Automatic Validation and Optimisation of Biological Models

Jonathan Cooper
St John's College

Computational Biology Research Group
Computing Laboratory
University of Oxford

Trinity Term 2008

This thesis is submitted to the Computing Laboratory, University of Oxford, for the degree of Doctor of Philosophy. This thesis is entirely my own work, and, except where otherwise indicated, describes my own research.

Jonathan Cooper
St John's College

Doctor of Philosophy
Trinity Term 2008

Automatic Validation and Optimisation of Biological Models

Abstract

Simulating the human heart is a challenging problem, with simulations being very time consuming, to the extent that some can take days to compute even on high performance computing resources. There is considerable interest in computational optimisation techniques, with a view to making whole-heart simulations tractable. Reliability of heart model simulations is also of great concern, particularly considering clinical applications. Simulation software should be easily testable and maintainable, which is often not the case with extensively hand-optimised software. It is thus crucial to automate and verify any optimisations.

CellML is an XML language designed for describing biological cell models from a mathematical modeller's perspective, and is being developed at the University of Auckland. It gives us an abstract format for such models, and from a computer science perspective looks like a domain specific programming language. We are investigating the gains available from exploiting this viewpoint. We describe various static checks for CellML models, notably checking the dimensional consistency of mathematics, and investigate the possibilities of provably correct optimisations. In particular, we demonstrate that partial evaluation is a promising technique for this purpose, and that it combines well with a lookup table technique, commonly used in cardiac modelling, which we have automated.

We have developed a formal operational semantics for CellML, which enables us to mathematically prove the partial evaluation of CellML correct, in that optimisation of models will not change the results of simulations. The use of lookup tables involves an approximation, thus introduces some error; we have analysed this using *a posteriori* techniques and shown how it may be managed.

While the techniques could be applied more widely to biological models in general, this work focuses on cardiac models as an application area. We present experimental results demonstrating the effectiveness of our optimisations on a representative sample of cardiac cell models, in a variety of settings.

Contents

Acknowledgements

Completing this thesis would not have been possible without help and support from many quarters.

Firstly, much gratitude is due to my supervisors, David Gavaghan, Steve McKeever and Jonathan Whiteley, for their consistent support and sage advice, helpful feedback and encouragement, and for giving me the freedom to pursue my own ideas. I would particularly like to thank Steve for introducing me to the field of partial evaluation, and Jon for teaching me about a posteriori error analysis.

Dave also deserves thanks for accepting me into the Life Sciences Interface Doctoral Training Centre, without which I would never have embarked on this work. I learnt many things during my time there, not only about physiology and mathematical modelling, but also how to present my ideas on paper and in person. The DTC also provided me with many good friends and colleagues.

Others at work have also supported me, not least the team involved in development of the Chaste framework. Thank you all for great times spent coding together. Members of the Computational Biology and Software Engineering groups have also been helpful, and this thesis is undoubtedly stronger for discussions with them.

My research was funded through the Integrative Biology Project, with money from EPSRC. I appreciate their willingness to fund interdisciplinary work.

Factors outside of work have also played a crucial role. I'd like to thank my friends, especially those who had the misfortune to live with me. Many thanks also to my parents, for their excellent care over the years. Most importantly of all, I would like to thank my wonderful wife, for looking after me so well these last two years, and for helping me through the stress of writing up, despite being exhausted herself.

Finally, thanks be to God, for his unceasing goodness, and for giving me the skills to pursue a subject I enjoy.

1 Introduction

Computer simulations have long played an important role in the physical sciences. Mathematical modelling of physical systems can aid our understanding, elucidating the underlying mechanisms which give rise to the behaviour we observe. The use of these techniques in the life sciences community has been more limited however, mostly due to the fact that biological systems are incredibly complex, and so simulating anything but the simplest systems has until recently been computationally intractable. The use of quantitative mathematical models to describe the behaviour of biological systems is now becoming increasingly common, however, with many groups investigating different systems. We consider the modelling of the heart in particular, as this is a field that has benefited from almost 50 years of research (Noble, 1960), starting with models of single cells.

Understanding the human heart is a fascinating and vital subject. Heart problems are a major cause of death in developed countries (World Health Organisation, 2008), and much effort has been put into elucidating the causes of heart disease in the hope of developing cures (see e.g. Noble, 2004; Rodriguez et al., 2006). Also, many drugs have adverse side effects on the heart, primarily due to interactions with some ionic channels potentially leading to fatal arrhythmias, which drug companies would like to avoid (see e.g. Noble, 2008).

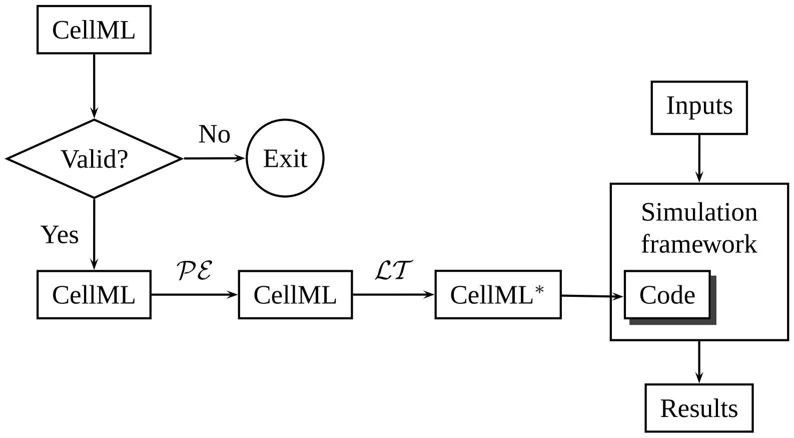
Computer simulations have played an important part in this research (Noble, 2002), correctly reproducing many features of both single cell and multicellular behaviour, and providing insight into the underlying causes of medical conditions in order to determine novel treatments. The idea of 'an indestructible test bed on which any theory [of how the heart works] may be safely tested and results analysed in minute detail' (Vigmond et al., 2003) is an attractive one. Much valuable work has been done, but even within the limits imposed by the nature of modelling there is still some way to go before this is realised. This is in large part due to the complexity of the computation required, which in turn derives from the complexity of the heart. The physiology of the heart, and how this may be modelled mathematically, is described in Sections 2.1 and 2.2.

This thesis is focused on models of single cardiac cells, a sample of which is presented in Section 2.2. These models can be described using CellML (Lloyd et al., 2004), an XML-based (and hence computer-readable) language described in Section 2.3. CellML was designed as a format for exchanging models between researchers. Our contribution has been to view it also as a programming language and explore the ideas suggested by this paradigm. We have thus defined a formal semantics for the language (Chapter 3), investigated suitable static checks for validating models (Chapter 4), as well as implemented and proved the correctness of some optimisation techniques (Chapters 5 and 6) enabling faster simulation of the models. The thesis structure thus somewhat mirrors program compilation and execution—validation, optimisation, then generation of experimental results (Chapter 7)—as shown in Figure 1.1.

The need for *automatic* optimisation is strongly demonstrated by two trends evident in our sample of cell models (see Section 2.2.13). As the available experimental data increases, models become more detailed and complex, and hence the need for effective optimisation increases. Also, the number of models in use increases, and so applying optimisations manually is not a good long term solution. The optimisation work in this thesis is thus both relevant and timely.

Another key theme is the requirement of confidence in the *correctness* of optimisations, in order to have confidence that the results of our simulations are valid derivations from our models. If simulation results differ from empirical data, it is important to know whether the mathematical model or the simulation code is at fault. Obviously, as simulation develops to

Figure 1.1 The programming language paradigm for CellML: validation, compilation, and execution. $\mathcal{PE}$ and $\mathcal{LT}$ are optimisation techniques described in Chapters 5 and 6 respectively, and implemented as source-to-source transformations. We need to extend CellML slightly to represent lookup tables, hence the asterisk after that transformation.



the point where it is of direct clinical relevance, reliability of the results will be even more important. Hence we need to be able to show that any transformations we apply to models do not change the results of simulations of the models in any significant way. In order to do this, we need to have a good definition of the *meaning* of a model, in order to prove that this is invariant under our transformations. This is the main driving force behind our development of a semantics for CellML that is mathematically tractable (Cooper and McKeever, 2007).

