated model-specific LLVM-bitcode that implements the required mathematical operations, see Appendix B for details on these.

C.2.4.1 *CoreFlat* Simulation IR to LLVM Bitcode

Conversion of the computational simulation-kernel in *CoreFlat* IR to LLVM bitcode proceeds according to the evaluation semantics presented Appendix C.1. The approach is similar to evaluation in the reference interpreter except we translate operators into equivalent bitcode instructions with the same semantics. The IR for the simulation-kernel is based on ANF, which has a simple translation into the SSA-form used by LLVM [6, 86]. Furthermore the *CoreFlat* IR types map simply onto the LLVM type system, Table C.1 presents the mapping for each type.

During conversion we traverse over the top-level expressions within the simulation-kernel, translating expressions into a fresh basic block. The highly-sequential nature of this IR eases

***CoreFlat* Types**	**LLVM Types**
Float	`double`
Boolean	`i1` (1-bit integer)
Unit	`i1` set to 0
Tuple	LLVM `struct`

Table C.1: Mapping from *CoreFlat* to LLVM types.

Listing C.3 Verhulst model/logistic equation for population growth in *Ode*, used as a simple ODE to demonstrate LLVM bitcode generation in Listing C.4.

```
module PopGrowth {
    init eta = 1
    val k = 1
    val mu = 20
    ode { initVal : eta } = k * (1 - eta / mu ) * eta
}
```

translation complexity. Value assignments are converted into LLVM static assignments, and references likewise translate straight across. Numerical values are translated into doubles, booleans into 1-bit integers and the unit value similarly. Basic mathematical operators are translated into primitive FP mathematical operations or integer-based relational and logical operations, while more complex functions call into the `libm` math library. Tuple creation and usage is translated into operations utilising LLVM aggregate *structs* that exhibit similar semantics.

Conditional `if` statements are trickier, requiring the creation of fresh BBs to handle the true, false and final/exit execution paths. A conditional branch is created that references the evaluated boolean condition and targets the true and false BBs as required. Code for both branches is generated that terminates with an unconditional branch to the exit BB. `if` statements are the only operation that require conditional branches within the otherwise sequential simulation kernel. An example of the generated *solver loop* for the *Ode* model given in Listing C.3 is provided in Listing C.4, it also corresponds to the *solver loop* callgraph in Fig. C.6.

Listing C.4 LLVM assembly fragment generated from the *Ode* code in Listing C.3. This fragment is from the `solverLoop` of the model-simulation code using the forward-Euler solver.

```
do.body:    ; the main solver block                ; preds = %do.cond, %entry
  %derefVal3 = load i64* @simCurLoop               ; sets up solver loop
  %incCurLoop = add i64 %derefVal3, 1
  store i64 %incCurLoop, i64* @simCurLoop
  %derefVal4 = load i64* @simCurLoop
  %convDouble = uitofp i64 %derefVal4 to double
  %timeDelta = fmul double %convDouble, 3.125000e-02
  %curTime = fadd double %timeDelta, 0.000000e+00
  store double %curTime, double* @simCurTime
  %stateVal5 = load double* @odeVal0StateRef       ; calculate y' = f(y, t)
  %derefVal6 = load double* @simCurTime
  %odeVal8 = fdiv double %stateVal5, 2.000000e+01
  %odeVal7 = fsub double 1.000000e+00, %odeVal8
  %odeVal6 = fmul double 1.000000e+00, %odeVal7
  %odeVal5 = fmul double %odeVal6, %stateVal5
  store double %odeVal5, double* @odeVal0DeltaRef
  %derefVal7 = load double* @odeVal0StateRef      ; perform FEuler integration
  %derefVal8 = load double* @odeVal0DeltaRef
  %deltaTime = fmul double %derefVal8, 3.125000e-02
  %newState = fadd double %derefVal7, %deltaTime
  store double %newState, double* @odeVal0StateRef
  %derefVal9 = load i64* @simCurPeriod
  %bWriteOut = icmp eq i64 %derefVal9, 1
  br i1 %bWriteOut, label %if.true, label %if.false

do.cond:    ; test to end sim                      ; preds = %if.end
  %derefVal14 = load double* @simCurTime
  %bWhileTime = fcmp olt double %derefVal14, 8.000000e+00
  br i1 %bWhileTime, label %do.body, label %do.end

do.end:     ; end sim                              ; preds = %do.cond
  call void @OdeStopSim()
  call void @OdeShutdown()
  ret void

if.true:    ; output sim state to file            ; preds = %do.body
  %derefVal11 = load double* @odeVal0StateRef
  store double %derefVal11, double* getelementptr inbounds
    ([2 x double]* @simOutData, i64 0, i64 1)
  %derefVal12 = load double* @simCurTime
  call void @OdeWriteState(double %derefVal12, double* getelementptr inbounds
    ([2 x double]* @simOutData, i64 0, i64 0))
  store i64 1, i64* @simCurPeriod
  br label %if.end

if.false:   ; update solver state                  ; preds = %do.body
  %derefVal13 = load i64* @simCurPeriod
  %incCurPeriod = add i64 %derefVal13, 1
  store i64 %incCurPeriod, i64* @simCurPeriod
  br label %if.end

if.end:                                            ; preds = %if.false, %if.true
  br label %do.cond
```

C.3 Results Tables

We provide several tables containing the full result data obtained from the implementation simulations and benchmarks described in Chapter 5.

Implementation Performance

Table C.2 displays the data obtained from simulating several cardiac models using *Ode* and alternate simulation systems, using the methodology described in Section 5.3.2. They are used within the implementation benchmarks in that section to compare the simulation performance of *Ode* against CellML-derived C models when using basic optimisations that preserve numerical accuracy. Simulation times are presented, alongside the maximum memory required during simulation, the size of the compiled binary, and the FP difference of a simulation against the native *Ode* equivalent.

Optimisation Benchmarks

Table C.3 presents data obtained from simulations utilising the optimisation strategies described in Section 5.4 within the optimisation simulation benchmarks in Section 5.4.5. The results are used to compare the same models in *Ode* and C when enabling more aggressive *Ode*-specific and general LLVM optimisations that may result in slight numerical inaccuracies. The 'Optimised' results in Table C.3 are the same as those in Table C.2, representing *safe* optimisations we can use as a reference. Simulation times for *Ode* and C equivalents are presented, and we compare the FP difference of a simulation against both unoptimised *Ode* and equivalently optimised C simulation results.

VecMath Algorithm Results

A simple example of vectorisation/ILP is provided in Listing C.5, this expands upon the pseudo-code variant in Listing 5.1, presenting the generated LLVM bitcode.

Table C.4 presents full data obtained from applying the *vecmath* optimisation to several cardiac models in *Ode* that were simulated using multiple maths libraries on a CPU with 2xdouble width vectors. The *VecOps* column indicates the initial and final number of maths operations having performed the *vecmath* optimisation. The GNU math library represents the baseline scalar case,

Listing C.5 Application of the *vecmath* to the same *Ode* model presented in Listing 5.1, depicting the generated LLVM assembly both pre- and post-optimisation.

```
// initial code, assumes initial value "x"
val a = exp(x+1)  // dependent on "x", vectorisation candidate
val b = exp(a)    // dependent on "a", cannot be vectorised
val c = exp(x+2)  // dependent on "x", vectorisation candidate pair with "a"
val d = a + b + c // dependent on "a", "b", and "c"

; pre-vecmath generated LLVM bitcode
%odeVal6 = fadd double %stateVal, 1.000000e+00
%odeVal1 = tail call double @exp(double %odeVal6) nounwind readnone
%odeVal2 = tail call double @exp(double %odeVal1) nounwind readnone
%odeVal7 = fadd double %stateVal, 2.000000e+00
%odeVal3 = tail call double @exp(double %odeVal7) nounwind readnone
%odeVal8 = fadd double %odeVal1, %odeVal2
%odeVal4 = fadd double %odeVal8, %odeVal3

; post-vecmath generated LLVM bitcode
%odeVal6 = fadd double %stateVal, 1.000000e+00
%odeVal7 = fadd double %stateVal, 2.000000e+00
%odeVal1.v.i0.1 = insertelement <2 x double> undef, double %odeVal6, i32 0
%odeVal1.v.i0.2 = insertelement <2 x double> %odeVal1.v.i0.1, double %odeVal7, i32 1
%odeVal1 = tail call <2 x double> @vecmath_exp_v2f64(<2 x double> %odeVal1.v.i0.2)
              nounwind readnone
%odeVal1.v.r1 = extractelement <2 x double> %odeVal1, i32 0
%odeVal1.v.r2 = extractelement <2 x double> %odeVal1, i32 1
%odeVal2 = tail call double @vecmath_exp_f64(double %odeVal1.v.r1)
              nounwind readnone
%odeVal8 = fadd double %odeVal1.v.r1, %odeVal2
%odeVal4 = fadd double %odeVal1.v.r2, %odeVal8
```

while the Intel and AMD libraries are highly-optimised and implement both scalar (x1) and vectorised (x2) math functions. As seen, the scalar GNU library demonstrates poor performance when compared to the Intel and AMD scalar equivalents. When using the AMD library, the vectorisation improves performance only slightly by ~5-10%.

Table C.5 presents the data from a second set of simulations performed on a different, modern, Intel CPU that supports both 2x and 4xdouble width vectors. The vecmath optimisation was applied with vector widths of 2 and 4 and simulated using several math libraries as per Table C.4.

Implementation	Time (s)	Runtime Memory (MB)	Binary Size (B)	FP Diff (eps)
HH52 Model				
Ode (Native) *	59.5	0.71	4168	*NA*
C	61.5	0.73	7541	2944
Ode (Interpreter)	9 561.0	242.78	*NA*	0
Python	10 463.0	14.45	*NA*	2944
MATLAB	10 340.0	85.00	*NA*	24224
N62 Model				
Ode (Native) *	60.8	0.71	4173	*NA*
C	61.1	0.73	7418	0
Ode (Interpreter)	8 450.0	226.31	*NA*	0
Python	5 612.0	14.53	*NA*	0
MATLAB	5 638.0	84.90	*NA*	342272
BR77 Model				
Ode (Native) *	57.7	0.74	6763	*NA*
C	58.0	0.75	9080	23936
Ode (Interpreter)	9 054.0	202.79	*NA*	0
Python	6 932.0	14.77	*NA*	23936
MATLAB	4 192.0	86.37	*NA*	23936
LRd94 Model				
Ode (Native) *	41.8	0.75	9226	*NA*
C	59.6	0.77	13933	43779
Ode (Interpreter)	22 357.0	77.06	*NA*	0
Python	9 629.0	15.90	*NA*	43779
MATLAB	6 240.0	86.89	*NA*	43894
TNNP04 Model				
Ode (Native) *	59.5	0.76	11299	*NA*
C	69.4	0.77	14006	2944
Ode (Interpreter)	21 433.0	77.04	*NA*	0
Python	13 051.0	15.60	*NA*	2944
MATLAB	8 365.0	87.84	*NA*	1920
ORd11 Model				
Ode (Native) *	47.9	0.78	20146	*NA*
C	55.8	0.78	23973	896
Ode (Interpreter)	41 670.0	389.00	*NA*	0
Python	8 303.0	17.30	*NA*	896
MATLAB	5 135.0	92.03	*NA*	528

Table C.2: Benchmark results from simulating several cardiac models using *Ode* (both compiled and interpreted) alongside the commonly used C, MATLAB and Python languages.

Optimisation	***Ode* Time (s)**	**C Time (s)**	***Ode*-Base FP Diff (eps)**	**C-*Ode* FP Diff (eps)**
HH52 model				
Unoptimised *	61.9	64.8	*NA*	2944
Optimised	59.5	61.5	0	2944
FastMath	56.0	57.7	24224	256
Constant Computations	37.8	*NA*	24224	*NA*
N62 model				
Unoptimised *	61.8	63.6	*NA*	0
Optimised	60.8	61.1	0	0
FastMath	56.1	57.2	386624	124576
Constant Computations	40.7	*NA*	1122592	*NA*
BR77 model				
Unoptimised *	58.5	59.4	*NA*	23936
Optimised	57.7	58.0	0	23936
FastMath	54.1	54.8	23936	960
Constant Computations	51.0	*NA*	23936	*NA*
LRd94 model				
Unoptimised *	63.9	63.1	*NA*	43779
Optimised	41.8	59.6	0	43779
FastMath	39.3	49.1	28757	8833
Constant Computations	34.3	*NA*	12027	*NA*
TNNP04 model				
Unoptimised *	75.1	77.2	*NA*	2944
Optimised	59.5	69.4	0	2944
FastMath	56.9	61.3	2944	2176
Constant Computations	50.0	*NA*	1792	*NA*
ORd11 model				
Unoptimised *	58.5	61.1	*NA*	896
Optimised	47.9	55.8	0	896
FastMath	43.5	50.5	512	384
Constant Computations	37.3	*NA*	512	*NA*

Table C.3: Benchmark results from simulating several cardiac models at various optimisation levels using *Ode* and C. For each model the results are compared to a reference to determine the numerical error introduced by each optimisation.

Model	VecOps	GNU x1 Time (s)	Intel x1 Time (s)	Intel x2 Time (s)	AMD x1 Time (s)	AMD x2 Time (s)
HH52	6 → 3	37.83	9.26	5.43	10.19	8.54
N62	8 → 4	40.67	8.90	5.98	8.93	8.05
BR77	26 → 14	51.03	12.07	7.11	11.23	10.73
LRd94	38 → 21	34.32	9.50	6.53	9.92	9.91
TNNP04	54 → 29	50.05	12.77	8.72	13.17	12.38
ORd11	68 → 35	37.47	11.83	8.23	11.68	10.90

Table C.4: Benchmark results from simulating several cardiac models using the *vecmath* optimisation within *Ode*. The models were simulated using several maths libraries on an Intel CPU with a 2xdouble vector-width. The VecOps column indicates the initial and final number of transcendental functions in the model.

Model	VecOps	GNU x1 Time (s)	Intel x1 Time (s)	Intel x2 Time (s)	Intel x4 Time (s)	AMD x1 Time (s)	AMD x2 Time (s)
HH52	6 → 2	6.63	4.62	3.02	3.15	4.72	4.19
N62	8 → 2	6.97	4.64	2.62	2.05	4.77	3.80
BR77	26 → 8	9.19	5.54	3.62	3.15	6.18	5.00
LRd94	38 → 14	7.72	4.94	3.41	3.09	5.22	4.87
TNNP04	54 → 18	9.59	6.44	3.94	3.47	6.60	5.40
ORd11	68 → 20	8.29	5.92	3.86	3.96	6.11	4.97

Table C.5: Benchmark results from simulating several cardiac models using the *vecmath* optimisation within *Ode*. The models were simulated using several maths libraries on a differing, modern Intel CPU to that in Table C.4 with a 2xdouble and 4xdouble vector-width.

C.4 FFI

A foreign-function interface (FFI) enables code written in one language to call code written in another through a well-defined interface with agreed conventions. In many cases is based on the C-language interface and conventions as provided by the host operating system. We have implemented a C-based FFI for *Ode* that allows systems created in other languages to call into the *Ode* generated code to evaluate the computationally intensive simulation-kernel.