The optimisations we apply are not novel in and of themselves. Partial evaluation ($\mathcal{PE}$) has long been studied within the computer science community (Jones et al., 1993), and lookup tables ($\mathcal{LT}$) have been used in hand-optimised cardiac models. Rather, the novelty comes either through their application to a new class of problem (in the case of $\mathcal{PE}$; Cooper et al. 2006) or through their automation and the analysis of correctness.

In partial evaluation, expressions are separated into those which need only be computed once, and those which must be recomputed after every time step of the simulation (because the values of variables in the equation change with time). A new model is then produced from the original, in which as much of the model as possible has been pre-computed.

Lookup tables are used to pre-compute the values of expressions that would otherwise be repeatedly calculated. Several expressions in most cardiac electrophysiological cell models (notably those controlling the gating variables in Hodgkin–Huxley formulations of ion channels) contain only one variable: the membrane potential, V. They also typically contain exponential functions, which are expensive to compute. Under normal physiological conditions, the membrane potential V usually lies between $-100\,\mathrm{mV}$ and $50\,\mathrm{mV}$, and so a table can be generated of pre-computed values of each suitable expression for potentials within this range. Then, given any membrane potential within the range, a value for each expression can quickly be computed using linear interpolation between two entries of the lookup table, which is faster than computing an exponential directly.

The results of applying these two optimisation techniques to our sample of cell models are shown in Chapter 7. We typically see 3 to 4-fold speedups for recent models when both techniques are used.

Finally, in Chapter 8 we summarise the main contributions of the thesis, and consider the place of this work within the wider research context, looking towards future research directions.

2

Background

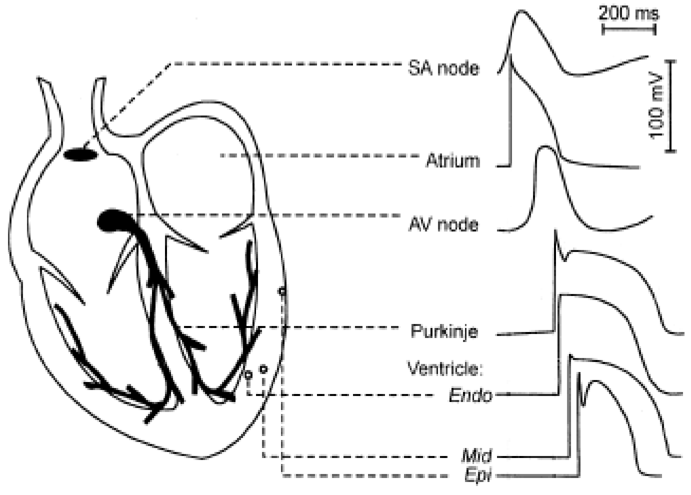
This chapter presents the background to the work of this thesis. In Sections 2.1 and 2.2 we give a brief introduction to the field of heart modelling, and thus give the context in which our research is placed. The former section looks at such modelling in general, while the latter section focuses on single cells.

Section 2.3 describes the modelling language CellML. Firstly we motivate its existence, and then in Section 2.3.1 we introduce the main constructs in the language, giving examples of their usage. Since the XML syntax of CellML is very verbose, we also define a compact syntax for use within the thesis.

2.1 Cardiac modelling

The beating of the heart is caused by myocytes—cardiac cells—contracting in a regular pattern. Myocytes are rod-like muscle cells approximately $100\,\mu\mathrm{m}$ long and $20\,\mu\mathrm{m}$ in diameter, the membrane of which responds characteristically to electrical stimulation by producing an electrical impulse called an action potential and triggering a contraction. Myocytes in different parts of the heart will respond differently (see Figure 2.1), and this together with the arrangement of cells within the heart determines how an action potential propagates through the whole organ. It is thus the flow of electric current around the heart that regulates the heart beat. Ab-

Figure 2.1 The main features of the heart, with regards to electrophysiology. The graphs show typical variation in action potential shape and timing for different classes of myocyte.



normal current patterns lead to life-threatening arrhythmias and fibrillation, the worst effect of which is blood not being pumped, causing cardiac arrest.

Under normal conditions, an action potential (AP) is initiated in the sino-atrial node (SAN) or pacemaker, which consists of self-exciting cells that are capable of autonomously producing a regular electrical impulse.[1] The excitation wave then propagates through the atria, passing from cell to cell via intercellular gap junctions. This triggers atrial contraction, thus filling the ventricles with blood. The atria are electrically isolated from the ventricles, except via the atrioventricular (AV) node, which delays the AP briefly. This ensures that the atria have ejected their blood into the ventricles before the ventricles contract. Another important property of the AV node is that the more frequently it is stimulated, the slower it conducts. This prevents rapid conduction to the ventricles in cases of atrial fibrillation.

The contraction of the ventricles is coordinated so as to maximise the pressure with which blood is forced through the circulation. From the AV node, an AP propagates along Purkinje fibres to the apex of the heart, then rapidly upwards through the ventricular walls. Contraction of the ventricles thus begins at the apex, progressively squeezing blood towards the arterial

[1]Certain other regions, notably around the AV node and the Purkinje fibres, are also capable of self-excitation if the SAN fails.

exits. The AP is sustained to prevent premature relaxation, maintaining initial contraction until the entire myocardium has had time to depolarize and contract completely.

For more information on heart physiology, see Sherwood (2001, Chapter 9) or Carmeliet and Vereecke (2002).

2.1.1 Mathematical modelling

Historically, mathematical modelling of the heart began with modelling the electrical behaviour of single myocytes, as this is where the bulk of experimental data was available on which to base models (Noble, 1960). Based on the insights of Hodgkin and Huxley into the squid giant axon (Hodgkin and Huxley, 1952), these models treat a cell like an electrical circuit, considering currents flowing across the cell membrane, and the effect of these currents on the transmembrane potential. Mathematically, this circuit is represented by a system of ordinary differential equations (ODEs). Simulating the models (for instance, to derive predictions, which can be tested against experiments) is then done by solving the ODE system using some numerical algorithm.

In Section 2.2 the main features of these cell models are introduced through consideration of a series of seminal models. Firstly, however, we consider the larger context—modelling a block of cardiac tissue, or indeed the whole organ. This is a complex challenge and the subject of ongoing research (see e.g. Hunter et al., 2003; Kerckhoffs et al., 2006). Within a tissue or organ level model, models of the individual cells need to be incorporated and coupled together in some fashion, in order to examine how the electrical activation of one cell affects its neighbours.

The most commonly used models of this electrical coupling are the monodomain and bidomain equations. The set of bidomain equations is currently the most complete mathematical model of electrical propagation, and was first applied to cardiac tissue in 1978 (Tung, 1978; Miller III and Geselowitz, 1978; see also Sepulveda et al., 1989). It treats cardiac tissue as an intracellular domain and an extracellular domain separated by an excitable membrane. These domains are superimposed in space; where no cell is present, the intracellular domain is empty.

The electrical activity in each domain is modelled by a partial differential equation (PDE), and these are coupled together by the ODE systems representing current flow across the cell membranes separating the domains. The complete system is thus given by:

$$\chi\left(C_m\frac{\partial V_m}{\partial t}+I_{\text{ion}}(V_m,\mathbf{u})\right)-\nabla\cdot\left(\sigma_i\nabla\left(V_m+\phi_e\right)\right) = 0, \tag{2.1.1}$$

$$\nabla\cdot\left((\sigma_i+\sigma_e)\nabla\phi_e+\sigma_i\nabla V_m\right) = 0, \tag{2.1.2}$$

$$\frac{\partial\mathbf{u}}{\partial t} = \mathbf{f}(\mathbf{u},V_m), \tag{2.1.3}$$

where:

- V_m is the transmembrane potential, i.e. the difference between the potentials in the intracellular and extracellular domains, and is the primary variable of interest;
- ϕ_e is the potential in the extracellular domain;
- χ is the surface-to-volume ratio;
- C_m is the membrane capacitance per unit area;
- σ_i is the intracellular conductivity tensor;
- σ_e is the extracellular conductivity tensor;
- $\mathbf{f}$ is the (vector-valued) function giving the ODE system representing the cell model, which must be solved at each point in space;
- I_{ion} is the ionic current across the membrane, also given by the cell model; and
- $\mathbf{u}$ is a vector of dependent variables in the cell model ODE system.

Suitable boundary conditions are given by

$$\left(\sigma_i\nabla\left(V_m+\phi_e\right)\right)\cdot\mathbf{n} = I_{s_i}, \tag{2.1.4}$$

$$\left(\sigma_e\nabla\phi_e\right)\cdot\mathbf{n} = I_{s_e}, \tag{2.1.5}$$

where $\mathbf{n}$ is the outward pointing normal vector to the boundary, and I_{s_i} and I_{s_e} give the external stimulus current per unit area applied to the intracellular and extracellular boundaries, respectively.

Under certain conditions (by assuming that the extracellular domain is infinitely conducting, or the two domains are equally anisotropic), ϕ_e may be eliminated from the bidomain equations, replacing Equations (2.1.1) and (2.1.2) by the single equation

$$\chi \left(C_m \frac{\partial V_m}{\partial t} + I_{\text{ion}}(V_m, \mathbf{u}) \right) - \nabla \cdot (\sigma \nabla V_m) = 0, \tag{2.1.6}$$

with a single 'bulk conductivity tensor' σ. Together with Equation (2.1.3) this forms the *monodomain* equations. Since there is only one PDE, the monodomain equations are much less demanding to solve than the bidomain equations. The accuracy penalty is acceptable in the absence of applied currents (Potse et al., 2006), but if external shocks (e.g. to investigate defibrillation, see for example Rodriguez et al. 2005, 2006) or the magnetic field (e.g. dos Santos et al., 2002) are to be modelled then the bidomain model is the only choice.

A variety of numerical schemes may be used to calculate a numerical solution of the monodomain or bidomain equations (see, for some examples and further citations, Sundnes et al. 2001; Smith et al. 2004; Whiteley 2006; Vigmond et al. 2008). As the heart has an irregular geometry, the equations with spatial derivatives in both the monodomain and bidomain equations are usually solved using the finite element method (Reddy 1993; also described in brief in Chapter 6). This discretises the problem domain into a 'mesh' of elements, and approximates the true solution by a function (e.g. a low-order polynomial) on each element. The PDEs at each time step may then be reduced to a large, sparse, linear system of equations which is solved to obtain values of the functions at nodes of the mesh. This approach also handles the derivative boundary conditions systematically.

An action potential usually propagates through cardiac tissue with a very steep wavefront. In order to resolve the wavefront accurately, a fine mesh must thus be used, typically with a resolution of 100–200 μm. Solving the large linear systems that result is thus very computationally demanding (Vigmond et al., 2008). Furthermore, the cell models increasingly include a large number of physiological processes that vary on a wide range of timescales: to accurately resolve the effect of the fast physiological processes, a short time-step must be used, commonly on the order of $10^{-2}\,\mathrm{ms}$. Using the computing power available in 2003, Hunter et al. (2003) reported that about thirty million grid points and fifty thousand time-steps would be required to compute a single heartbeat ($0.5\,\mathrm{s}$) in a realistic anatomical model ($250\,\mathrm{cm}^3$) of the heart, necessitating several days to compute even on the most powerful of contemporary high-performance computers. More recently, Potse et al. (2006) state that it takes 2 days to simulate a single human heartbeat using the bidomain model on a 32-processor machine (albeit at a fairly coarse spatial scale). It should also be noted that these simulations only consider the electrical activity. If one also considers modelling the mechanical contraction of the heart, fluid flow of blood through the circulation, and the coupling between these systems, the computational requirements increase significantly (see the review by Kerckhoffs et al. 2006, for example).

In such a context, the importance of optimisation to increase simulation speed is clearly seen. As well as the use of advanced numerical algorithms (for an example, and other citations, see Whiteley 2006), traditional approaches from computer science have their part to play. Whichever mathematical model and numerical algorithm are chosen for performing tissue simulations, it is still necessary to solve the ODE system representing the cell model at each point in space, at each time step. Solving these millions of ODE systems accounts for a not insignificant portion of the total computational effort (with the exact proportion depending on the particular algorithms and cell models used). In the present work we have thus chosen to focus on optimising this area of the simulation.

2.2 Cardiac cell models

As we have indicated, our contribution in this thesis is focused on the optimisation of single cell models. There have been several review papers giving a history of cardiac modelling (for example, Noble and Rudy 2001; Noble 2002; Hunter et al. 2003; Noble 2004; Rudy and Silva 2006), and we do not seek to reproduce them here. Rather, through the use of a selection of both seminal and recent models we aim to present the main features of cardiac models, and provide a representative sample of the field which may be used to test our optimisation techniques.

Our selection is limited to twelve models, in order to avoid excessive clutter in the results graphs. We also restrict our attention to models where we have access to the model in a form to which we can apply our optimisation techniques; this rules out very few models however.[2] The selection presented here consists of five seminal and seven recent models, covering five species of animal (guinea–pig, rabbit, mouse, dog, and human) and the major types of cardiac myocyte (SAN, atrium, Purkinje fibre, and ventricle). They also span a range of 'model genealogies'—while they do share certain features, since they are all models of cardiac myocytes, they have not all been developed by successive modification of the same original model.

2.2.1 Hodgkin–Huxley 1952 (Hodgkin and Huxley, 1952)

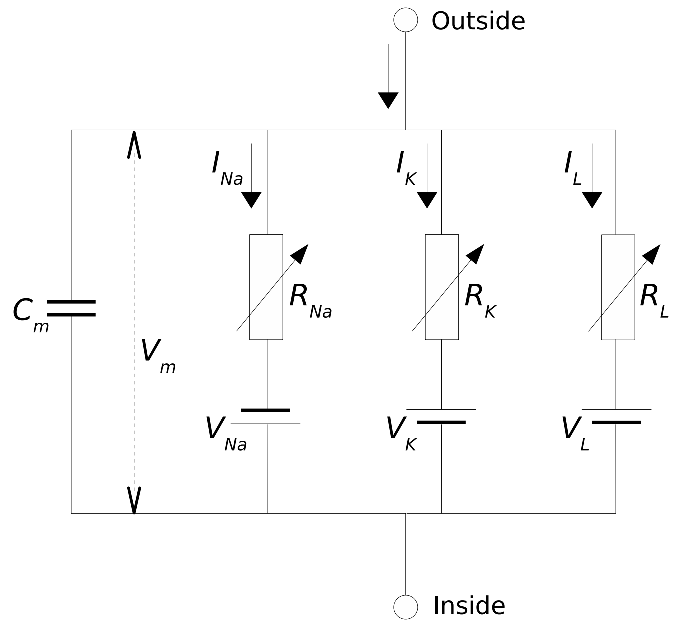
The paper "A quantitative description of membrane current and its application to conduction and excitation in nerve" (Hodgkin and Huxley, 1952) concluded a series of papers concerned with the flow of electric current through the surface membrane of a giant nerve fibre, by presenting a mathematical model capable of reproducing the experimental results seen. While Hodgkin and Huxley were concerned with nerve signal propagation in squid, their insights and the general structure of their mathematical formulation have underpinned cardiac cellular modelling for the past four decades.

This was the first model to use mathematical reconstruction of ion channel behaviour, rather than abstract equations, thus linking modelling directly to the underlying physiological processes. It was also extremely successful, correctly predicting the action potential shape, the impedance changes, and the conduction velocity.