This will allow using *Ode* to generate models and compile a highly-optimised computational kernel that may be utilised with external software. This may include more complex and advanced solvers, e.g. adaptive mechanisms, or multi-scale model simulation, e.g. facilitating integration into Chaste tissue and organ simulations [120]. We utilise the FFI when simulating models using the integrated adaptive solver within *Ode*. In this mode the system generates an object-file that exports the FFI API and links it to a pre-compiled external solver system written in C that utilises CVODE. This enables efficient adaptive simulation within the DSL whilst serving to test the FFI implementation.

The FFI is used in conjunction with the object-file backend, resulting in a native-code library that may be utilised by any system capable of communicating with a C-based API. The generated code is allocation-free and thread-safe, a task made simpler due to the design of the simulation-kernel. This enables easy integration with existing, potentially multi-threaded, code. Currently the FFI exports only the model simulation-kernel and is only useful for ODE and SDE-based models.

To use the calling implementation must allocate 3 double arrays of size equal to the model; these hold the current state, delta/drift and diffusion values for each model state element, i.e. (y, y', w). A C header is provided that defines the functions and simulation parameters available in the API, many of these map to values configured by the console (see Appendix A.1) and are described in the following table. There are two main functions, `OdeModelInitials` to set the initial values, $y(t_0)$, and `OdeModelRHS` which implements the calculation of the derivatives in standard-form, $y' = f(t, y)$.

Declaration	Description
`const double OdeParamStartTime`	Initial simulation time, t_{start}
`const double OdeParamStopTime`	Final simulation time, t_{stop}
`const double OdeParamTimestep`	Simulation timestep, h, used as the minimum timestep with an adaptive solver
`const double OdeParamMaxTimestep`	Max. timestep, t_{max}, during adaptive simulation
`const uint64_t OdeParamMaxNumSteps`	Max. number of adaptive steps to take from current to next time $t_n \rightarrow t_{n+1}$
`const double OdeParamRelativeError`	Relative error during adaptive simulation
`const double OdeParamAbsoluteError`	Absolute error during adaptive simulation
`const enum OdeParamModelTypes OdeParamModelType`	Model type when performing adaptive simulation <Stiff, NonStiff>
`const double OdeParamPeriod`	Interval period, t_i, to save model simulation state
`const uint64_t OdeParamPeriodInterval`	Number of timesteps in t_i
`const double OdeParamStartOutputTime`	Time, t_{out}, from when simulation state is saved
`const char OdeParamOutput[]`	Filename for saving simulation results
`const enum OdeParamSimTypes OdeParamSimType`	Simulation type configured by the model <ODE, SDE, SSA, Hybrid>
`const uint64_t OdeParamStateSize`	Number of system state variables, i.e. $\mid y \mid$
`void OdeModelInitials(const double time, double* state)`	Calculates, and returns, the initial model values, i.e. $y(t_0)$
`void OdeModelRHS(const double time, const double* state, double* delta, double* wiener)`	Calculates the system deltas given the current time and state, i.e. $y' = f(t, y)$

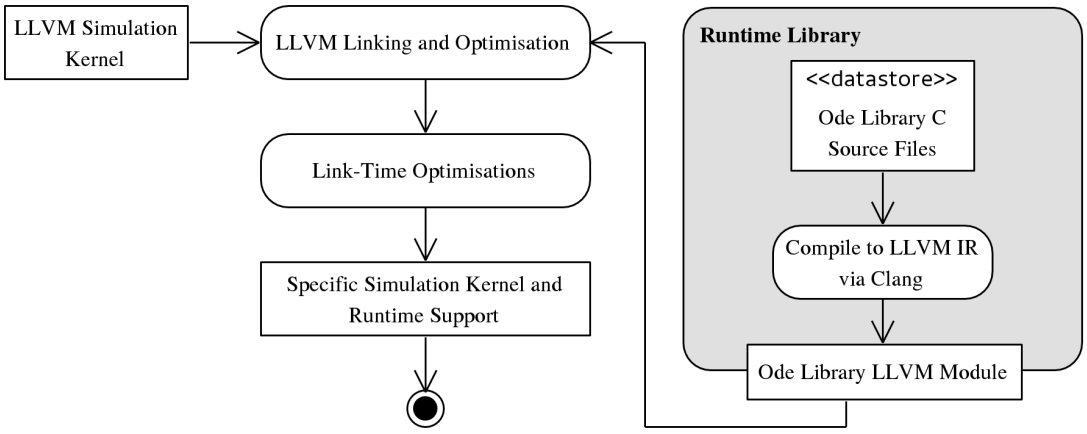


Figure C.7: Linking the *Ode* run-time library and simulation code. The C-based *Ode* run-time library is combined with the custom LLVM model-simulation code during compilation into a single, specific, native code block.

C.5 *Ode* Runtime Library

Compiled *Ode* simulations require several auxiliary functions to run successfully. These are provided by the *Ode* runtime library, this is similar in concept to the C runtime library `libc` provided by most operating systems.

The *Ode* runtime library is implemented in C and compiled to optimised LLVM bitcode using the Clang C compiler[3]. This is because the functionality is static and reusable rather than model-specific, and thus can be implemented within a higher-level language than as low-level bitcode without affecting performance. As depicted in Fig. C.7, the runtime library is linked during compilation to the code-generated *Ode* model-simulation module using LLVM tools. We perform link-time optimisation (LTO), enabling late linking and optimisation of the standard library functions with respect to the specific model-simulation code. In many cases this fully inlines the runtime library and several optimisations that, for instance, replace array copies with direct references.

The library includes functions to setup and shutdown simulations, file-writing routines (using the format described in Appendix A.6), fast random number generation and numerical projection functions. They are listed in the following table using C-style declarations.

[3] `http://clang.llvm.org`

Declaration	Description
`void OdeInit(void)`	Initialise the *Ode* solver environment
`void OdeShutdown(void)`	Shutdown the *Ode* solver environment
`void OdeStartSim(const char* filename, const uint64_t inStateSize)`	Called when starting a simulation run, sets the output filename and system size
`void OdeStopSim(void)`	Called after completing a simulation run
`void OdeWriteState(const double time, const double* state)`	Save simulation state, as specified by the `time` and `state` inputs
`double OdeRandUniform(void)`	Return a uniformly-distributed random number
`double OdeRandNormal(void)`	Return a normally-distributed random number
`void OdeProjectVector( double* xs, const uint64_t xsSize)`	Used by Projecting-SDE solver to project an array of doubles in-place onto a simplex, where $0 \leq x_n \leq 1$ and $\Sigma x_n = 1$, adapted from [22, 37]

Appendix D

Models & Tutorial

This appendix provides listings of several cardiac models used during the design, development, and testing of the *Ode* and *Ion* DSLs. It includes models that showcase the DSL modelling features and functionality, from deterministic and stochastic perspectives. We also provide a tutorial demonstrating the use of *Ode* to model and simulate a simple ion channel model, and an introduction to the use of UML to graphically depict such models. Finally we provide (abridged) CellML versions of the HH52 and BR77 AP models to compare and contrast against their *Ode* counterparts. We note that the code listings in this appendix are provided in a smaller font to save space.

D.1 *Ode* Tutorial

This tutorial depicts the use of *Ode* to model a simple Markov-formulation ion channel system, using ODEs, that is then simulated over a fixed voltage, as was demonstrated in Section 2.2.

D.1.1 Model

Our model is based on the simple 3-state reaction system described in Section 2.2,

$$X1 \underset{b_1}{\overset{a_1}{\rightleftharpoons}} X2 \underset{b_2}{\overset{a_2}{\rightleftharpoons}} X3.$$

We define voltage-dependent reaction rates based upon the HH52 K channel described in Fig. 2.5. Essentially we are modelling a subset of the HH52 K channel with 2 voltage-dependent gates rather than 4 at a fixed membrane voltage, V, of $15mV$. The model parameters are as follows,

$$V = 15$$

$$\alpha_n = \frac{-0.01(V+65)}{e^{\frac{-(V+65)}{10}} - 1} \qquad \beta_n = 0.125e^{\frac{V+75}{80}}$$

$$a_1 = 2\alpha_n \qquad b_1 = \beta_n \qquad a_2 = \alpha_n \qquad b_2 = 2\beta_n.$$

Using the approach presented in Section 2.2.5, we derive the following deterministic approximation of the system,

$$\Delta X_1 = -a_1 X_1 + b_1 X_2$$

$$\Delta X_2 = a_1 X_1 + -(b_1 + a_2) X_2 + b_2 X_3$$

$$\Delta X_3 = a_2 X_2 + -b_2 X_3.$$

This system can be modelled and simulated in *Ode* (the *Ion* DSL can automatically perform this transformation from the system reactions, as detailed in Section 6.2).

D.1.2 *Ode* Model Representation

To capture the above model in *Ode* we first create a file, named `Examples.od3`, and save it within the `$HOME/DefaultRepo` directory, representing a model repository. Within this file

we add the following line to define the `IonChannelModel` module that will contain the model,

```
module IonChannelModel {
```

We add the following code to the module, representing constant computations evaluated on each timestep and required in the ODE definitions. These map to the model parameters previously defined above,

```
  val V = 0.0
  val alpha_n = -0.01 * (V + 65) / (exp(-(V + 65) / 10) - 1)
  val beta_n = 0.125 * exp((V + 75) / 80)
  val a1 = 2*alpha_n
  val b1 = beta_n
  val a2 = alpha_n
  val b2 = 2*beta_n
```

Next we add the initial values that capture the model's state over time. The system consists of three state values, representing the proportion of ion channels in each state. They are created with the following initial values that determine their value at t_{start},

```
  init X1 = 1.0
  init X2 = 0.0
  init X3 = 0.0
```

Finally we define the ODEs themselves, based on the syntax in Section 3.1.6, and close the module. The ODE definitions are based on those provided earlier and make use of previously calculated values in the module.

```
  ode {initVal : X1} = -a1*X1 + b1*X2
  ode {initVal : X2} = a1*X1 + -(b1+a2)*X2 + b2*X3
  ode {initVal : X3} = a2*X2 + -b2*X3
}
```

This code models the proportion of ion channels in each state from a continuous, deterministic viewpoint. If we wanted to include such a model within a larger cardiac electrophysiological model, we could add the following code, as in Eq. (2.2), to calculate the ionic current generated by the open channels,

```
  val G_I = 36.0
  val E_I = -87.0
  val current = G_I*(X3)*(V-E_I)
```

D.1.3 *Ode* Simulation

We can simulate this example model by starting the *Ode* implementation from the command line,

```
~> ode
Welcome to Ode, please enter commands.
» _
```

From this console interface we can load modules, and configure and perform simulations (see Section 3.4.1). To load the module, we first add the repository and then explicitly import the module into the environment as follows,

```
:addRepo ~/DefaultRepo
import Example.IonChannelModel
```

Next we configure the simulation parameters (see Appendix A.1). The following commands set the simulation duration, specify the ODE solver, and set the output parameters,

```
:startTime 0
:stopTime 10
:timestep 0.01
:period 0.1
:odeSolver "euler"
:output "results.bin"
```

Finally we enter the `simulate` command while specifying the already-loaded `IonChannelModel` to start simulation, and exit the *Ode* console,

```
:simulate IonChannelModel
:quit
```

D.1.4 Results

The `simulate` command will perform a single simulation run of the model specified by `Example.IonChannelModel` according to the desired simulation configuration. The state of the model during simulation is saved to the `results.bin` file at 1*ms* intervals, using the file format specified in Appendix A.6.

We can examine these results using several tools included with *Ode* that utilise the NumPy[1] numerical calculation and plotting library. These include tools to load and save files, print and

[1] `http://numpy.org`

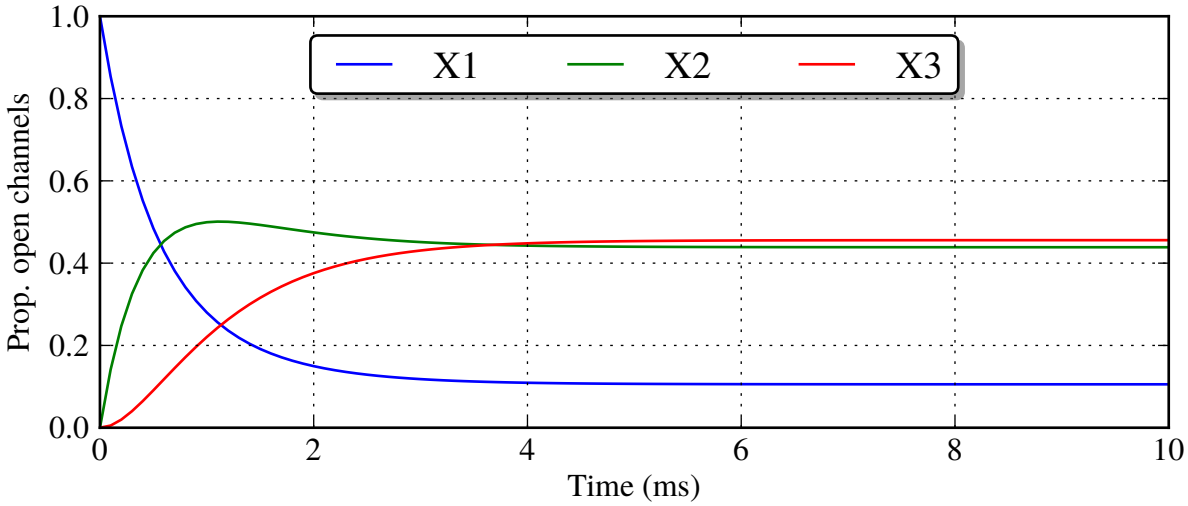


Figure D.1: Plot of the `IonChannelModel` model simulation results using the parameters specified in the tutorial. The results are as expected, with a large proportion of channels in the 'open' X_3 state at V=15.

dump data, plot the results and perform numerical comparisons between multiple simulation runs.

Fig. D.1 plots the data encoded in the `results.bin` results file, as expected a large proportion of channels open at V=15.

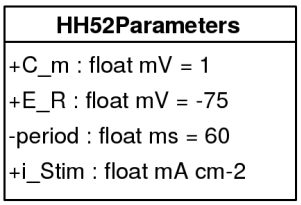
D.2 *Ode* UML Representation

This section presents an introduction to Unified Modelling Language (UML) diagrams. We use UML figures throughout the thesis to help visualise the structure of *Ode* models and their composition and dependence.

UML class diagrams are often used to model the structural relationships between objects in object-orientated languages, such as Java and C++. However UML class diagrams rely on concepts and patterns common to most structured programming languages and are familiar to many programmers, acting as a 'visual pseudocode'. We use a subset of UML class diagram nomenclature to describe the structure of mathematical biological models using the *Ode* module system presented in Chapter 4. This appendix describes these UML concepts for readers unfamiliar with the notation.