The electrical circuit used to represent a nerve cell is shown in Figure 2.2. The total current

[2]CellML files encoding the models in our sample may be downloaded from `https://chaste.ediamond.ox.ac.uk/cellml/sample.zip`.

Figure 2.2 Electrical circuit representing a squid giant axon membrane (Hodgkin and Huxley, 1952). Resistors represent ion channels. The lipid bilayer that forms the cell membrane acts as a capacitor with capacitance C_m. V_m is the transmembrane potential.



is divided into capacitative and ionic contributions: the cell membrane acts as a capacitor, as a result of its hydrophobic nature that makes it impermeable to charged ions, and charge is also carried by a variety of ions moving down their respective electrochemical gradients, through ‘channels’ in the membrane. The change in V_m can then be described by the ODE

$$\frac{\mathrm{d}V_m}{\mathrm{d}t} = -\frac{1}{C_m} I_{ion},$$

where I_{ion} is the total transmembrane ionic current and C_m is the membrane capacitance. To normalise for variability in cell size, models typically consider capacitance per unit area of membrane, and current densities, so C_m might be given in $\mu\mathrm{F/cm}^2$ and I_{ion} in $\mu\mathrm{A/cm}^2$.

The model considers I_{ion} to be composed of three currents—those caused by sodium and potassium ions (I_{Na} and I_K), and a small ‘leakage current’ due to other ions, such as chloride. Each species of ion has its own conductance and equilibrium potential, and the current is calculated using Ohm's law, for example

$$I_{Na} = g_{Na}(V_m - E_{Na})$$

where g_{Na} is the sodium channel conductance $[\mathrm{mS/cm^2}]$ and E_{Na} is the equilibrium potential $[\mathrm{mV}]$, giving I_{Na} in $\mu\mathrm{A/cm^2}$.

Hodgkin and Huxley posited the existence of activating and inactivating molecules for the ion channels, which function as gates determining whether the channel is open or closed. The conductance is then a function of the maximum conductance (assumed constant) and the open probabilities of the gates (which can vary with time, depending on the transmembrane potential); for a given type of gate this probability ranges from 0 (all gates closed) to 1 (all open), and can be described by an ODE of the form

$$\frac{\mathrm{d}m}{\mathrm{d}t} = \alpha(1 - m) - \beta m,$$

where m is the open probability (and hence $1 - m$ is the closed probability), and α and β are V_m-dependent opening and closing transition rates, described by exponential expressions. Gate transitions are assumed to be independent of each other. The sodium current was found to be accurately modelled by three identical activation gates (each with open probability m) and one inactivation gate (with open probability h), giving the conductance as

$$g_{Na} = \bar{g}_{Na} m^3 h,$$

where $\bar{g}_{Na}$ is the maximum conductance.

The squid giant axon is an unusually large nerve cell, with a diameter typically around $0.5\,\mathrm{mm}$. It is this feature that was key to Hodgkin and Huxley's work—they were able to insert electrodes inside the axon, and thus obtain the experimental data needed to develop their model. This illustrates the 'middle-out' approach to modelling: start where the data is available, yet the system is simple enough to comprehend sufficiently to model. As we shall see below, subsequent advances in experimental techniques have triggered advances in modelling. Modelling has also provided insight into designing new experiments, giving an iterative interplay between models and experiments (Noble and Rudy, 2001).

2.2.2 Noble 1962 (Noble, 1962)

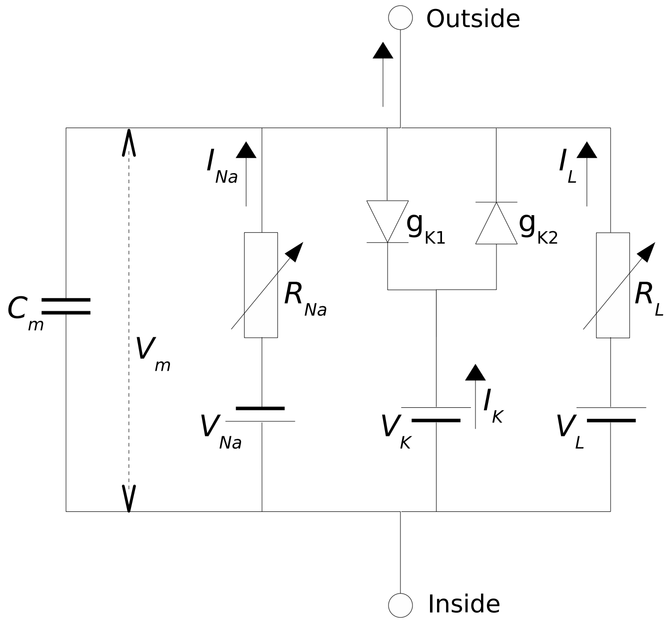
There are obvious similarities between nerve cells and myocytes: both feature an excitable membrane and produce an action potential when stimulated. The APs produced are quite different, however. A nerve AP is of extremely brief duration, on the order of 1 millisecond, whereas a human ventricular AP may last 400 milliseconds, with many intracellular events during that time controlling the mechanical contraction. In the early 1960s research was done to investigate whether, with certain modifications, Hodgkin and Huxley's formulation of the properties of excitable membranes might also be used to describe the long-lasting action and pace-maker potentials of the Purkinje fibres of the heart. Noble was the first to show that a model incorporating *two* potassium currents could produce a long AP duration without requiring unrealistically high conductances, thus reducing the energy cost by an order of magnitude (Noble, 2002, 2004).

Various conventions of cardiac models differ from those used by Hodgkin and Huxley. The primary change is in the sign of V_m: in myocytes V_m is the potential of the inside with respect to the outside of the membrane (i.e. $V_m = \phi_i - \phi_e$). This means that an AP is a positive variation in the transmembrane potential, and the resting potential is negative. Positive currents are thus flowing outward. This is illustrated in the circuit diagram for this model, Figure 2.3.

2.2.3 DiFrancesco–Noble 1985 (DiFrancesco and Noble, 1985)

In 1964 the voltage clamp experimental technique was successfully applied to cardiac muscle for the first time (Deck and Trautwein, 1964). The existence of a current carried by calcium ions was soon discovered. Along with other discoveries, this formed the basis of the MNT (McAllister et al., 1975) and Beeler–Reuter (Beeler and Reuter, 1977) models. The latter, due to its simplicity as compared with later models of the ventricular action potential, has been exten-

Figure 2.3 Electrical circuit representing the 1962 Noble model (Noble, 1962). Resistors represent ion channels. The potassium current is assumed to flow through two non-linear resistances which have preferential directions for current flow. The lipid bilayer that forms the cell membrane acts as a capacitor with capacitance C_m. V_m is the transmembrane potential.



sively used in multicellular simulations.

We have chosen to focus instead on the later DiFrancesco–Noble model of Purkinje fibres, which was the first to incorporate ionic concentration changes within the cell. This was a fundamental advance, since these are essential to the study of some disease states (Noble and Rudy, 2001). In modelling such changes, it was also necessary to model the processes by which the ionic gradients can be restored and maintained, and thus in a modelling 'avalanche' several linked ion exchangers and pumps had to be included. As the names suggests, these allow respectively for the exchange of ions (e.g. sodium for calcium) and for metabolic control of certain currents. The model was also the first to consider events occurring inside the cell, rather than just at the cell membrane, by including a model of calcium release from the sarcoplasmic reticulum.

While not included in the DiFrancesco–Noble model, modelling of changes in calcium concentration has been crucial in later models that consider electro-mechanical coupling (Kerckhoffs et al., 2006), since it is the release of calcium into the cytoplasm of the cell that triggers contraction. The sarcoplasmic reticulum is a structure within the cell that stores large quantities of calcium, which is taken up from the cytoplasm while the cell relaxes. When an action potential is initiated, the transfer of a small amount of calcium from the extracellular space into the cell triggers the rapid release of calcium from the sarcoplasmic reticulum (this is known as calcium-induced calcium release) which causes mechanical contraction.

2.2.4 Noble–Noble 1984 (Noble and Noble, 1984)

Noble and Noble also developed a variation of the DiFrancesco–Noble model to apply to the mammalian sino-atrial node, primarily by altering various parameter values to match experimental data. One of the differences between the pacemaker activity of Purkinje fibres and the SAN is that the latter is far less sensitive to extracellular potassium, and so in this sense the rhythm generated is more robust. Again this is a seminal model, being the first model of the SAN to incorporate intracellular ion concentration changes.

It also attempted to account for differences in behaviour between cells in different regions of the SAN: peripheral regions of the SAN show a higher resting potential than the centre. Various parameter modifications were able to reproduce these differences.

2.2.5 Luo–Rudy 1991 (Luo and Rudy, 1991)

With the development of single-cell and single-channel recording techniques in the 1980s, the limitations of voltage-clamp measurements were overcome and the intracellular and extracellular ionic environments could be controlled. The additional data thus available were used by Luo and Rudy to develop a quantitative model of the mammalian ventricular action potential. Their 1991 paper marks the first step, formulating the fast inward sodium current and four outward potassium currents, and incorporating the possibility of changing the extracellular potassium concentration.

Many subsequent models have been based upon the Luo–Rudy formulations, and in particular upon the second phase of development (Luo and Rudy, 1994), which incorporated the dynamics of intracellular ionic concentrations, and is thus known as the LRd model. It includes most of the ion exchangers and pumps, as well as implementing cell compartmentalization (myoplasm, junctional and nonjunctional sarcoplasmic reticulum), calcium buffers in the myoplasm (troponin, calmodulin) and in the junctional sarcoplasmic reticulum (calsequestrin), and calcium-induced calcium release.

2.2.6 Courtemanche *et al.* 1998 (Courtemanche et al., 1998)

There are many interspecies differences in myocyte behaviour. For example, the transient outward current I_{to} present in rabbit and human atrial cells has been shown to recover from inactivation at least two orders of magnitude faster in humans than in rabbits (Courtemanche et al., 1998). Thus, while earlier models of atrial cells based solely on animal data have provided valuable insights into the mechanisms underlying AP generation in those species, the interspecies differences and the amount of human data available led Courtemanche *et al.* to develop a model based specifically on direct measurements of human atrial currents, albeit supplemented with animal data where human data were lacking. Many of the current formulations are based on the LRd model (Luo and Rudy, 1994).

The AP produced by the model resembles those recorded from human atrial samples, and responds to various interventions in a manner consistent with experimental data. The model also has some clinical relevance, as it provides insights into the basic mechanisms underlying a variety of clinical determinants of atrial fibrillation.

The model of Nygren et al. (1998) was developed independently during the same period, and also models a human atrial myocyte. We have arbitrarily elected to use the Courtemanche *et al.* model instead.

2.2.7 Noble *et al.* 1998 (Noble et al., 1998)

Modelling calcium handling is crucial when considering whole-heart behaviour, rather than focusing purely on electrical activity. Many cardiac disease states are not purely electrical (even if electrical mayhem is often the final villain) but incorporate biochemical and mechanical factors. Mechanical contraction is triggered by the release of calcium within the cell, and hence by the electrical activity, but this causality is not one way—various ion channels are sensitive to cell shape changes.

Detailed models of calcium handling are, however, computationally expensive. Indeed, the fact that one must compromise between model complexity and computability is one of the reasons for the multiplicity of cell models available. Different models are designed to answer different physiological questions. Some seek to model subcellular processes in great detail. Others use simplified models of subcellular activity in order to make tissue-level simulations tractable. The simplified dyadic space model of Noble et al. (1998), of the guinea–pig ventricle, falls into the latter category, and reproduces many of the features of the initiation of calcium signalling, including some mechanoelectric feedback, with only a modest increase in computation time.

2.2.8 Zhang *et al.* 2000 (Zhang et al., 2000)

This was the first model to consider variations in behaviour between the centre and periphery of the (rabbit) sinoatrial node based on experimental data. The model was tested by investigating the effect of blocking various ionic currents, and by measuring resting potentials after blocking spontaneous activity; in each case results compared favourably with experimental data.

An assumption made in this model illustrates an important consideration when simulating many beats. Unlike models which consider dynamic changes in intracellular ionic concentrations (e.g. LRd), all the intracellular concentrations are assumed to be constant. Since the transmembrane currents are carried by ions, however, the model is only valid for simulations of a few beats, since the law of conservation of charge is not obeyed. If ions responsible for a stimulus current are not accounted for, this can also be a problem for other models (Rudy and Silva, 2006, p. 63).

Garny et al. (2003a) note that the published equations do not reproduce the figures shown in the Zhang *et al.* paper, an instance demonstrating the importance of standard computer-readable model description formats such as CellML. In their evaluation of 1D rabbit SAN models Garny *et al.* have thus implemented three CellML versions of the model: the published version, a version reproducing the single-cell simulations, and a version which may be used to reproduce the 1D simulation results.

This encoding of the model is particularly interesting for our purposes because all these versions are contained in a single CellML document. One parameter is used to select the desired version, and another parameter sets the cell position within the SAN. Decision making based on these parameters is nested within the model equations, thus there is considerable scope for one of our optimisation techniques to promote these decisions from run-time to compile-time, as we shall see in Chapters 5 and 7.

2.2.9 Faber–Rudy 2000 (Faber and Rudy, 2000)

This is a recent version of the LRd model (see also Section 2.2.5) and was developed to study behaviour under sodium overload within the cell, which can accompany various pathologies and lead to arrhythmia. Various features of the model have been reformulated in order to cover the range of greatly elevated intracellular sodium and calcium: the sodium–potassium pump current, the sodium–calcium exchanger current, and the process of calcium-induced calcium release from the (junctional) sarcoplasmic reticulum. A sodium-activated potassium current is also incorporated to investigate the hypothesis that this plays a significant role in APD shortening during conditions of elevated intracellular sodium.

We have included this model as it is the most recent guinea–pig LRd-based model available at the time of writing, and it is in common usage.

2.2.10 Fox *et al.* 2002 (Fox et al., 2002)

The period from the end of one action potential to the start of the next is known as the diastolic interval, during which the ventricles relax. The length of the diastolic interval has a large effect on the duration of the subsequent action potential, and the relationship between the two is known as the action potential duration restitution relation. If the slope of the restitution relation is at least 1, then high-frequency pacing commonly leads to an alternation of action potential duration, or electrical alternans, with ‘normal’ APs interleaved with shortened APs.

Such alternans may be a precursor to the development of ventricular arrhythmias and fibrillation, and so Fox et al. (2002) aimed to develop a model which would exhibit stable alternans, and hence to identify the ionic currents responsible for their occurrence. These currents were then manipulated to eliminate alternans. This illustrates one important use for quantitative modelling—gaining insight into the underlying causes of medical conditions in order to determine novel treatments.

Their model was based primarily on the earlier Winslow model of the canine ventricle (Winslow et al. 1999; itself based on the LRd model), with some parts also taken from the LRd and Chudin et al. (1999) models, altered as necessary to fit experimental voltage-clamp data from canine ventricular myocytes. As in Section 2.2.6, this illustrates the trend towards species-specific models based on species-specific experimental data, in contrast to earlier generic ‘mammalian’ models.

2.2.11 Bondarenko *et al.* 2004 (Bondarenko et al., 2004)

In recent years a large body of knowledge has accumulated on the molecular structure of cardiac ion channels, their function, and their modification by genetic mutations associated with various cardiac disorders. Incorporating this knowledge into cell-level models, in order to understand how molecular-level changes affect cellular function, remains a major challenge. In a recent review article, Rudy and Silva (2006) explain how *Markov* models of ion channel function can be included within integrated models of cardiac cells.