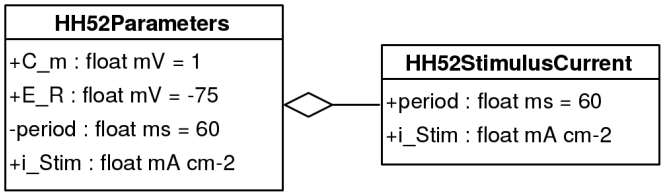
The modularisation of the HH52 model cell-level parameters (see Section 2.1.3) is used as a motivating example, making use of the modelling techniques described in Section 4.1. Further information on UML can be found in [12, 16].

Encapsulation



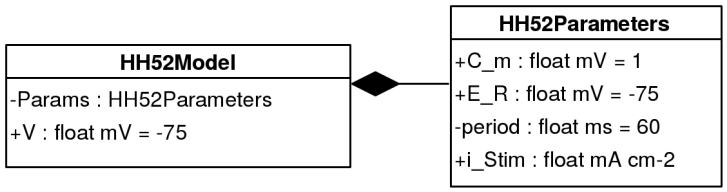
Encapsulation is the concept of grouping (logically) related data objects into a single entity from which only the pertinent information is exported. We model this using a basic UML class diagram as above, encoding several cell-level parameters from the HH52 model within a single module. Each entry has an identifying name, type, and default value. The plus and minus signs indicate whether the value is accessible externally.

Containment — Aggregation



Containment is the means by which encapsulated objects may reference and include one another. Aggregation is represented by a line between two modules, as above, where the position of the white-diamond indicates that the object contains, or aggregates, the other object. In the above diagram we model the stimulus current for the `HH52Parameters` module using the aggregated `HH52StimulusCurrent` module that may be used elsewhere. Aggregation is stronger than association, implying that a module can be used as a part of another but that it also independent and may be used elsewhere.

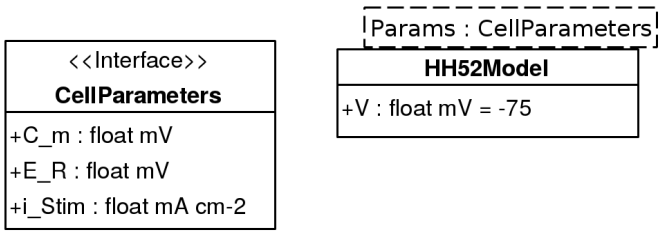
Containment — Composition



Composition also represents association and containment, however implies a stronger relationship than aggregation. It indicates that the composed object is a part of its contained object, i.e. is dependent upon it and cannot be used elsewhere.

As with aggregation, composition is represented by a line between the two connected modules, where the position of the black-diamond indicates that the object is composed from the other object. In the above diagram we model the `HH52Model` as being created from the static composition of its parameters, termed `Params`, along with its own data.

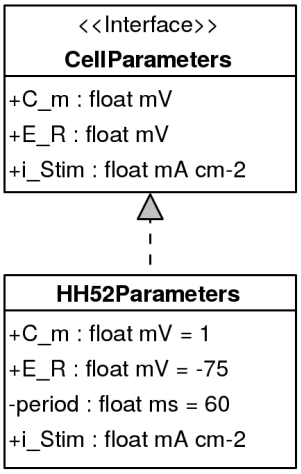
Interfaces — Definition



An interface represents a common protocol that an object, or module, must correspond to for correct usage. It does not contain code, but only specifies the 'interface' that compatible code should possess. In *Ode* an interface is defined by the type-signature of a module, and is implicitly created when using parameterised modules.

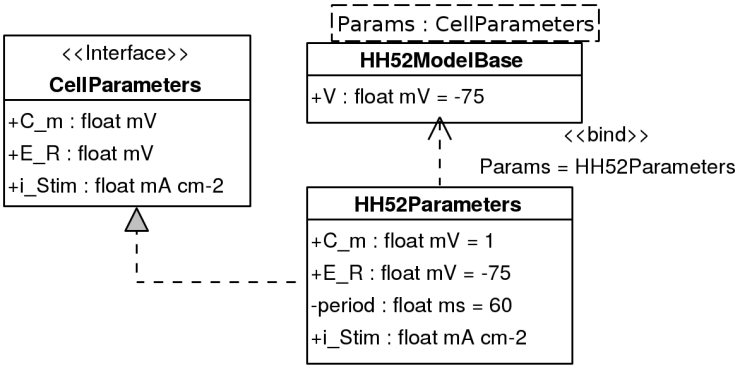
The above diagram indicates how we specify this structure in UML. The `HH52Model` is parameterised by a module termed `Params`, seen in the dotted box above the module description. This `Params` module represents the HH52 cell parameters and utilises the implicitly defined interface termed `CellParameters`, whose protocol/signature is shown on the left.

Interfaces — Implementation



An interface may be implemented by several modules that may be exchanged so long as they correspond to the same common protocol defined by the interface. The above UML diagram indicates how we may represent this relationship — a dotted, filled arrow is drawn from a code-containing concrete implementation to the abstract interface it implements. Here the module `HH52Parameters` implements the interface `CellParameters` as is required.

Interfaces — Instantiation



Finally, a concrete module compatible with a specified interface may be used and instantiated to represent the parameter within a parameterised module. The above digram indicates how we may express this *Ode* concept using UML. Following from the previous subsection, the `HH52Model`

module is parameterised on a module termed `Params` that must implement the `CellParameters` interface. The dotted, filled arrow from `HH52Parameters` to the `CellParameters` interface indicates that this module implements the required interface.

Finally, the dotted, open arrow from `HH52Parameters` to `HH52Model` depicts the instantiating and binding of this specific concrete module to represent `Params` within the `HH52Model` module, as indicated further by the *bind* text next to the arrow.

D.3 CellML

This section lists several electrophysiological models created using the CellML modelling DSL. Both the models and the DSL were referenced during the design and development of the *Ode* modelling DSL. These models were used within the implementation benchmarks shown in Chapter 5 to generate C, MATLAB, and Python representations for simulation. However we have modified their presentation below and removed all mathematical expressions for brevity due to the use of MathML.

When comparing these models to their *Ode* counterparts in Appendix D.4 we see that the structural abstraction facilities provided by *Ode* allow for significantly shorter and succinct models. In contrast the structure and connecting of components must be provided explicitly in CellML, greatly complicating and lengthening the model definitions, as seen below.

D.3.1 HH52 CellML Model

The following listing provides the canonical encoding of the Hodgkin-Huxley AP model [69] (see Section 2.1.3) in CellML, taken from the CellML repository.

```
<?xml version="1.0"?>
<model xmlns="http://www.cellml.org/cellml/1.0#" xmlns:cmeta="http://www.cellml.org/metadata/1.0#"
  xml:base="" cmeta:id="hodgkin_huxley_squid_axon_1952" name="hodgkin_huxley_squid_axon_1952">
  <units name="millisecond">
    <unit prefix="milli" units="second"/>
  </units>
  <units name="per_millisecond">
    <unit exponent="-1" prefix="milli" units="second"/>
  </units>
  <units name="millivolt">
    <unit prefix="milli" units="volt"/>
  </units>
  <units name="per_millivolt_millisecond">
    <unit exponent="-1" units="millivolt"/>
    <unit exponent="-1" prefix="milli" units="second"/>
  </units>
  <units name="milliS_per_cm2">
    <unit prefix="milli" units="siemens"/>
    <unit exponent="-2" prefix="centi" units="metre"/>
  </units>
  <units name="microF_per_cm2">
    <unit prefix="micro" units="farad"/>
    <unit exponent="-2" prefix="centi" units="metre"/>
  </units>
  <units name="microA_per_cm2">
    <unit prefix="micro" units="ampere"/>
    <unit exponent="-2" prefix="centi" units="metre"/>
  </units>
  <component name="environment">
    <variable cmeta:id="environment_time" name="time" public_interface="out" units="millisecond"/>
  </component>
  <component name="membrane">
    <variable cmeta:id="membrane_V" initial_value="-75" name="V" public_interface="out"
      units="millivolt"/>
    <variable initial_value="-75" name="E_R" public_interface="out" units="millivolt"/>
```

```
    <variable initial_value="1" name="Cm" units="microF_per_cm2"/>
    <variable name="time" public_interface="in" units="millisecond"/>
    <variable name="i_Na" public_interface="in" units="microA_per_cm2"/>
    <variable name="i_K" public_interface="in" units="microA_per_cm2"/>
    <variable name="i_L" public_interface="in" units="microA_per_cm2"/>
    <variable name="i_Stim" units="microA_per_cm2"/>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <!- MathML goes here - removed for brevity ->
    </math>
  </component>
  <component name="sodium_channel">
    <variable cmeta:id="sodium_channel_i_Na" name="i_Na" public_interface="out"
      units="microA_per_cm2"/>
    <variable initial_value="120" name="g_Na" units="milliS_per_cm2"/>
    <variable name="E_Na" units="millivolt"/>
    <variable name="time" private_interface="out" public_interface="in" units="millisecond"/>
    <variable name="V" private_interface="out" public_interface="in" units="millivolt"/>
    <variable name="E_R" public_interface="in" units="millivolt"/>
    <variable name="m" private_interface="in" units="dimensionless"/>
    <variable name="h" private_interface="in" units="dimensionless"/>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <!- MathML goes here - removed for brevity ->
    </math>
  </component>
  <component name="sodium_channel_m_gate">
    <variable initial_value="0.05" name="m" public_interface="out" units="dimensionless"/>
    <variable name="alpha_m" units="per_millisecond"/>
    <variable name="beta_m" units="per_millisecond"/>
    <variable name="V" public_interface="in" units="millivolt"/>
    <variable name="time" public_interface="in" units="millisecond"/>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <!- MathML goes here - removed for brevity ->
    </math>
  </component>
  <component name="sodium_channel_h_gate">
    <variable initial_value="0.6" name="h" public_interface="out" units="dimensionless"/>
    <variable name="alpha_h" units="per_millisecond"/>
    <variable name="beta_h" units="per_millisecond"/>
    <variable name="V" public_interface="in" units="millivolt"/>
    <variable name="time" public_interface="in" units="millisecond"/>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <!- MathML goes here - removed for brevity ->
    </math>
  </component>
  <component name="potassium_channel">
    <variable cmeta:id="potassium_channel_i_K" name="i_K" public_interface="out"
      units="microA_per_cm2"/>
    <variable initial_value="36" name="g_K" units="milliS_per_cm2"/>
    <variable name="E_K" units="millivolt"/>
    <variable name="time" private_interface="out" public_interface="in" units="millisecond"/>
    <variable name="V" private_interface="out" public_interface="in" units="millivolt"/>
    <variable name="E_R" public_interface="in" units="millivolt"/>
    <variable name="n" private_interface="in" units="dimensionless"/>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <!- MathML goes here - removed for brevity ->
    </math>
  </component>
  <component name="potassium_channel_n_gate">
    <variable initial_value="0.325" name="n" public_interface="out" units="dimensionless"/>
    <variable name="alpha_n" units="per_millisecond"/>
    <variable name="beta_n" units="per_millisecond"/>
    <variable name="V" public_interface="in" units="millivolt"/>
    <variable name="time" public_interface="in" units="millisecond"/>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <!- MathML goes here - removed for brevity ->
    </math>
  </component>
  <component name="leakage_current">
    <variable cmeta:id="leakage_current_i_L" name="i_L" public_interface="out"
      units="microA_per_cm2"/>
    <variable initial_value="0.3" name="g_L" units="milliS_per_cm2"/>
    <variable name="E_L" units="millivolt"/>
    <variable name="time" public_interface="in" units="millisecond"/>
    <variable name="V" public_interface="in" units="millivolt"/>
    <variable name="E_R" public_interface="in" units="millivolt"/>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
```

```
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<group>
  <relationship_ref relationship="containment"/>
  <component_ref component="membrane">
    <component_ref component="sodium_channel">
      <component_ref component="sodium_channel_m_gate"/>
      <component_ref component="sodium_channel_h_gate"/>
    </component_ref>
    <component_ref component="potassium_channel">
      <component_ref component="potassium_channel_n_gate"/>
    </component_ref>
    <component_ref component="leakage_current"/>
  </component_ref>
</group>
<group>
  <relationship_ref relationship="encapsulation"/>
  <component_ref component="sodium_channel">
    <component_ref component="sodium_channel_m_gate"/>
    <component_ref component="sodium_channel_h_gate"/>
  </component_ref>
  <component_ref component="potassium_channel">
    <component_ref component="potassium_channel_n_gate"/>
  </component_ref>
</group>
<connection>
  <map_components component_1="membrane" component_2="environment"/>
  <map_variables variable_1="time" variable_2="time"/>
</connection>
<connection>
  <map_components component_1="sodium_channel" component_2="environment"/>
  <map_variables variable_1="time" variable_2="time"/>
</connection>
<connection>
  <map_components component_1="potassium_channel" component_2="environment"/>
  <map_variables variable_1="time" variable_2="time"/>
</connection>
<connection>
  <map_components component_1="leakage_current" component_2="environment"/>
  <map_variables variable_1="time" variable_2="time"/>
</connection>
<connection>
  <map_components component_1="membrane" component_2="sodium_channel"/>
  <map_variables variable_1="V" variable_2="V"/>
  <map_variables variable_1="E_R" variable_2="E_R"/>
  <map_variables variable_1="i_Na" variable_2="i_Na"/>
</connection>
<connection>
  <map_components component_1="membrane" component_2="potassium_channel"/>
  <map_variables variable_1="V" variable_2="V"/>
  <map_variables variable_1="E_R" variable_2="E_R"/>
  <map_variables variable_1="i_K" variable_2="i_K"/>
</connection>
<connection>
  <map_components component_1="membrane" component_2="leakage_current"/>
  <map_variables variable_1="V" variable_2="V"/>
  <map_variables variable_1="E_R" variable_2="E_R"/>
  <map_variables variable_1="i_L" variable_2="i_L"/>
</connection>
<connection>
  <map_components component_1="sodium_channel" component_2="sodium_channel_m_gate"/>
  <map_variables variable_1="m" variable_2="m"/>
  <map_variables variable_1="time" variable_2="time"/>
  <map_variables variable_1="V" variable_2="V"/>
</connection>
<connection>
  <map_components component_1="sodium_channel" component_2="sodium_channel_h_gate"/>
  <map_variables variable_1="h" variable_2="h"/>
  <map_variables variable_1="time" variable_2="time"/>
  <map_variables variable_1="V" variable_2="V"/>
</connection>
<connection>
  <map_components component_1="potassium_channel" component_2="potassium_channel_n_gate"/>
  <map_variables variable_1="n" variable_2="n"/>
  <map_variables variable_1="time" variable_2="time"/>
```

```
    <map_variables variable_1="V" variable_2="V"/>
  </connection>
</model>
```