Markov models explicitly represent ion channel states, and assign probabilities for the transitions between states. It is assumed that transitions depend only on the present state of the channel, not on previous behaviour. Differential equations can then be used to compute the probability of an ion channel occupying any given state, or equivalently the proportion of ion channels of that type in each state, at a given point in time. Note that unlike ‘true’ Markov models, which consider small populations and are therefore stochastic, these differential equation models are deterministic, and hence rely on the assumption that there are many ion channels of each type in a cell. With these models, channels may have a variety of open, closed, and inactivated states. As well as being able to express the situations described by a Hodgkin–Huxley formulation of channel activity, Markov models can also account for cases where activation and inactivation of a channel are not independent, but coupled. The state transitions of a Markov model typically represent specific channel movements that have been characterised experimentally. Many genetic manipulations and diseases alter the kinetic properties of a single ion channel and can be related to changes in specific transition probabilities in the Markov models for these channels.

Bondarenko *et al.* developed a mouse ventricular cell model from voltage clamp data, using Markov models to represent ion channels where possible. The attribution of putative molecular bases for several of the component currents enables this model to be used to simulate the behaviour of genetically modified transgenic mice (which have genetic defects associated with human diseases). The model also has detailed intracellular calcium dynamics, including spatial localisation.

Due to the high level of detail in this model, it is extremely computationally intensive. Also, it is the only model in our sample to use Markov models extensively, rather than using Hodgkin–Huxley-style formulations for the ion channels. It thus provides an interesting test case for our optimisation techniques.

2.2.12 ten Tusscher and Panfilov 2006 (ten Tusscher and Panfilov, 2006)

The ten Tusscher models (ten Tusscher et al., 2004; ten Tusscher and Panfilov, 2006) are currently the most advanced human ventricular models, designed with multicellular simulations in mind. They are thus not as detailed as some models in modelling certain features, particularly of the calcium dynamics.

The 2006 model is based on more recent experimental measurements of human AP duration restitution (see also Section 2.2.10), and includes a more extensive description of intracellular calcium dynamics than the earlier model, by considering localisation of calcium within the cell and using a reduced Markov model to represent calcium-induced calcium release (see also Section 2.2.3). These features are used to investigate the causes of ventricular fibrillation, a major cause of death, by simulating blocks of cardiac tissue or a whole ventricle, and examining how the chaotic electrical patterns characteristic of fibrillation can be induced. It is hoped that this will lead to an understanding of how they can be prevented.

The model also comes in three variants, describing differences between cells at different depths within the ventricular wall: endocardial cells on the interior layer, midicardial (M) cells in the central region, and epicardial cells forming the outer layer. Our simulations in Chapter 7 will use the midicardial variant.

2.2.13 Cell model summary

In the previous sections some of the key features of cardiac cell models have been introduced: ion transfer across the cell membrane (either by Hodgkin–Huxley style formulations or Markov models), ion concentration changes within the cell, and intracellular calcium handling (including the link with mechanical contraction). Also, we have seen the ongoing trend towards models designed for specific uses, aimed at detailed predictions of behaviour in a single species for particular disease states, and hence based increasingly on data solely from the relevant species.

Two important points for the current work can be drawn from this. Firstly, as more and more data becomes available, models become increasingly detailed and complex, and the need

Table 2.1 Number of ODEs and ionic currents included in each model of our sample.

Model	**ODEs**	**Currents**
Hodgkin–Huxley 1952	4	3
Noble 1962	4	3
DiFrancesco–Noble 1985	16	10
Noble–Noble 1984	15	9
Luo–Rudy 1991	8	6
Courtemanche *et al.* 1998	21	12
Noble *et al.* 1998	26	20
Zhang *et al.* 2000	15	15
Faber–Rudy 2000	25	16
Fox *et al.* 2002	13	13
Bondarenko *et al.* 2004	41	15
ten Tusscher and Panfilov 2006	19	12

for effective optimisation techniques increases. This is illustrated in Table 2.1. Secondly, as mentioned in Section 2.2.7, there is no single 'best model'; rather it is necessary to choose the most appropriate model for the scientific questions asked. Faced with a multiplicity of models, the use of languages such as CellML (see the next section) for describing them in a portable fashion is essential. In this context, the work of this thesis is most timely.

2.3 The CellML modelling language

Implementing a published cell model is rarely a straightforward exercise. Simple typographical errors can easily be introduced by either the author or the reader. Documentation on the model (e.g. initial conditions and units, as well as more general comments) can be lacking, making it difficult to produce working code from an abstract mathematical description. Some authors have tried to overcome this problem by publishing source code implementing their model on the internet (e.g. Luo and Rudy 1994[3]), which greatly reduces errors, but does not prevent them when porting a model to another simulation environment (that is, a piece of software or a software framework for simulating such models). Since newer environments are likely to be more advanced, it is desirable to be able to port models easily. Also, if each cell model is

[3]`http://rudylab.wustl.edu/research/cell/methodology/`

written in an ad-hoc coding format, it is then difficult to integrate these in a coherent manner into higher-level simulations (e.g. of the whole heart).

The modelling language CellML (Hedley et al., 2001; Cuellar et al., 2003; Lloyd et al., 2004) was developed by the Bioengineering Institute at the University of Auckland to facilitate the exchange of biological cell-level models, ameliorating the problems described above. Its focus is on enabling those developing mathematical models of cellular phenomena to write their models easily in an abstract, well-defined form. The basic constituents and structure of CellML are simple, providing a common basis for describing models, and facilitating the creation of complex models from simpler ones by combining models and/or adding detail to existing models. It is an XML-based language, and hence is well suited to enable easy manipulation of models by computer programs, for example generating graphical or LaTeX representations of models, or integrating models into a simulation environment. Various such environments have support for CellML models, including PCEnv[4], COR[5] (Garny et al., 2003b), CESE[6] (Missan and McDonald, 2005), Virtual Cell[7] (Loew and Schaff, 2001), JSim[8], and CMISS[9]. A review of tool support for CellML has recently been written by Garny et al. (2008).

CellML provides us with a clear, abstract format for describing cardiac ionic models. A large component of this D.Phil. has been to automatically transform these model descriptions to produce efficient implementations of the models, and the XML nature of CellML expedites this process. There are, however, other biological modelling languages which could be used to represent such models, and provide a basis for our transformations, notably SBML (Hucka et al., 2004). The primary reason for favouring CellML for this work is the availability of a repository of CellML models[10] which includes many curated cardiac ionic cell models. With CellML we thus have access to a large test base for our techniques. SBML also has a narrower scope than

[4]`http://www.cellml.org/tools/pcenv/`
[5]`http://cor.physiol.ox.ac.uk/`
[6]`http://cese.sourceforge.net/`
[7]`http://www.nrcam.uchc.edu/`
[8]`http://nsr.bioeng.washington.edu/PLN/`
[9]`http://www.cmiss.org/`
[10]`http://www.cellml.org/models`

CellML, being focused on describing sub-cellular processes such as reaction networks. It is thus less suited to the ODE system models of cardiac cells which we address.

2.3.1 CellML syntax

As an XML dialect, CellML is a very verbose language. To illustrate this, consider the exceedingly simplified model

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{I}{C}, \qquad I = gV, \qquad C, g \text{ constants.} \tag{2.3.1}$$

This could be encoded by the CellML document shown in Listing 2.1. Within this thesis we therefore give examples in a compact syntax based on the ‘readable format’ used by COR (Garny et al., 2003b). In this compact syntax, the above model would be represented as shown in Listing 2.2.

Listing 2.1: A possible CellML encoding of the simple model given in (2.3.1).

```
<model name='example' xmlns='http://www.cellml.org/cellml/1.0#'>
  <units name='ms'>
    <unit prefix='milli' units='second'/>
  </units>
  <component name='A'>
    <variable name='time'    units='ms'     public_interface='out'/>
    <variable name='voltage' units='volt'   public_interface='out'
                                            private_interface='out'/>
    <variable name='current' units='ampere' private_interface='in'/>
    <variable name='C'       units='farad'  initial_value='1'/>
    <math xmlns='http://www.w3.org/1998/Math/MathML'>
      <apply><eq/>
        <apply><diff/>
          <bvar><ci>time</ci></bvar>
          <ci>voltage</ci>
        </apply>
        <apply><divide/>
          <ci>current</ci>
          <ci>C</ci>
        </apply>
      </apply>
    </math>
  </component>
  <component name='B'>
    <variable name='V' units='volt'     public_interface='in'/>
    <variable name='I' units='ampere'   public_interface='out'/>
    <variable name='g' units='siemens'  initial_value='0.3'/>
    <math xmlns='http://www.w3.org/1998/Math/MathML'>
```

```
          <apply><eq/>
            <ci>I</ci>
            <apply><times/>
              <ci>g</ci>
              <ci>V</ci>
            </apply>
          </apply>
        </math>
      </component>
      <group>
        <relationship_ref relationship='encapsulation'/>
        <component_ref component='A'>
          <component_ref component='B'/>
        </component_ref>
      </group>
      <connection>
        <map_components component_1='A' component_2='B'/>
        <map_variables variable_1='voltage' variable_2='V'/>
        <map_variables variable_1='current' variable_2='I'/>
      </connection>
    </model>
```

Listing 2.2: The model of Listing 2.1 described using a compact syntax.

```
def model example as
    def unit ms from
        unit second {pref: milli};
    def comp A as
        var time: ms {pub: out};
        var voltage: volt {priv: out, pub: out};
        var current: ampere {priv: in};
        var C: farad {init: 1};
        ode(voltage, time) = current / C;
    def comp B as
        var V: volt {pub: in};
        var I: ampere {pub: out};
        var g: siemens {init: 0.3};
        I = g * V;
    def group as encapsulation for
        comp A incl
            comp B;;
    def map between A and B for
        vars voltage and V;
        vars current and I;
```

The compact syntax can be (approximately) described using an EBNF[11] notation. This consists of rules defining terms (representing parts of the language) by expressions, which have some similarity to regular expressions. They may contain literals of the compact syntax (in

[11]Extended Backus–Naur Form; see Wirth (1977).

bold type) or references to terms (in italics). Certain characters have special meaning: text in square brackets is optional; a vertical bar indicates a choice between options; a superscript asterisk denotes that the preceding item may occur any number of times (including not at all); whereas a superscript plus symbol means the item must occur at least once. Round brackets are used for grouping. In the EBNF below, the term $\mathcal{M}$ represents a CellML model.

$$
\begin{array}{rcl}
\mathcal{M} & = & \textbf{def model } \textit{name } \textbf{as } \mathcal{U}^* \; \mathcal{C}^* \; \mathcal{G}^* \; \mathcal{K}^* \\
\mathcal{U} & = & \textbf{def unit } \textit{uname } [\textbf{from} \\
 & & (\textbf{unit } \textit{uname } \{ \; \textit{urefs} \; \};)^+] \\
\textit{urefs} & = & \textit{uref} \mid \textit{uref} \;,\; \textit{urefs} \\
\textit{uref} & = & m \mid p \mid e \mid o \\
m & = & \textbf{mult}\text{: } \textit{real} \\
p & = & \textbf{pref}\text{: } \textit{prefix} \\
e & = & \textbf{expo}\text{: } \textit{real} \\
o & = & \textbf{offset}\text{ : } \textit{real} \\
\mathcal{C} & = & \textbf{def comp } \textit{cname } \textbf{as } \mathcal{V}^+ \; \mathcal{E}^* \\
\mathcal{V} & = & \textbf{var } \textit{vname} \text{ : } \textit{uname} \; [\{ \; \textit{init}, \textit{interface} \; \}] \; ; \\
\mathcal{E} & = & (\textit{vname} \mid \textbf{ode}(\; \textit{vname} \;,\; \textit{vname} \;)) \\
 & & = \textit{mathematics} \; ; \\
\mathcal{G} & = & \textbf{def group as } \textit{type } \textbf{for } \mathcal{T}^+ \\
\textit{type} & = & \textit{containment} \mid \textit{encapsulation} \\
\mathcal{T} & = & \textbf{comp } \textit{cname } [\textbf{incl } \mathcal{T}^+]; \\
\mathcal{K} & = & \textbf{def map between } \textit{cname } \textbf{and } \textit{cname } \textbf{for} \\
 & & \mu^+ \\
\mu & = & \textbf{vars } \textit{vname } \textbf{and } \textit{vname} \; ; \\
\textit{uname} & = & \textit{vname} = \textit{cname} = \textit{name}
\end{array}
$$

In explaining the usage of the various constructs available in CellML, we illustrate them using a modified version of the Hodgkin–Huxley equations (see Section 2.2.1), shown in Listing 2.3. The only alterations to the published model have been to change the conventions used to match those in cardiac models (see Section 2.2.2).

A CellML model $\mathcal{M}$ is represented as a collection of discrete components $\mathcal{C}$ linked by connections $\mathcal{K}$ to form a network. A component is a functional unit that may correspond to a physical compartment (e.g. the *membrane* component represents the cell membrane), a collection of entities engaged in similar tasks (e.g. the *sodium_channel* component, which represents all the sodium channels in the cell membrane), or a convenient modelling abstraction (e.g. the