D.3.2 BR77 CellML Model

The following listing provides the canonical encoding of the Beeler-Reuter cardiac ventricular AP model [69] (see Section 2.1.3) in CellML, taken from the CellML repository. The abbreviated listing below is ~300 lines, however the unedited model is over 1200 lines; this can be compared to the much shorter *Ode* version provided in Appendix D.4.4.

```
<?xml version="1.0" encoding="utf-8"?>
<model xmlns="http://www.cellml.org/cellml/1.0#" xmlns:cmeta="http://www.cellml.org/metadata/1.0#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:bqs="http://www.cellml.org/bqs/1.0#"
  xmlns:cellml="http://www.cellml.org/cellml/1.0#" xmlns:dcterms="http://purl.org/dc/terms/"
  xmlns:vCard="http://www.w3.org/2001/vcard-rdf/3.0#"
  xmlns:ns7="http://www.cellml.org/metadata/simulation/1.0#" cmeta:id="beeler_reuter_1977"
  name="beeler_reuter_1977_version06">
  <units name="ms">
    <unit units="second" prefix="milli"/>
  </units>
  <units name="per_ms">
    <unit units="second" prefix="milli" exponent="-1"/>
  </units>
  <units name="mV">
    <unit units="volt" prefix="milli"/>
  </units>
  <units name="per_mV">
    <unit units="volt" prefix="milli" exponent="-1"/>
  </units>
  <units name="per_mV_ms">
    <unit units="mV" exponent="-1"/>
    <unit units="ms" exponent="-1"/>
  </units>
  <units name="mS_per_mm2">
    <unit units="siemens" prefix="milli"/>
    <unit units="metre" prefix="milli" exponent="-2"/>
  </units>
  <units name="uF_per_mm2">
    <unit units="farad" prefix="micro"/>
    <unit units="metre" prefix="milli" exponent="-2"/>
  </units>
  <units name="uA_per_mm2">
    <unit units="ampere" prefix="micro"/>
    <unit units="metre" prefix="milli" exponent="-2"/>
  </units>
  <units name="concentration_units">
    <unit units="mole" prefix="nano"/>
    <unit units="metre" prefix="milli" exponent="-3"/>
  </units>
  <units name="per_concentration_units">
    <unit units="concentration_units" exponent="-1"/>
  </units>
  <units name="coulomb_per_mole">
    <unit units="coulomb"/>
    <unit units="mole" exponent="-1"/>
  </units>
  <units name="per_mm">
    <unit units="metre" prefix="-3" exponent="-1"/>
  </units>
  <component name="environment">
    <variable units="ms" public_interface="out" name="time" cmeta:id="environment_time"/>
  </component>
  <component name="membrane">
    <variable units="mV" public_interface="out" cmeta:id="membrane_V" name="V"
```

```
    initial_value="-84.624"/>
  <variable units="uF_per_mm2" cmeta:id="membrane_C" name="C" initial_value="0.01"/>
  <variable units="ms" public_interface="in" name="time"/>
  <variable units="uA_per_mm2" public_interface="in" name="i_Na"/>
  <variable units="uA_per_mm2" public_interface="in" name="i_s"/>
  <variable units="uA_per_mm2" public_interface="in" name="i_x1"/>
  <variable units="uA_per_mm2" public_interface="in" name="i_K1"/>
  <variable units="uA_per_mm2" public_interface="in" name="Istim"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML" cmeta:id="membrane_voltage_diff_eq">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<component name="sodium_current">
  <variable units="uA_per_mm2" public_interface="out" name="i_Na" cmeta:id="sodium_current_i_Na"/>
  <variable units="mS_per_mm2" name="g_Na" initial_value="4e-2"/>
  <variable units="mV" name="E_Na" initial_value="50"/>
  <variable units="mS_per_mm2" name="g_Nac" initial_value="3e-5"/>
  <variable units="ms" public_interface="in" private_interface="out" name="time"/>
  <variable units="mV" public_interface="in" private_interface="out" name="V"/>
  <variable units="dimensionless" private_interface="in" name="m"/>
  <variable units="dimensionless" private_interface="in" name="h"/>
  <variable units="dimensionless" private_interface="in" name="j"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<component name="sodium_current_m_gate">
  <variable units="dimensionless" public_interface="out" name="m" initial_value="0.011"/>
  <variable units="per_ms" name="alpha_m"/>
  <variable units="per_ms" name="beta_m"/>
  <variable units="mV" public_interface="in" name="V"/>
  <variable units="ms" public_interface="in" name="time"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<component name="sodium_current_h_gate">
  <variable units="dimensionless" public_interface="out" name="h" initial_value="0.988"/>
  <variable units="per_ms" name="alpha_h"/>
  <variable units="per_ms" name="beta_h"/>
  <variable units="mV" public_interface="in" name="V"/>
  <variable units="ms" public_interface="in" name="time"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<component name="sodium_current_j_gate">
  <variable units="dimensionless" public_interface="out" name="j" initial_value="0.975"/>
  <variable units="per_ms" name="alpha_j"/>
  <variable units="per_ms" name="beta_j"/>
  <variable units="mV" public_interface="in" name="V"/>
  <variable units="ms" public_interface="in" name="time"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<component name="slow_inward_current">
  <variable units="uA_per_mm2" public_interface="out" name="i_s"
    cmeta:id="slow_inward_current_i_s"/>
  <variable units="mS_per_mm2" name="g_s" initial_value="9e-4"/>
  <variable units="mV" name="E_s"/>
  <variable units="concentration_units" name="Cai" initial_value="1e-4"/>
  <variable units="ms" public_interface="in" private_interface="out" name="time"/>
  <variable units="mV" public_interface="in" private_interface="out" name="V"/>
  <variable units="dimensionless" private_interface="in" name="d"/>
  <variable units="dimensionless" private_interface="in" name="f"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<component name="slow_inward_current_d_gate">
  <variable units="dimensionless" public_interface="out" name="d" initial_value="0.003"/>
  <variable units="per_ms" name="alpha_d"/>
  <variable units="per_ms" name="beta_d"/>
  <variable units="mV" public_interface="in" name="V"/>
  <variable units="ms" public_interface="in" name="time"/>
```

```
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<component name="slow_inward_current_f_gate">
  <variable units="dimensionless" public_interface="out" name="f" initial_value="0.994"/>
  <variable units="per_ms" name="alpha_f"/>
  <variable units="per_ms" name="beta_f"/>
  <variable units="mV" public_interface="in" name="V"/>
  <variable units="ms" public_interface="in" name="time"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<component name="time_dependent_outward_current" cmeta:id="time_dependent_outward_current">
  <variable units="uA_per_mm2" public_interface="out" name="i_x1"
    cmeta:id="time_dependent_outward_current_i_x1"/>
  <variable units="ms" public_interface="in" private_interface="out" name="time"/>
  <variable units="mV" public_interface="in" private_interface="out" name="V"/>
  <variable units="dimensionless" private_interface="in" name="x1"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<component name="time_dependent_outward_current_x1_gate">
  <variable units="dimensionless" public_interface="out" name="x1" initial_value="0.0001"/>
  <variable units="per_ms" name="alpha_x1"/>
  <variable units="per_ms" name="beta_x1"/>
  <variable units="mV" public_interface="in" name="V"/>
  <variable units="ms" public_interface="in" name="time"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<component name="time_independent_outward_current">
  <variable units="uA_per_mm2" public_interface="out" name="i_K1"
    cmeta:id="time_independent_outward_current_i_K1"/>
  <variable units="ms" public_interface="in" name="time"/>
  <variable units="mV" public_interface="in" name="V"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<component name="stimulus_protocol">
  <variable units="uA_per_mm2" public_interface="out" name="Istim"/>
  <variable units="ms" name="IstimStart" initial_value="10"/>
  <variable units="ms" name="IstimEnd" initial_value="50000"/>
  <variable units="uA_per_mm2" name="IstimAmplitude" initial_value="0.5"/>
  <variable units="ms" name="IstimPeriod" initial_value="1000"/>
  <variable units="ms" name="IstimPulseDuration" initial_value="1"/>
  <variable units="ms" public_interface="in" name="time"/>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <!- MathML goes here - removed for brevity ->
  </math>
</component>
<group>
  <relationship_ref relationship="containment"/>
  <component_ref component="membrane">
    <component_ref component="sodium_current">
      <component_ref component="sodium_current_m_gate"/>
      <component_ref component="sodium_current_h_gate"/>
      <component_ref component="sodium_current_j_gate"/>
    </component_ref>
    <component_ref component="slow_inward_current">
      <component_ref component="slow_inward_current_d_gate"/>
      <component_ref component="slow_inward_current_f_gate"/>
    </component_ref>
    <component_ref component="time_dependent_outward_current">
      <component_ref component="time_dependent_outward_current_x1_gate"/>
    </component_ref>
    <component_ref component="time_independent_outward_current"/>
  </component_ref>
</group>
<group>
  <relationship_ref relationship="encapsulation"/>
  <component_ref component="sodium_current">
```

```
    <component_ref component="sodium_current_m_gate"/>
    <component_ref component="sodium_current_h_gate"/>
    <component_ref component="sodium_current_j_gate"/>
  </component_ref>
  <component_ref component="slow_inward_current">
    <component_ref component="slow_inward_current_d_gate"/>
    <component_ref component="slow_inward_current_f_gate"/>
  </component_ref>
  <component_ref component="time_dependent_outward_current">
    <component_ref component="time_dependent_outward_current_x1_gate"/>
  </component_ref>
</group>
<connection>
  <map_components component_2="environment" component_1="membrane"/>
  <map_variables variable_2="time" variable_1="time"/>
</connection>
<connection>
  <map_components component_2="environment" component_1="sodium_current"/>
  <map_variables variable_2="time" variable_1="time"/>
</connection>
<connection>
  <map_components component_2="environment" component_1="slow_inward_current"/>
  <map_variables variable_2="time" variable_1="time"/>
</connection>
<connection>
  <map_components component_2="environment" component_1="time_dependent_outward_current"/>
  <map_variables variable_2="time" variable_1="time"/>
</connection>
<connection>
  <map_components component_2="environment" component_1="time_independent_outward_current"/>
  <map_variables variable_2="time" variable_1="time"/>
</connection>
<connection>
  <map_components component_2="sodium_current" component_1="membrane"/>
  <map_variables variable_2="V" variable_1="V"/>
  <map_variables variable_2="i_Na" variable_1="i_Na"/>
</connection>
<connection>
  <map_components component_2="slow_inward_current" component_1="membrane"/>
  <map_variables variable_2="V" variable_1="V"/>
  <map_variables variable_2="i_s" variable_1="i_s"/>
</connection>
<connection>
  <map_components component_2="time_dependent_outward_current" component_1="membrane"/>
  <map_variables variable_2="V" variable_1="V"/>
  <map_variables variable_2="i_x1" variable_1="i_x1"/>
</connection>
<connection>
  <map_components component_2="time_independent_outward_current" component_1="membrane"/>
  <map_variables variable_2="V" variable_1="V"/>
  <map_variables variable_2="i_K1" variable_1="i_K1"/>
</connection>
<connection>
  <map_components component_2="sodium_current_m_gate" component_1="sodium_current"/>
  <map_variables variable_2="m" variable_1="m"/>
  <map_variables variable_2="time" variable_1="time"/>
  <map_variables variable_2="V" variable_1="V"/>
</connection>
<connection>
  <map_components component_2="sodium_current_h_gate" component_1="sodium_current"/>
  <map_variables variable_2="h" variable_1="h"/>
  <map_variables variable_2="time" variable_1="time"/>
  <map_variables variable_2="V" variable_1="V"/>
</connection>
<connection>
  <map_components component_2="sodium_current_j_gate" component_1="sodium_current"/>
  <map_variables variable_2="j" variable_1="j"/>
  <map_variables variable_2="time" variable_1="time"/>
  <map_variables variable_2="V" variable_1="V"/>
</connection>
<connection>
  <map_components component_2="time_dependent_outward_current_x1_gate"
    component_1="time_dependent_outward_current"/>
  <map_variables variable_2="x1" variable_1="x1"/>
  <map_variables variable_2="time" variable_1="time"/>
  <map_variables variable_2="V" variable_1="V"/>
```

```
  </connection>
  <connection>
    <map_components component_2="slow_inward_current_d_gate" component_1="slow_inward_current"/>
    <map_variables variable_2="d" variable_1="d"/>
    <map_variables variable_2="time" variable_1="time"/>
    <map_variables variable_2="V" variable_1="V"/>
  </connection>
  <connection>
    <map_components component_2="slow_inward_current_f_gate" component_1="slow_inward_current"/>
    <map_variables variable_2="f" variable_1="f"/>
    <map_variables variable_2="time" variable_1="time"/>
    <map_variables variable_2="V" variable_1="V"/>
  </connection>
  <connection>
    <map_components component_2="stimulus_protocol" component_1="membrane"/>
    <map_variables variable_2="Istim" variable_1="Istim"/>
  </connection>
  <connection>
    <map_components component_2="stimulus_protocol" component_1="environment"/>
    <map_variables variable_2="time" variable_1="time"/>
  </connection>
</model>
```

D.3.3 HH52 Fragment CellML Shorthand Form

To reduce the verbosity when working with CellML models, the COR and OpenCOR environments provide a 'shorthand' CellML syntax that can be mapped to the standard XML form. This form uses inline expressions to represent model expressions rather than MathML, as shown below for a fragment of the HH52 model that encodes the 'membrane' component.

This shorthand form served as an inspiration for the *Ode* syntax, however the form is a syntactic change only and implements the same CellML semantics. As described in Sections 2.3.1.1 and 3.1, these suffer from a lack of structural and abstraction features, require the explicit mapping of elements, and have limited modular functionality compared to *Ode*.

```
def comp membrane as
  var V: millivolt {init: -75, pub: out};
  var E_R: millivolt {init: -75, pub: out};
  var Cm: microF_per_cm2 {init: 1};
  var time: millisecond {pub: in};
  var i_K: microA_per_cm2 {pub: in};
  var i_Stim: microA_per_cm2;

  def math as
    i_Stim = piecewise(
    case time >= 10 {unit: millisecond} and time <= 10.5 {unit: millisecond} then
    20 {unit: microA_per_cm2} else
    0 {unit: microA_per_cm2} );
    d(V)/d(time) = -((-(i_Stim) + i_K)) / Cm;
  enddef;
enddef;
```

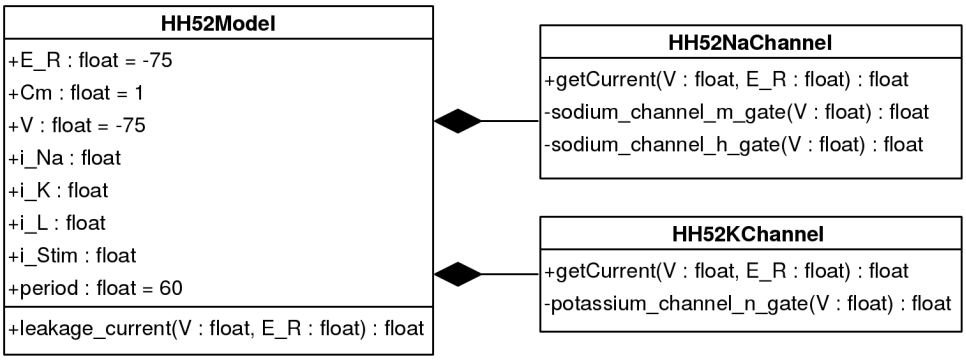


Figure D.2: UML diagram depicting the structure of the HH52 model using statically imported modules for each ion channel.

D.4 Deterministic Cardiac Models

This section lists several cardiac electrophysiological models created in *Ode* used during development and simulation studies that demonstrate DSL features to enable concise model creation, reuse and simulation. To increase the readability of these features, unit-annotations are only provided on select models where necessary.

D.4.1 *Ode* Hodgkin-Huxley Model

The following listing is one of many encodings of the Hodgkin-Huxley AP model [69] (see Section 2.1.3) in *Ode*, several variants of this model are used within the implementation benchmarks in Section 5.3.2. The system determines the potential across the cell membrane, V_m, from the currents generated from cellular ion channels. This particular variant utilises modules to represent the ion channels that are statically imported into the main `HH52Model`, as depicted in Fig. D.2

```
/* Channel - Sodium Current exporting IonChannel interface ******************/
module HH52NaChannel {
  export(get_current)

  component sodium_channel_m_gate(V) {
    val alpha_m = -0.1 * (V + 50) / (exp(-(V + 50) / 10) - 1)
    val beta_m = 4 * exp(-(V + 75) / 18)
    init m = 0.05
    ode {initVal : m, deltaVal : dm } = alpha_m*(1 - m) - beta_m*m
    return (m)
  }

  component sodium_channel_h_gate(V) {
    val alpha_h = 0.07 * exp(-(V + 75) / 20)
    val beta_h = 1 / (exp(-(V + 45) / 10) + 1)
    init h = 0.6
    ode {initVal : h, deltaVal: dh } = alpha_h*(1 - h) - beta_h * h
    return (h)
  }
  /* Returns the ionic current generated by this channel **********************/
  component get_current(V, E_R) {
    val m = sodium_channel_m_gate(V)
```

```
    val h = sodium_channel_h_gate(V)
    val g_Na = 120
    val E_Na = E_R + 115
    val i_Na = g_Na * pow(m, 3.0) * h * (V - E_Na)
    return (i_Na)
  }
}

/* Channel - Potassium Current exporting IonChannel interface ****************/
module HH52KChannel {
  export(get_current)

  component potassium_channel_n_gate(V) {
    val alpha_n = -0.01 * (V + 65) / (exp(-(V + 65) / 10) - 1)
    val beta_n = 0.125 * exp((V + 75) / 80)
    init n = 0.325
    ode {initVal : n, deltaVal: dn } = alpha_n*(1 - n) - beta_n*n
    return (n)
  }
  /* Returns the ionic current generated by this channel **********************/
  component get_current(V, E_R) {
    val n = potassium_channel_n_gate(V)
    val g_K = 36
    val E_K = E_R - 12
    val i_K = g_K * pow(n, 4.0) * (V - E_K)
    return (i_K)
  }
}

/* Main HH52 Cell Model *****************************************************/
module HH52Model {
  import CardiacElec.HH52.HH52NaChannel as NaChannel
  import CardiacElec.HH52.HH52KChannel as KChannel
  val E_R = -75
  val Cm = 1
  /* Returns the membrane leakage current ***********************************/
  component leakage_current(V, E_R) {
    val g_L = 0.3
    val E_L = E_R + 10.613
    val i_L = g_L * (V - E_L)
    return (i_L)
  }
  init V = -75
  // Channels currents - from modules
  val i_Na = NaChannel.get_current(V, E_R)
  val i_K = KChannel.get_current(V, E_R)
  val i_L = leakage_current(V, E_R)
  // Stimulus Current, 0.5ms duration every 60ms
  val period = 60
  val i_Stim = piecewise {
    time % period >= 10 and time % period <= 10.5 : 20,
    default: 0
  }
  // Calculate voltage from sum of currents
  ode { initVal : V, deltaVal : dV } = -(-i_Stim + i_Na + i_K + i_L) / Cm
}
```

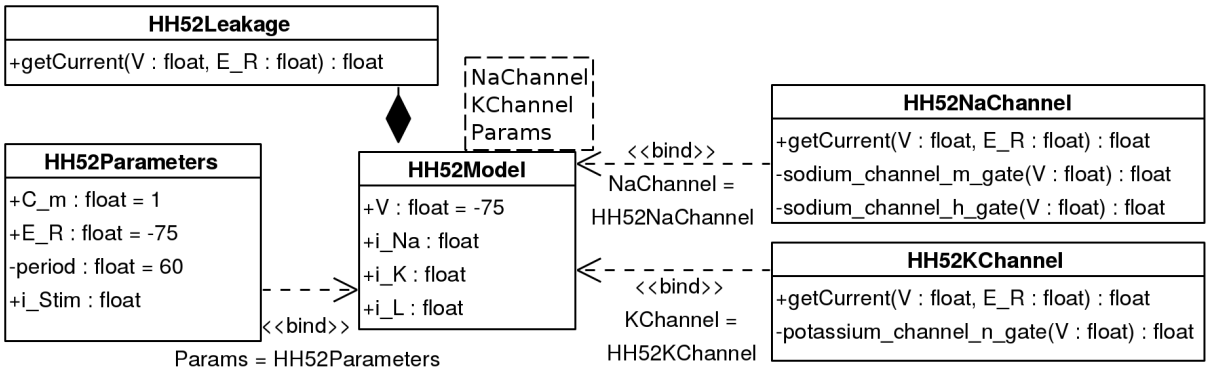


Figure D.3: UML diagram depicting the structure of the HH52 model using parameterised modules for ion channels and cell-level parameters.

D.4.2 Parameterised *Ode* Hodgkin-Huxley Model

The following listing is alternate encoding of the Hodgkin-Huxley AP model [69] used to describe the module system implementation in Section 4.3 and within the implementation benchmarks in Section 5.3.2. This listing modifies the earlier `HH52Model` model to parameterise and abstract the ion channels and cell-level parameters, as depicted in Fig. D.3. This model can be instantiated with channel modules to form, potentially custom, simulation-ready HH52 models.

```
/* Returns the membrane leakage current ************************************/
module HH52Leakage {
  component get_current(V, E_R) {
    val g_L = 0.3
    val E_L = E_R + 10.613
    val i_L = g_L * (V - E_L)
    return (i_L)
} }
/* Cell Membrane Parameters *************************************************/
module HH52Parameters {
  export (Cm, E_R, i_Stim)
  val Cm = 1
  val E_R = -75
  // Stimulus Current, 0.5ms duration every 60ms
  val period = 60
  val i_Stim = piecewise {
    time % period >= 10 and time % period <= 10.5 : 20,
    default: 0
} }
/* Main HH52 Cell Model *****************************************************/
module HH52Model(Params, NaChannel, KChannel) {
  import CardiacElec.HH52.HH52Leakage as Leakage
  init V = -75
  // Channels currents - from modules
  val i_Na = NaChannel.get_current(V, Params.E_R)
  val i_K = KChannel.get_current(V, Params.E_R)
  val i_L = Leakage.get_current(V, Params.E_R)
  // Calculate voltage from sum of currents
  ode { initVal : V, deltaVal : dV } = -(-Params.i_Stim + i_Na + i_K + i_L) / Params.Cm
}

import CardiacElec.HH52.HH52NaChannel as NaChannel
import CardiacElec.HH52.HH52KChannel as KChannel
import CardiacElec.HH52.HH52Parameters as Params
module HH52Original = HH52Model(Params, NaChannel, KChannel)
```

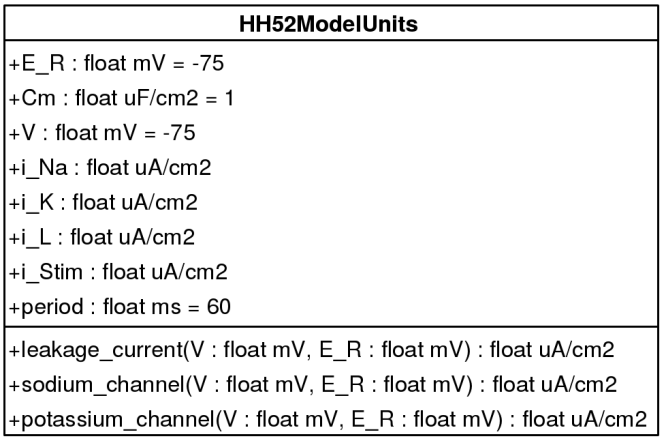


Figure D.4: UML diagram depicting the structure of a unit-checked HH52 model using nested subcomponents for each ion channel.

D.4.3 Unit-checked *Ode* Hodgkin-Huxley Model

The following listing is a unit-checked variant of the HH52 model used within simulations in Section 5.3.2. This particular model consists of a single flattened model module containing nested components to represent the cellular ion channels, as depicted in Fig. D.4.

```
/* Main HH52 Cell Model ********************************************************/
module HH52ModelUnits {
  val E_R = -75 { unit : mV }
  val Cm = 1 { unit : uF.cm^-2 }

  /* Channel - Sodium Current ***************************************************/
  component sodium_channel(V, E_R) {
    component sodium_channel_m_gate(V) {
      val alpha_m = -0.1 { unit : mV^-1 } * (V + 50 { unit : mV })
          / (exp(-(V + 50 { unit : mV }) / 10 { unit : mV }) - 1)
      val beta_m = 4 * exp(-(V + 75 { unit : mV }) / 18 { unit : mV})
      init m = 0.05
      ode {initVal : m, deltaVal : dm }
          = (alpha_m*(1 - m) - beta_m*m) * 1 { unit : ms^-1 }
      return (m)
    }

    component sodium_channel_h_gate(V) {
      val alpha_h = 0.07 * exp(-(V + 75 { unit : mV }) / 20 { unit : mV })
      val beta_h = 1 / (exp(-(V + 45 { unit : mV }) / 10 { unit : mV }) + 1)
      init h = 0.6
      ode {initVal : h, deltaVal: dh }
          = (alpha_h*(1 - h) - beta_h * h) * 1 { unit : ms^-1 }
      return (h)
    }
    // Returns the ionic current generated by this channel
    val m = sodium_channel_m_gate(V)
    val h = sodium_channel_h_gate(V)
    val g_Na = 120 { unit : mS.cm^-2 }
    val E_Na = E_R + 115 { unit : mV }
    val i_Na = g_Na * upow(m, 3) * h * (V - E_Na)
    return (i_Na { unit : current })
  }

  /* Channel - Potassium Current ************************************************/
  component potassium_channel(V, E_R) {
    component potassium_channel_n_gate(V) {
      val alpha_n = -0.01 { unit : mV^-1 } * (V + 65 { unit : mV })
          / (exp(-(V + 65 { unit : mV }) / 10 { unit : mV }) - 1)
      val beta_n = 0.125 * exp((V + 75 { unit : mV }) / 80 { unit : mV })
```

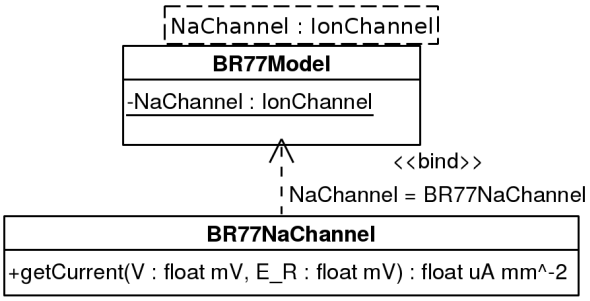


Figure D.5: UML diagram depicting the structure of the unit-checked BR77 model parameterised on I_{Na}, it follows the structure given in Fig. 4.13.

```
      init n = 0.325
      ode {initVal : n, deltaVal: dn }
          = (alpha_n*(1 - n) - beta_n*n) * 1 { unit : ms^-1 }
      return (n)
    }
    // Returns the ionic current generated by this channel
    val n = potassium_channel_n_gate(V)
    val g_K = 36 { unit : mS.cm^-2 }
    val E_K = E_R - 12 { unit : mV }
    val i_K = g_K * upow(n, 4) * (V - E_K)
    return (i_K { unit : current })
  }

  /* Returns the membrane leakage current ************************************/
  component leakage_current(V, E_R) {
    val g_L = 0.3 { unit : mS.cm^-2 }
    val E_L = E_R + 10.613 { unit : mV }
    val i_L = g_L * (V - E_L)
    return (i_L { unit : current })
  }

  // Cell Membrane Parameters
  init V = -75 { unit : mV }
  val i_Na = sodium_channel(V, E_R)
  val i_K = potassium_channel(V, E_R)
  val i_L = leakage_current(V, E_R)
  // Stimulus Current, 0.5ms duration every 60ms
  val period = 60 { unit : ms }
  val i_Stim = piecewise {
    time % period >= 10 { unit : ms } and time % period <= 10.5 { unit : ms }
        : 20 { unit : current },
    default: 0 { unit : current }
  }

  // Calculate voltage from sum of currents
  ode { initVal : V, deltaVal : dV }
      = (-(-1 * i_Stim + i_Na + i_K + i_L) / Cm) { unit : mV.ms^-1 }
}
```

D.4.4 Parameterised *Ode* BR77 Model

The following listings form a complete *Ode* version of the BR77 cardiac ventricular AP model [9] (see Section 2.1.3). The BR77 cellular model is parameterised on the I_{Na} channel model, as shown in the UML diagram in Fig. D.5; these specific models were used within the modular simulation study in Section 4.4.

The combined cellular and channel model can be compared to the CellML BR77 model

provided in Appendix D.3.2. The *Ode* model is significantly shorter and uses several features for structuring modular models, demonstrating the DSL's use and suitability for model development. The model is unit-checked, however a large number of unit annotations can be seen. This is for two reasons, firstly the original BR77 model was never unit-consistent, and thus in several places we have to to re-dimensionalise values to ensure successful verification. Secondly, the *Ode* units system strictly disallows ambiguous expressions, as explained in Section 3.3.2.3.

```
/* Main BR77 Cell Model ****************************************************/
module BR77Model(NaChannel) {
  // Cell parameters
  init V = -84.624 { unit : mV }
  val E_R = -84.624 { unit : mV }
  val C = 0.01 { unit : uF_per_mm2 }

  /* Channel - Slow Inward Current *******************************************/
  val g_s = 9e-4 { unit : mS_per_mm }
  init Cai = 1e-4
  // Channel gate - d
  init d = 0.003
  val alpha_d = (0.095 * exp(-(V - 5.0 { unit : mV }) / 100.0 { unit : mV }))
      / (1.0 + exp(-(V - 5.0 { unit : mV }) / 13.89 { unit : mV }))
  val beta_d = (0.07 * exp(-(V + 44.0 { unit : mV }) / 59.0 { unit : mV }))
      / (1.0 + exp((V + 44.0 { unit : mV }) / 20.0 { unit : mV }))
  ode { initVal : d } = (alpha_d * (1.0 - d) - beta_d * d) * 1 { unit : ms^-1}
  // Channel gate - f
  init f = 0.994
  val alpha_f = (0.012 * exp(-(V + 28.0 { unit : mV }) / 125.0 { unit : mV }))
      / (1.0 + exp((V + 28.0 { unit : mV }) / 6.67 { unit : mV }))
  val beta_f = (0.0065 * exp(-(V + 30.0 { unit : mV }) / 50.0 { unit : mV }))
      / (1.0 + exp(-(V + 30.0 { unit : mV }) / 5.0 { unit : mV }))
  ode { initVal : f } =  (alpha_f * (1.0-f) - beta_f * f) * 1 { unit : ms^-1}
  // Calculate channel current
  val E_s = (-82.3 - 13.0287 * log(Cai * 0.001)) * 1 { unit : mV }
  val i_s = (g_s * d * f * (V - E_s)) { unit : uA_per_mm }
  ode { initVal : Cai } = ((-0.01 * i_s / 1 { unit : uA_per_mm }) + 0.07
      * (0.0001 - Cai)) * 1 { unit : ms^-1}

  /* Time-dependent Outward Current ******************************************/
  // gate - x1
  init x1 = 0.0001
  val alpha_x1 = (0.0005 * exp((V + 50.0 { unit : mV }) / 12.1 { unit : mV}))
      / (1.0 + exp((V + 50.0 { unit : mV }) / 17.5 { unit : mV } ))
  val beta_x1 = (0.0013 * exp(-(V + 20.0 { unit : mV }) / 16.67 { unit : mV }))
      / (1.0 + exp(-(V + 20.0 { unit : mV }) / 25.0 { unit : mV }))
  ode { initVal : x1 } =
      (alpha_x1 * (1.0 - x1) - beta_x1 * x1) * 1 { unit : ms^-1 }
  // Calculate current
  val i_x1 = ((x1 * 0.008 * (exp(0.04 { unit : mV^-1 }
      * (V + 77.0 { unit : mV })) - 1.0)) / exp(0.04 { unit : mV^-1 }
      * (V + 35.0 { unit : mV }))) * 1 { unit : uA_per_mm }

  /* Time-independent Outward Current ****************************************/
  val i_K1 = (0.0035 * ((4.0 * (exp(0.04 { unit : mV^-1 }
      * (V + 85.0 { unit : mV })) - 1.0)) / (exp(0.08 { unit : mV^-1 }
      * (V + 53.0 { unit : mV })) + exp(0.04 { unit : mV^-1 }
      * (V + 53.0 { unit : mV }))) + (0.2 { unit : mV^-1 }
      * (V + 23.0 { unit : mV })) / (1.0 - exp(-0.04 { unit : mV^-1 }
      * (V + 23.0 { unit : mV }))))) * 1 { unit : uA_per_mm }

  // Stimulus Current, 1ms duration every 1000ms
  val i_StimAmplitude = 0.5 { unit : uA_per_mm }
  val i_StimPeriod = 1000 { unit : ms }
  val i_StimStart = 10 { unit : ms }
  val i_StimDuration = 1 { unit : ms }
  val i_Stim = piecewise{ ( time % i_StimPeriod >= i_StimStart) and
      (time % i_StimPeriod <= (i_StimStart + i_StimDuration)) : i_StimAmplitude
      , default :  0.0 { unit : uA_per_mm } }
```

```
  // Channel - Sodium Current from parameterised module
  val i_Na = NaChannel.getCurrent(V, E_R)

  // Calculate voltage from sum of currents
  ode { initVal : V } =
      ((i_Stim - (i_Na + i_s + i_x1 + i_K1)) / C) { unit : mV.ms^-1 }
}
```

```
/* Channel - Sodium Current exporting ion channel interface ******************/
module BR77NaChannel {
  export (getCurrent)
  // Channel gate - m
  component getMGate(V) {
    init m = 0.011
    val alpha_m = -(V + 47.0 { unit : mV }) / (exp(-0.10 { unit : mV^-1 }
        * (V + 47.0 { unit : mV })) - 1.0) * 1 { unit : mV^-1 }
    val beta_m = 40.0 * exp(-0.056 { unit : mV^-1 } * (V + 72.0 { unit : mV }))
    ode { initVal : m } = (alpha_m * (1.0 - m) - beta_m * m) * 1 { unit : ms^-1 }
    return (m)
  }
  // Channel gate - h
  component getHGate(V) {
    init h = 0.988
    val alpha_h = 0.126 * exp(-0.25 { unit : mV^-1 } * (V + 77.0 { unit : mV }))
    val beta_h = 1.7 / (exp(-0.082 { unit : mV^-1 }
        * (V + 22.5 { unit : mV })) + 1.0)
    ode { initVal : h } = (alpha_h * (1.0 - h) - beta_h * h) * 1 { unit : ms^-1 }
    return (h)
  }
  // Channel gate - j
  component getJGate(V) {
    init j = 0.975
    val alpha_j = (0.055 * exp(-0.25 { unit : mV^-1 }
        * (V + 78.0 { unit : mV }))) / (exp(-0.2 { unit : mV^-1 } * (V + 78.0 { unit : mV })) + 1.0)
    val beta_j = 0.3 / (exp(-0.1 { unit : mV^-1 } * (V + 32.0 { unit : mV })) + 1.0)
    ode { initVal : j } = (alpha_j * (1.0 - j) - beta_j * j) * 1 { unit : ms^-1 }
    return (j)
  }
  /* Returns the ionic current generated by this channel **********************/
  component getCurrent(V, E_R) {
    // Channel parameters
    val E_Na = 50 { unit : mV }
    val g_Na = 4e-2 { unit : mS_per_mm2 }
    val g_Nac = 3e-5 { unit : mS_per_mm2 }

    // Returns the ionic current generated by this channel
    val i_Na = ((g_Na * pow(getMGate(V), 3.0) * getHGate(V) * getJGate(V) + g_Nac)
        * (V - E_Na)) { unit : uA_per_mm2 }
    return (i_Na)
  }
}
```

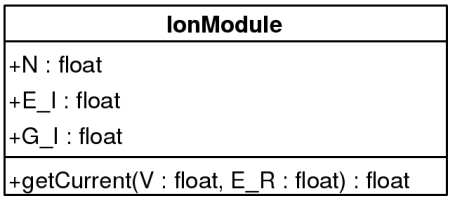


Figure D.6: UML class diagram illustrating structure of the *Ion*-generated HH52 K channel module interface.

D.5 Stochastic Cardiac Models

D.5.1 *Ion* HH52 Model

The listings in this section contain the output generated from translating the *Ion* representation of the Markov-formulation HH52 K^+ channel provided in Fig. 6.12. The channel is translated into *Ode* simulation-ready code using the ODE-, SDE-, and SSA-based representations. These modules were used within the case study in Section 6.3 to demonstrate hybrid-stochastic modelling of the HH52 cell model. Their interface is shown in Fig. D.6.

HH52 I_K *Ion* model translated to *Ode* utilising ODEs

```
module SimKChannelODE {
  val N = 1000
  val G_I = 36.0
  val E_I = -87.0
  /* Externally called component to generate channel current */
  component getCurrent(V, E_R, alpha, beta) {
    /* Setup initial values */
    init C1 = 0.532
    init C2 = 0.256
    init C3 = 0.123
    init C4 = 0.059
    init O = 0.030

    /* Temporary values to hold repeated/costly calculations */
    val _rate1 = (4*alpha)
    val _rate2 = (1*beta)
    val _rate3 = (3*alpha)
    val _rate4 = (2*beta)
    val _rate5 = (2*alpha)
    val _rate6 = (3*beta)
    val _rate7 = (1*alpha)
    val _rate8 = (4*beta)

    /* Setup state values (based on ODE/SDE/SSA form) */
    ode {initVal : C1} = -_rate1*C1+_rate2*C2
    ode {initVal : C2} = -(_rate3+_rate2)*C2+_rate4*C3+_rate1*C1
    ode {initVal : C3} = -(_rate5+_rate4)*C3+_rate6*C4+_rate3*C2
    ode {initVal : C4} = -(_rate7+_rate6)*C4+_rate8*O+_rate5*C3
    ode {initVal : O}  = -_rate8*O+_rate7*C4

    /* Calculate channel current */
    val current = G_I*(O)*(V-E_I)
    return current
} }
```

HH52 I_K *Ion* model translated to *Ode* utilising SDEs

```
module SimKChannelSDE {
  val N = 1000
  val G_I = 36.0
  val E_I = -87.0

  /* Externally called component to generate channel current */
  component getCurrent(V, E_R, alpha, beta) {
    /* Setup initial values */
    init C1 = 0.532
    init C2 = 0.256
    init C3 = 0.123
    init C4 = 0.059
    init O = 0.030

    /* Temporary values to hold repeated/costly calculations */
    val _n1 = 1/sqrt(N)
    val _rate1 = (4*alpha)
    val _rate2 = (1*beta)
    val _rate3 = (3*alpha)
    val _rate4 = (2*beta)
    val _rate5 = (2*alpha)
    val _rate6 = (3*beta)
    val _rate7 = (1*alpha)
    val _rate8 = (4*beta)

    /* Setup state values (based on ODE/SDE/SSA form) */
    val _prop1 = sqrt(abs(C1*_rate1+C2*_rate2))
    val _prop2 = sqrt(abs(C2*_rate3+C3*_rate4))
    val _prop3 = sqrt(abs(C3*_rate5+C4*_rate6))
    val _prop4 = sqrt(abs(C4*_rate7+O*_rate8))
    val _w1 = wiener
    val _w2 = wiener
    val _w3 = wiener
    val _w4 = wiener
    sde {initVal : C1, diffusion : _n1*(-_prop1*_w1)}
        = -_rate1*C1+_rate2*C2
    sde {initVal : C2, diffusion : _n1*(_prop1*_w1+-_prop2*_w2)}
        = -(_rate3+_rate2)*C2+_rate4*C3+_rate1*C1
    sde {initVal : C3, diffusion : _n1*(_prop2*_w2+-_prop3*_w3)}
        = -(_rate5+_rate4)*C3+_rate6*C4+_rate3*C2
    sde {initVal : C4, diffusion : _n1*(_prop3*_w3+-_prop4*_w4)}
        = -(_rate7+_rate6)*C4+_rate8*O+_rate5*C3
    sde {initVal : O, diffusion : _n1*(_prop4*_w4)}
        = -_rate8*O+_rate7*C4

    /* Calculate channel current */
    val current = G_I*(O)*(V-E_I)
    return current
  }
}
```

HH52 I_K *Ion* model translated to *Ode* utilising SSA-reactions

```
module SimKChannelSSA {
  /* Externally called component to generate channel current */
  val N = 1000
  val G_I = 36.0
  val E_I = -87.0

  component getCurrent(V, E_R, alpha, beta) {
    /* Setup initial values */
    init C1 = 532.0
    init C2 = 256.0
    init C3 = 123.0
    init C4 = 59.0
    init O = 30.0

    /* Temporary values to hold repeated/costly calculations */
    val _rate1 = (4*alpha)
    val _rate2 = (1*beta)
    val _rate3 = (3*alpha)
    val _rate4 = (2*beta)
    val _rate5 = (2*alpha)
    val _rate6 = (3*beta)
    val _rate7 = (1*alpha)
    val _rate8 = (4*beta)

    /* Setup state values (based on ODE/SDE/SSA form) */
    reaction {rate : _rate1} = C1 -> C2
    reaction {rate : _rate2} = C2 -> C1
    reaction {rate : _rate3} = C2 -> C3
    reaction {rate : _rate4} = C3 -> C2
    reaction {rate : _rate5} = C3 -> C4
    reaction {rate : _rate6} = C4 -> C3
    reaction {rate : _rate7} = C4 -> O
    reaction {rate : _rate8} = O -> C4

    /* Calculate channel current */
    val current = G_I*(O/N)*(V-E_I)
    return current
  }
}
```

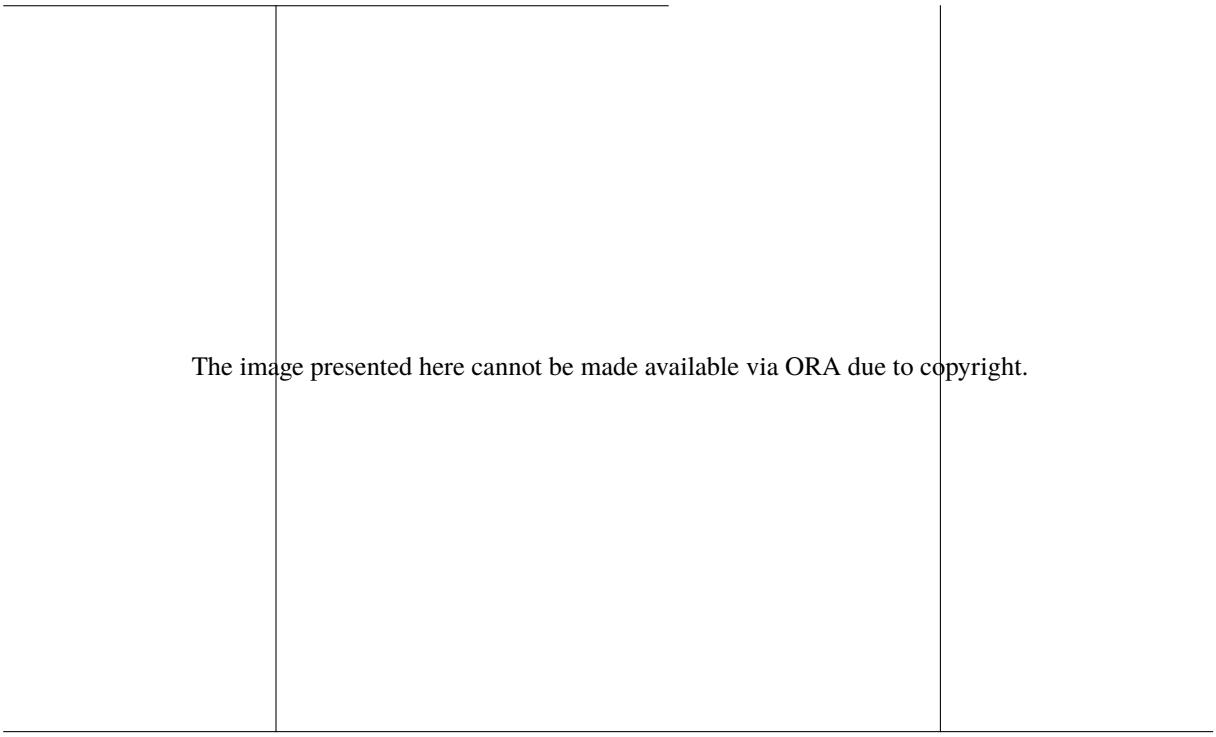


Figure D.7: Transition graph for the I_{Ks} channel model, open states are defined by O_1 and O_2. The transition rates may be found in [38] (Reproduced from Section II.B in [38]).

D.5.2 Decker I_{Ks} channel Model

The 17-state Decker I_{Ks} channel Markov-formulation ion channel model was used within the benchmarking studies undertaken in Sections 6.1.1.3 and 6.1.2.3 to test the efficiency of the SDE and SSA solvers respectively. Fig. D.7 reproduces the model state transitions as initially published in [38].

This model provided the initial inspiration for the *Ion* DSL and was widely used during the DSL development and testing. The following listing represents an *Ion* representation of the model transitions given in Fig. D.7, this can then be translated into simulation-ready *Ode* code using ODEs, SDEs, or SSA-reactions,

```
channel Decker_Ks {
  sim_type : ssa,
  density : 1000,
  equilibrium_potential : 2.2,
  channel_conductance : 3.3,
  initial_states: { C1 : 1.0, C2 : 0, C3 : 0, C4 : 0, C5 : 0, C6 : 0, C7 : 0,
                    C8 : 0, C9 : 0, C10 : 0, C11 : 0, C12 : 0, C13 : 0,
                    C14 : 0, C15 : 0, O1 : 0, O2 : 0 }
  additional_inputs: {alpha, beta, gamma1, delta, eta, theta, omega, psi}
  open_states : {O1, O2},
  transitions : {
// top row - l -> r
    {transition: C1 <-> C2, f_rate: 4*alpha, r_rate : 1*beta},
    {transition: C2 <-> C3, f_rate: 3*alpha, r_rate : 2*beta},
    {transition: C3 <-> C4, f_rate: 2*alpha, r_rate : 3*beta},
    {transition: C4 <-> C5, f_rate: 1*alpha, r_rate : 4*beta},
```

```
// 2nd row - l -> r
    {transition: C6 <-> C7, f_rate: 3*alpha, r_rate : 1*beta},
    {transition: C7 <-> C8, f_rate: 2*alpha, r_rate : 2*beta},
    {transition: C8 <-> C9, f_rate: 1*alpha, r_rate : 3*beta},
// 3nd row - l -> r
    {transition: C10 <-> C11, f_rate: 2*alpha, r_rate : 1*beta},
    {transition: C11 <-> C12, f_rate: 1*alpha, r_rate : 2*beta},
// 4nd row - l -> r
    {transition: C13 <-> C14, f_rate: 1*alpha, r_rate : 1*beta},
// top col
    {transition: C2 <-> C6, f_rate: 1*gamma1, r_rate : 1*delta},
    {transition: C3 <-> C7, f_rate: 2*gamma1, r_rate : 1*delta},
    {transition: C4 <-> C8, f_rate: 3*gamma1, r_rate : 1*delta},
    {transition: C5 <-> C9, f_rate: 4*gamma1, r_rate : 1*delta},
// 2nd col
    {transition: C7 <-> C10, f_rate: 1*gamma1, r_rate : 2*delta},
    {transition: C8 <-> C11, f_rate: 2*gamma1, r_rate : 2*delta},
    {transition: C9 <-> C12, f_rate: 3*gamma1, r_rate : 2*delta},
// 3rd col
    {transition: C11 <-> C13, f_rate: 1*gamma1, r_rate : 3*delta},
    {transition: C12 <-> C14, f_rate: 2*gamma1, r_rate : 3*delta},
// 4rd col
    {transition: C14 <-> C15, f_rate: 1*gamma1, r_rate : 4*delta},
// final/open state transitions
    {transition: C15 <-> O1, f_rate: theta, r_rate : eta},
    {transition: O1 <-> O2, f_rate: psi, r_rate : omega}
  }
}
```

Bibliography

[1] D. Abramson, J. Giddy, and L. Kotler. "High performance parametric modeling with Nimrod/G". In: *Parallel and Distributed Processing Symposium (IPDPS 2000)*. IEEE. 2000, pp. 520–528.

[2] Ada-Europe. *Ada 2012 language reference manual*.

[3] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Vol. 1009. Pearson/Addison Wesley, 2007.

[4] C. Alexander, S. Ishikawa, and M. Silverstein. "Pattern languages". In: *Center for Environmental Structure* 2 (1977).

[5] A.W. Appel. *Modern compiler implementation in ML*. Cambridge University Press, 1997.

[6] A.W. Appel. "SSA is functional programming". In: *ACM SIGPLAN Notices* 33.4 (Apr. 1998), pp. 17–20. ISSN: 03621340. DOI: 10.1145/278283.278285.

[7] H.P. Barendregt. *The Lambda Calculus: its syntax and semantics*. Vol. 103. Access Online via Elsevier, 1985.

[8] E. Bartocci, E.M. Cherry, J. Glimm, R. Grosu, S.A. Smolka, and F.H. Fenton. *Toward real-time simulation of cardiac dynamics*. Tech. rep. 2010.

[9] G.W. Beeler and H. Reuter. "Reconstruction of the action potential of ventricular myocardial fibres". In: *The Journal of physiology* 268.1 (1977), p. 177.

[10] N. Begeman. "Building an efficient JIT". In: *LLVM Developers Meeting*. 2008.

[11] A.J.C. Bik, M. Girkar, P.M. Grey, and X. Tian. "Automatic intra-register vectorization for the Intel® architecture". In: *International Journal of Parallel Programming* 30.2 (2002), pp. 65–98.

[12] G. Booch. *Object oriented analysis and design with applications*. Pearson, 2006.

[13] G.E.P Box and M.E. Muller. "A note on the generation of random normal deviates". In: *The Annals of Mathematical Statistics* 29.2 (1958), pp. 610–611.

[14] I.C. Bruce. "Evaluation of stochastic differential equation approximation of ion channel gating models." In: *Annals of biomedical engineering* 37.4 (Apr. 2009), pp. 824–38. ISSN: 1521-6047. DOI: 10.1007/s10439-009-9635-z.

[15] I.C. Bruce. "Implementation issues in approximate methods for stochastic Hodgkin-Huxley models." In: *Annals of biomedical engineering* 35.2 (Feb. 2007), 315–8; author reply 319. ISSN: 0090-6964. DOI: 10.1007/s10439-006-9174-9.

[16] F. Buschmann, K. Henney, and D. Schimdt. *Pattern-oriented software architecture: on patterns and pattern language*. Vol. 5. John Wiley & Sons, 2007.

[17] L. Cardelli. *Extensible records in a pure calculus of subtyping*. Digital Systems Research Center, 1992.

[18] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. *Modula-3 report (revised)*. Vol. 52. Digital Systems Research Center, 1989.

[19] J. Carro. "An improved human ventricular cell model for investigation of cardiac arrhythmias under hyperkalemic conditions". PhD thesis. Universidad Zaragoza, 2011, pp. 2010–2011.

[20] J. Carro, J.F. Rodríguez, P. Laguna, and E. Pueyo. "A human ventricular cell model for investigation of cardiac arrhythmias under hyperkalaemic conditions." In: *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences* 369.1954 (Nov. 2011), pp. 4205–32. ISSN: 1364-503X. DOI: 10.1098/rsta.2011.0127.

[21] S. Chellappa, F. Franchetti, and M. Püschel. *How to write fast numerical code: A small introduction*. Tech. rep. 2008, pp. 196–259.

[22] Y. Chen and X. Ye. *Projection onto a simplex*. Tech. rep. 1. 2011, pp. 1–7.

[23] G.R Christie, P.M.F. Nielsen, S.A. Blackett, C.P. Bradley, and P.J. Hunter. "FieldML: concepts and implementation". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 367.1895 (2009), pp. 1869–1884.

[24] G. Ciobanu. "Software verification of biomolecular systems". In: *Modelling in Molecular Biology* (2004), p. 106.

[25] G. Ciobanu, V. Ciubotariu, and B. Tanasa. "A pi-calculus model of the Na pump". In: *Genome Informatics* 471 (2002), pp. 469–471.

[26] C.E. Clancy and Y. Rudy. "Linking a genetic defect to its cellular phenotype in a cardiac arrhythmia". In: *Nature* 400.6744 (1999), pp. 566–569.

[27] S.D. Cohen and A.C. Hindmarsh. "CVODE, a stiff/nonstiff ODE solver in C". In: *Computers in physics* (1996).

[28] J.R. Collier, N.A.M. Monk, P.K. Maini, and J.H. Lewis. "Pattern formation by lateral inhibition with feedback: a mathematical model of delta-notch intercellular signalling". In: *Journal of Theoretical Biology* 183.4 (1996), pp. 429–446.

[29] J.P. Cooper. "Automatic validation and optimisation of biological models". PhD thesis. University of Oxford, 2008.

[30] J.P. Cooper and S. McKeever. "A model-driven approach to automatic conversion of physical units". In: *Software: Practice and Experience* 38.4 (2008), pp. 337–359. DOI: 10.1002/spe.

[31] J.P. Cooper and S. McKeever. "Experience report: a Haskell interpreter for CellML". In: *ACM SIGPLAN Notices* 42.9 (2007), pp. 247–250.

[32] J.P. Cooper, G.R. Mirams, and S.A. Niederer. "High-throughput functional curation of cellular electrophysiology models." In: *Progress in biophysics and molecular biology* 107.1 (Oct. 2011), pp. 11–20. ISSN: 1873-1732. DOI: 10.1016/j.pbiomolbio.2011.06.003.

[33] M. Courtot, N. Juty, C. Knüpfer, D. Waltemath, A. Zhukova, A. Dräger, et al. "Controlled vocabularies and semantics in systems biology". In: *Molecular systems biology* 7.1 (2011).

[34] D. Crockford. *The application/json media type for Javascript Object Notation (JSON)*. Tech. rep. 2006.

[35] A. Cuellar, P.F. Nielsen, M. Halstead, D. Bullivant, D. Nickerson, W. Hedley, et al. *CellML 1.1 specification*. The CellML Project, 2006.

[36] K. Czarnecki, J.T. O'Donnell, and J. Striegnitz. "DSL implementation in MetaOCaml, Template Haskell, and C++". In: *Domain-Specific Program Generation* (2004), pp. 1–22.

[37] C. Dangerfield. "Stochastic modelling of ion channels". PhD thesis. University of Oxford, 2012.

[38] K.F. Decker, J. Heijman, J.R. Silva, T.J. Hund, and Y. Rudy. "Properties and ionic mechanisms of action potential adaptation, restitution, and accommodation in canine epicardium (Supplementary Material)". In: *American Journal of Physiology-Heart and Circulatory Physiology* 296.4 (2009), pp. 1017–1026.

[39] K.F. Decker, J. Heijman, J.R. Silva, T.J. Hund, and Y. Rudy. "Properties and ionic mechanisms of action potential adaptation, restitution, and accommodation in canine epicardium." In: *American journal of physiology. Heart and circulatory physiology* 296.4 (Apr. 2009), H1017–26. ISSN: 0363-6135. DOI: 10.1152/ajpheart.01216.2008.

[40] L. Dematté and D. Prandi. "GPU computing for systems biology." In: *Briefings in bioinformatics* 11.3 (May 2010), pp. 323–33. ISSN: 1477-4054. DOI: 10.1093/bib/bbq006.

[41] D. DiFrancesco and D. Noble. "A model of cardiac electrical activity incorporating ionic pumps and concentration changes". In: *Philosophical Transactions of the Royal Society of London. B, Biological Sciences* 307.1133 (1985), p. 353.

[42] S.Y. Diallo, H. Herencia-zapana, J.J. Padilla, and A. Tolk. "Understanding interoperability". In: *Proceedings of the 2011 Emerging M&S Applications in Industry and Academia Symposium*. 2001, pp. 84–91.

[43] G. Dos Reis and B. Stroustrup. "Specifying C++ concepts". In: *ACM SIGPLAN Notices*. Vol. 41. 1. ACM, 2006, pp. 295–308.

[44] F. H Fenton, E. M. Cherry, and L. Glass. "Cardiac arrhythmia". In: *Scholarpedia* 3.7 (2008), p. 1665.

[45] H. Finkel. *Autovectorization with LLVM - Basic-Block Autovectorization*. Tech. rep. 2012, pp. 1–29.

[46] J. Fisher and T.A. Henzinger. "Executable cell biology". In: *Nature Biotechnology* 25.11 (Nov. 2007), pp. 1239–49. ISSN: 1087-0156. DOI: 10.1038/nbt1356.

[47] C. Flanagan, A. Sabry, B.F. Duba, and M. Felleisen. "The essence of compiling with continuations : Retrospective". In: *PLDI*. New York, New York, USA: ACM Press, 2003, pp. 237–247. ISBN: 0897915984. DOI: 10.1145/155090.155113.

[48] C. Flanagan, A. Sabry, B.F. Duba, and M. Felleisen. "The essence of compiling with continuations". In: *ACM SIGPLAN Notices* 28.6 (June 1993), pp. 237–247. ISSN: 03621340. DOI: 10.1145/173262.155113.

[49] M. Fowler. *Domain-specific languages*. Addison-Wesley Professional, 2010.

[50] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[51] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design patterns: Abstraction and reuse of object-oriented design". In: 1993.

[52] A. Garny, D.P. Nickerson, J.P. Cooper, R. Weber dos Santos, A.K Miller, S. McKeever, et al. "CellML and associated tools and techniques." In: *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences* 366.1878 (Sept. 2008), pp. 3017–43. ISSN: 1364-503X. DOI: 10.1098/rsta.2008.0094.

[53] A. Garny, D. Noble, P.J. Hunter, and P. Kohl. "Cellular Open Resource (COR): current status and future directions." In: *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences* 367.1895 (May 2009), pp. 1885–905. ISSN: 1364-503X. DOI: 10.1098/rsta.2008.0289.

[54] J.H. Gennari, M.L. Neal, and B.E. Carlson. "Integration of multi-scale biosimulation models via light-weight semantics". In: *Pacific Symposium on Biocomputing*. 2008, pp. 414–425.

[55] M. Gill, S. McKeever, and D.J. Gavaghan. "Model composition for biological mathematical systems". In: *International Conference on Model-Driven Engineering and Software Development (Modelsward 2014)*. 2014.

[56] M. Gill, S. McKeever, and D.J. Gavaghan. "Modular mathematical modelling of biological systems". In: *Symposium on Theory of Modeling and Simulation (TMS'12)*. 2012.

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

[58] D.T. Gillespie. "Approximate accelerated stochastic simulation of chemically reacting systems". In: *The Journal of Chemical Physics* 115.4 (2001), p. 1716. ISSN: 00219606. DOI: 10.1063/1.1378322.

[59] D.T. Gillespie. "Exact stochastic simulation of coupled chemical reactions". In: *The journal of physical chemistry* 93555.1 (1977), pp. 2340–2361.

[60] D.T. Gillespie. "Stochastic simulation of chemical kinetics." In: *Annual review of physical chemistry* 58 (Jan. 2007), pp. 35–55. ISSN: 0066-426X. DOI: 10.1146/annurev.physchem.58.032806.104637.

[61] D.T. Gillespie. "The chemical Langevin equation". In: *The Journal of Chemical Physics* 113.1 (2000), pp. 297–306.

[62] D. Goldberg. "What every computer scientist should know about floating-point arithmetic". In: *ACM Computing Surveys* 23.1 (Mar. 1991), pp. 5–48. ISSN: 03600300. DOI: 10.1145/103162.103163.

[63] E. Grandi, F.S. Pasqualini, and M.B. Bers. "A novel computational model of the human ventricular action potential and Ca transient". In: *Journal of molecular and cellular* 48.1 (2010), pp. 1–20. DOI: 10.1016/j.yjmcc.2009.09.019.A.

[64] J.L. Greenstein, R. Hinch, and R.L. Winslow. "Mechanisms of excitation-contraction coupling in an integrative model of the cardiac ventricular myocyte." In: *Biophysical journal* 90.1 (Jan. 2006), pp. 77–91. ISSN: 0006-3495. DOI: 10.1529/biophysj.105.065169.

[65] H.B. Henninger, S.P. Reese, A.E. Anderson, and J.A. Weiss. "Validation of computational models in biomechanics". In: *Proceedings of the Institution of Mechanical Engineers, Part H: Journal of Engineering in Medicine* 224.7 (2010), pp. 801–812.

[66] D.J. Higham. "Modeling and simulating chemical reactions". In: *SIAM Review* 50.2 (2008), p. 347. ISSN: 00361445. DOI: 10.1137/060666457.

[67] B. Hille. *Ion channels of excitable membranes*. 3rd Edition. Sinauer Associates, Inc., 2001.

[68] T. Hoare. "Science and Engineering: A collusion of cultures". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)* (June 2007), pp. 2–9. DOI: 10.1109/DSN.2007.87.

[69] A.L. Hodgkin and A.F. Huxley. “A quantitative description of membrane current and its application to conduction and excitation in nerve”. In: *The Journal of physiology* 117.4 (1952), p. 500.

[70] C. Hofer, K. Ostermann, and T. Rendel. “Polymorphic embedding of DSLs”. In: *Generative programming and component engineering*. 2008.

[71] L. Hood, J.R. Heath, M.E. Phelps, and B. Lin. “Systems biology and new technologies enable predictive and preventative medicine”. In: *Science Signaling* 306.5696 (2004), p. 640.

[72] S. Howard and S. Vassilios. “Multiscale Hy3S: hybrid stochastic simulation for supercomputers”. In: *BMC Bioinformatics* 21 (2006), pp. 1–21. DOI: 10.1186/1471-2105-7-93.

[73] M. Hucka, A. Finney, B.J. Bornstein, S.M. Keating, and B.E. Shapiro. “Evolving a lingua franca and associated software infrastructure for computational systems biology: the Systems Biology Markup Language (SBML) project”. In: *Systems Biology* (2004). DOI: 10.1049/sb.

[74] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test,and deployment automation*. Pearson Education, 2010.

[75] V. Iyer, R. Mazhari, and R.L. Winslow. “A computational model of the human left-ventricular epicardial myocyte.” In: *Biophysical journal* 87.3 (Sept. 2004), pp. 1507–25. ISSN: 0006-3495. DOI: 10.1529/biophysj.104.043299.

[76] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[77] A.J. Kennedy. “Compiling with continuations, continued”. In: *ACM SIGPLAN Notices* 42.9 (Oct. 2007), p. 177. ISSN: 03621340. DOI: 10.1145/1291220.1291179.

[78] A.J. Kennedy. “Dimension types”. In: *Programming Languages and Systems - ESOP’94* 788 (1994), pp. 348–362.

[79] A.J. Kennedy. “Relational parametricity and units of measure”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Vol. 1. January. 1997, pp. 442–455.

[80] H. Kim, A. Reuther, and J. Kepner. *Writing Parallel Parameter Sweep Applications with pMatlab*. Tech. rep. 2007.

[81] O. Kiselyov, S. Peyton Jones, and C. Shan. *Fun with type functions*. Tech. rep. 2010, pp. 1–35.

[82] G. Klingbeil, R. Erban, and M. Giles. *Parallel stochastic simulation using graphics processing units for the Systems Biology Toolbox for MATLAB Software usage guide*. Tech. rep., pp. 1–9.

[83] P.E. Kloeden and E. Platen. *Numerical solution of stochastic differential equations*. Springer, Berlin, 1999.

[84] D. Köhn and N. Le Novère. “SED-ML — an XML format for the implementation of the MIASE guidelines”. In: *Computational Methods in Systems Biology*. Springer. 2008, pp. 176–190.

[85] S. Larsen and S. Amarasinghe. “Exploiting superword level parallelism with multimedia instruction sets”. In: *Programming language design and implementation (PLDI)*. 2000.

[86] C. Lattner. *Introduction to the LLVM compiler system*. Tech. rep. 2008.

[87] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004.* (2004), pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[88] D. Leijen. "Extensible records with scoped labels". In: *Trends in Functional Programming* 5 (2005), pp. 297–312.

[89] X. Leroy. "A modular module system". In: *Journal of Functional Programming* 10.3 (May 2000), pp. 269–303. ISSN: 09567968. DOI: 10.1017/S0956796800003683.

[90] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Remy, and J. Vouillon. *The Objective Caml system release 3.12*. 2010.

[91] H. Li, Y. Cao, L.R. Petzold, and D.T. Gillespie. "Algorithms and software for stochastic simulation of biochemical reacting systems." In: *Biotechnology progress* 24.1 (2008), pp. 56–61. ISSN: 8756-7938. DOI: 10.1021/bp070255h.

[92] H. Li and L.R. Petzold. "Stochastic simulation of biochemical systems on the Graphics Processing Unit". In: *Bioinformatics* 0.x (2005), pp. 1–5.

[93] C.M. Lloyd, M.D.B. Halstead, and P.F. Nielsen. "CellML: its future, present and past". In: *Progress in Biophysics and Molecular Biology* 85 (2004), pp. 433–450. DOI: 10.1016/j.pbiomolbio.2004.01.004.

[94] C.H. Luo and Y. Rudy. "A dynamic model of the cardiac ventricular action potential. I. Simulations of ionic currents and concentration changes." In: *Circulation research* 74.6 (June 1994), pp. 1071–96. ISSN: 0009-7330.

[95] C.H. Luo and Y. Rudy. "A dynamic model of the cardiac ventricular action potential. II. Afterdepolarizations, triggered activity, and potentiation." In: *Circulation research* 74.6 (June 1994), pp. 1097–113. ISSN: 0009-7330.

[96] C.H. Luo and Y. Rudy. "A model of the ventricular cardiac action potential. Depolarization, repolarization, and their interaction". In: *Circulation research* 68.6 (1991), p. 1501.

[97] A. Mallavarapu and M. Thomson. "Programming with models: modularity and abstraction provide powerful capabilities for systems biology". In: *Interface* May (2009), pp. 1–15. DOI: 10.1098/rsif.2008.0205.

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

[99] B. Mélykúti, K. Burrage, and K.C. Zygalakis. "Fast stochastic simulation of biochemical reaction systems by alternative formulations of the chemical Langevin equation." In: *The Journal of chemical physics* 132.16 (Apr. 2010), p. 164109. ISSN: 1089-7690. DOI: 10.1063/1.3380661.

[100] B. Meyer. "Applying design by contract". In: *IEEE Computer* 25.10 (2002), pp. 40–51.

[101] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML: revised 1997*. The MIT Press, 1997.

[102] H. Mino, J.T. Rubinstein, and J.A. White. "Comparison of algorithms for the simulation of action potentials with stochastic sodium channels". In: *Annals of Biomedical Engineering* 30.4 (Apr. 2002), pp. 578–587. ISSN: 0090-6964. DOI: 10.1114/1.1475343.

[103] J.D. Murray. *Mathematical biology*. 3rd Edition. Vol. I. An Introduction. Springer, 2002.

[104] F.Z. Nardelli. *Objective Caml module system*. Tech. rep. 2005.

[105] S.A. Niederer, M. Fink, D. Noble, and N.P. Smith. "A meta-analysis of cardiac electrophysiology computational models." In: *Experimental physiology* 94.5 (May 2009), pp. 486–95. ISSN: 1469-445X. DOI: 10.1113/expphysiol.2008.044610.

[106] D. Noble. "A modification of the Hodgkin—Huxley equations applicable to Purkinje fibre action and pacemaker potentials". In: *The Journal of Physiology* 160.2 (1962), p. 317. ISSN: 0022-3751.

[107] D. Noble and Y. Rudy. "Models of cardiac ventricular action potentials: iterative interaction between experiment and simulation". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 359.1783 (June 2001), pp. 1127–1142. ISSN: 1364-503X. DOI: 10.1098/rsta.2001.0820.

[108] B. Novak, Z. Pataki, A. Ciliberto, and J.J. Tyson. "Mathematical model of the cell division cycle of fission yeast". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 11.1 (2001), pp. 277–286.

[109] T. O'Hara, L. Virág, A. Varró, and Y. Rudy. "Simulation of the undiseased human cardiac ventricular action potential: model formulation and experimental validation." In: *PLoS computational biology* 7.5 (May 2011), e1002061. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1002061.

[110] P. Orio and D. Soudry. "Simple, fast and accurate implementation of the diffusion approximation algorithm for stochastic ion channels with multiple states." In: *PloS one* 7.5 (Jan. 2012), e36670. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0036670.

[111] J. Pahle. "Biochemical simulations: stochastic, approximate stochastic and hybrid approaches." In: *Briefings in bioinformatics* 10.1 (Jan. 2009), pp. 53–64. ISSN: 1477-4054. DOI: 10.1093/bib/bbn050.

[112] F. Panneton, P. L'ecuyer, and M. Matsumoto. "Improved long-period generators based on linear recurrences modulo 2". In: *ACM Transactions on Mathematical Software (TOMS)* 32.1 (2006), pp. 1–16.

[113] D.A. Patterson and J.L. Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2009.

[114] M. Pedersen and G.D. Plotkin. "A language for biochemical systems: design and formal specification". In: *Transactions on computational systems biology XII*. Springer, 2010, pp. 77–145.

[115] L.R. Petzold and U.M. Ascher. *Computer methods for ordinary differential equations and differential-algebraic equations*. Vol. 61. Siam, 1998.

[116] S. Peyton Jones. *Haskell 98 language and libraries — the revised report*. Cambridge, England: Cambridge University Press, 2003.

[117] S. Peyton Jones. *The implementation of functional programming languages*. Prentice-Hall, Inc., 1987.

[118] A. Phillips and L. Cardelli. "Efficient, correct simulation of biological processes in the stochastic pi-calculus". In: *Computational methods in systems biology*. 2007, pp. 184–199.

[119] B.C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[120] J. Pitt-Francis, P. Pathmanathan, M.O. Bernabeu, R. Bordas, J.P. Cooper, A.G. Fletcher, et al. "Chaste: A test-driven approach to software development for biological modelling". In: *Computer Physics Communications* 180.12 (Dec. 2009), pp. 2452–2471. ISSN: 00104655. DOI: 10.1016/j.cpc.2009.07.019.

[121] C. Priami, A. Regev, E. Shapiro, and W. Silverman. "Application of a stochastic name-passing calculus to representation and simulation of molecular processes". In: *Information Processing Letters* 80 (2001), pp. 25–31.

[122] A. Regev and E. Shapiro. “Cells as computation”. In: *Nature* 419.September (2002), p. 343.

[123] B. J. Roth. “Bidomain model”. In: *Scholarpedia* 3.4 (2008), p. 6221.

[124] Y. Rudy and J.R. Silva. “Computational biology in the study of cardiac ion channels and cell electrophysiology”. In: *Quarterly reviews of biophysics* 39.1 (July 2006), pp. 57–116. DOI: 10.1016/j.bbi.2008.05.010.

[125] H.M. Sauro, T.T. Karlsson, M. Swat, M. Galdzicki, and A. Somogyi. “libRoadRunner: A high performance SBML compliant simulator”. In: *bioRxiv* (2013).

[126] L.F. Shampine and S. Thompson. “Initial value problems”. In: *Scholarpedia* 2.2 (2007), p. 2861.

[127] T. Sheard and S. Peyton Jones. “Template meta-programming for Haskell”. In: *ACM SIGPLAN Notices* (2002).

[128] J.R. Silva, H. Pan, D. Wu, A. Nekouzadeh, K.F. Decker, J. Cui, et al. “A multiscale model linking ion-channel molecular dynamics and electrostatics to the cardiac action potential.” In: *Proceedings of the National Academy of Sciences of the United States of America* 106.27 (July 2009), pp. 11102–6. ISSN: 1091-6490. DOI: 10.1073/pnas.0904505106.

[129] I. Sommerville. *Software engineering*. Addison Wesley, 2010.

[130] M. Sperber, R. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. “Revised report on the algorithmic language Scheme”. In: *Journal of Functional Programming* 19.S1 (2009), pp. 1–301.

[131] D. Syme and J. Margetson. *The F# programming language*. 2010.

[132] Q.M. Tran. *The OCaml module system*. 2010.

[133] K.H.W.J. ten Tusscher, D. Noble, P.J. Noble, and A.V. Panfilov. “A model for human ventricular tissue.” In: *American journal of physiology. Heart and circulatory physiology* 286.4 (Apr. 2004), H1573–89. ISSN: 0363-6135. DOI: 10.1152/ajpheart.00794.2003.

[134] K.H.W.J. ten Tusscher and A.V. Panfilov. “Alternans and spiral breakup in a human ventricular tissue model.” In: *American journal of physiology. Heart and circulatory physiology* 291.3 (Sept. 2006), H1088–100. ISSN: 0363-6135. DOI: 10.1152/ajpheart.00109.2006.

[135] J.J. Tyson, K.C. Chen, and B. Novak. “Sniffers, buzzers, toggles and blinkers: dynamics of regulatory and signaling pathways in the cell”. In: *Current Opinion in Cell Biology* 15.2 (Apr. 2003), pp. 221–231. ISSN: 09550674. DOI: 10.1016/S0955-0674(03)00017-6.

[136] P. Wadler. “The essence of functional programming”. In: *POPL*. 1992.

[137] M. Wand and P. O’Keefe. “Automatic dimensional inference”. In: *Computational Logic-Essays in Honor of Alan Robinson*. 1991, pp. 479–483.

[138] S. Wang, S. Liu, M.J. Morales, H.C. Strauss, and R.L. Rasmusson. “A quantitative analysis of the activation and inactivation kinetics of HERG expressed in Xenopus oocytes.” In: *The Journal of physiology* 502 (Pt 1 (July 1997), pp. 45–60. ISSN: 0022-3751.

[139] International Bureau of Weights and Measures. *The international System of Units (SI)*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2001.

[140] D.J. Wilkinson. "Stochastic modelling for quantitative description of heterogeneous biological systems." In: *Nature reviews. Genetics* 10.2 (Feb. 2009), pp. 122–33. ISSN: 1471-0064. DOI: 10.1038/nrg2509.

[141] S.M. Wimalaratne, M.D.B. Halstead, C.M. Lloyd, M.T. Cooling, E.J. Crampin, and P.F. Nielsen. "Facilitating modularity and reuse: guidelines for structuring CellML 1.1 models by isolating common biophysical concepts." In: *Experimental physiology* 94.5 (May 2009), pp. 472–85. ISSN: 1469-445X. DOI: 10.1113/expphysiol.2008.045161.