Listing 2.3: The modified Hodgkin–Huxley equations encoded in CellML, taken from `http://www.cellml.org/models/hodgkin_huxley_1952_version07`.

```
def model hodgkin_huxley_squid_axon_1952 as
    def unit millisecond from
        unit second {pref: milli};
    def unit per_millisecond from
        unit second {pref: milli, expo: -1};
    def unit millivolt from
        unit volt {pref: milli};
    def unit milliS_per_cm2 from
        unit siemens {pref: milli};
        unit metre   {pref: centi, expo: -2};
    def unit microF_per_cm2 from
        unit farad {pref: micro};
        unit metre {pref: centi, expo: -2};
    def unit microA_per_cm2 from
        unit ampere {pref: micro};
        unit metre  {pref: centi, expo: -2};

    def comp environment as
        var time: millisecond {pub: out};

    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_Na:   microA_per_cm2 {pub: in};
        var i_K:    microA_per_cm2 {pub: in};
        var i_L:    microA_per_cm2 {pub: in};
        var i_Stim: microA_per_cm2 {pub: in};

        ode(V, time) = -(-i_Stim+i_Na+i_K+i_L)/Cm;

    def comp sodium_channel as
        var i_Na: microA_per_cm2 {pub: out};
        var g_Na: milliS_per_cm2 {init: 120};
        var E_Na: millivolt;
        var time: millisecond {pub: in, priv: out};
        var V:    millivolt   {pub: in, priv: out};
        var E_R:  millivolt   {pub: in};
        var m:    dimensionless {priv: in};
        var h:    dimensionless {priv: in};

        E_Na = E_R+115{millivolt};
        i_Na = g_Na*pow(m, 3{dimensionless})*h*(V-E_Na);

    def comp sodium_channel_m_gate as
        var m:       dimensionless {init: 0.05, pub: out};
        var alpha_m: per_millisecond;
        var beta_m:  per_millisecond;
        var V:       millivolt     {pub: in};
    var time:       millisecond {pub: in};

    alpha_m = −0.1{per_millisecond}*(V+50{millivolt})/
              (exp(−(V+50{millivolt})/10{dimensionless})
                −1{dimensionless});
    beta_m = 4{per_millisecond}*exp(−(V+75{millivolt})/
                18{millivolt});
    ode(m, time) = alpha_m*(1{dimensionless}−m)−beta_m*m;

def comp sodium_channel_h_gate as
    var h:          dimensionless {init: 0.6, pub: out};
    var alpha_h:    per_millisecond;
    var beta_h:     per_millisecond;
    var V:          millivolt   {pub: in};
    var time:       millisecond {pub: in};

    alpha_h = 0.07{per_millisecond}*exp(−(V+75{millivolt})/
                 20{millivolt});
    beta_h = 1{per_millisecond}/
             (exp(−(V+45{millivolt})/10{dimensionless})
               +1{dimensionless});
    ode(h, time) = alpha_h*(1{dimensionless}−h)−beta_h*h;

def comp potassium_channel as
    var i_K:    microA_per_cm2 {pub: out};
    var g_K:    milliS_per_cm2 {init: 36};
    var E_K:    millivolt;
    var time:   millisecond    {pub: in, priv: out};
    var V:      millivolt      {pub: in, priv: out};
    var E_R:    millivolt      {pub: in};
    var n:      dimensionless  {priv: in};

    E_K = E_R−12{millivolt};
    i_K = g_K*pow(n, 4{dimensionless})*(V−E_K);

def comp potassium_channel_n_gate as
    var n:          dimensionless {init: 0.325, pub: out};
    var alpha_n:    per_millisecond;
    var beta_n:     per_millisecond;
    var V:          millivolt   {pub: in};
    var time:       millisecond {pub: in};

    alpha_n = −0.01{per_millisecond}*(V+65{millivolt})/
              (exp(−(V+65{millivolt})/10{dimensionless})
                −1{dimensionless});
    beta_n = 0.125{per_millisecond}*exp((V+75{millivolt})/
                80{millivolt});
    ode(n, time) = alpha_n*(1{dimensionless}−n)−beta_n*n;

def comp leakage_current as
    var i_L:    microA_per_cm2 {pub: out};
    var g_L:    milliS_per_cm2 {init: 0.3};
    var E_L:    millivolt;
    var time:  millisecond {pub: in};
    var V:     millivolt   {pub: in};
    var E_R:   millivolt   {pub: in};

    E_L = E_R+10.613{millivolt};
    i_L = g_L*(V-E_L);

def comp stimulus_protocol as
    var Istim:               microA_per_cm2 {pub: out};
    var IstimStart:          millisecond    {init: 50};
    var IstimEnd:            millisecond    {init: 50000};
    var IstimAmplitude:      microA_per_cm2 {init: 20};
    var IstimPeriod:         millisecond    {init: 200};
    var IstimPulseDuration:  millisecond    {init: 0.5};
    var time:                millisecond    {pub: in};

    Istim = sel
        case (time ≥ IstimStart) and (time ≤ IstimEnd) and
             (time - IstimStart -
                 floor((time-IstimStart)/IstimPeriod)*IstimPeriod
               ≤ IstimPulseDuration):
            IstimAmplitude;
        otherwise:
            0{microA_per_cm2};

def group as containment for
    comp membrane incl
        comp sodium_channel incl
            comp sodium_channel_m_gate;
            comp sodium_channel_h_gate;;
        comp potassium_channel incl
            comp potassium_channel_n_gate;;
        comp leakage_current;;

def group as encapsulation for
    comp sodium_channel incl
        comp sodium_channel_m_gate;
        comp sodium_channel_h_gate;;
    comp potassium_channel incl
        comp potassium_channel_n_gate;;

def map between membrane and stimulus_protocol for
    vars i_Stim and Istim;
def map between environment and stimulus_protocol for
    vars time and time;
def map between membrane and environment for
    vars time and time;
def map between sodium_channel and environment for
    vars time and time;
def map between potassium_channel and environment for
    vars time and time;
def map between leakage_current and environment for
    vars time and time;
```

```
def map between membrane and sodium_channel for
    vars V and V;
    vars E_R and E_R;
    vars i_Na and i_Na;
def map between membrane and potassium_channel for
    vars V and V;
    vars E_R and E_R;
    vars i_K and i_K;
def map between membrane and leakage_current for
    vars V and V;
    vars E_R and E_R;
    vars i_L and i_L;
def map between sodium_channel and sodium_channel_m_gate for
    vars m and m;
    vars time and time;
    vars V and V;
def map between sodium_channel and sodium_channel_h_gate for
    vars h and h;
    vars time and time;
    vars V and V;
def map between potassium_channel and potassium_channel_n_gate for
    vars n and n;
    vars time and time;
    vars V and V;
```

environment component which just contains the simulation time). Components thus bear some similarities to modules in conventional programming languages. They may contain variables $\mathcal{V}$, and mathematical relationships $\mathcal{E}$ that specify the interactions between those variables (in CellML these are described using MathML content markup). The *init* property can be used to specify an initial value for a state variable (which has an ODE describing its behaviour, e.g. *V* and *h* in Listing 2.3), or the value of a constant variable (e.g. *Cm*).

Variables may be local to a component, or made visible to other components via *interface* attributes. The effect of these is determined by the group structure $\mathcal{G}$. There are two types of group defined by CellML: containment and encapsulation. Containment refers to physical relationships between components; for example in Listing 2.3 the sodium channels are physically located in the cell membrane. Encapsulation is a logical relationship similar to the programming concept, and is used to hide the details of child components within a parent component, producing an *encapsulation hierarchy*. In Listing 2.3 this is used to hide internal details of the sodium channel (namely the gating variable) from the rest of the model. The behaviour implied by interfaces depends on the encapsulation hierarchy. The *private interface* of a variable says whether its value is exported to or imported from a child component (e.g. in the *sodium_channel* component the gating variable h is imported). The *public interface* specifies whether the variable's value is exported to or imported from a parent or sibling component (e.g. the *sodium_channel_h_gate* component exports the value of the gating variable h to its parent). Note that both interfaces may be specified, although some combinations of settings clearly do not make sense. Note also that a mapping between variables in different components must be specified in the connections section; giving interfaces and groups is not sufficient, although it is these that determine the direction of a mapping for each pair of mapped variables. Components without an explicit parent are treated as siblings.

There is no explicit distinction within CellML as to the role which a variable may play. For example, there is no analogy of the `const` keyword indicating a constant. Instead, the whole of the mathematics comprising the model must be analysed to determine the use to which a variable is put. All the mathematics in the models we consider consists of simple assignments as the only type of top-level expression: such expressions are applications of `eq` with a single term (either a variable or a derivative) on the left hand side, as indicated in the EBNF above. This makes classification of variables straightforward (as shown in Section 2.3.2) and simplifies the semantics presented in Chapter 3. The CellML language, however, allows more generality than is typically used by models (largely due to a lack of tool support)—expressions such as $a+b = c+d$ are permitted. In such cases more complex analysis is required to determine which variables have known values (based on other expressions) and which must be computed using the given expression. Algebraic rearrangement may be required to evaluate the expression, or several such expressions grouped together and the resulting system of equations solved.

Every variable must have physical units associated with it. This is also required for constants within a mathematics section. The CellML standard defines various base units (e.g. the SI base units, as well as `dimensionless` and `boolean`), and derived units $\mathcal{U}$ can be defined in terms of base units or other derived units (SI derived units may be assumed to be defined). We present further details of this model of units in Chapter 4.

Although not presented in our compact syntax, CellML models will typically also contain *metadata* described using the Resource Description Framework.[12] This contains information such as the model authors, a reference to the publication that describes the model, references to the experimental data used to derive or test the model, a history of changes made to the CellML document, links to ontologies specifying the biological system(s) being modelled, and much more. Further information can be found in the overview paper by Cuellar et al. (2003), and specifications for the use of metadata within models are available on the CellML website.[13]

There are currently two versions of the CellML language: 1.0 and 1.1. All of the features described so far are common to both. The major addition in CellML 1.1 is the *import* element, which allows unit, variable and component definitions in one model to be included in another, greatly increasing the reusability of CellML models. Support for CellML 1.1 models in the `cellml.org` repository is still under development.

A new version of CellML, 1.2, is currently being discussed. This will probably include several backwards-incompatible changes to the language, including the removal of reaction elements (not described above) and all group relationships apart from encapsulation—other relationships should be described using metadata. Another proposed change is to remove the explicit directionality of interfaces, so that variables are merely exposed rather than imported or exported; the direction of information flow will then have to be determined by analysis of the mathematics.

2.3.2 Variable classification

As mentioned in Section 2.3.1, CellML does not explicitly declare the roles played by variables; instead this information is implicit in the mathematical relationships defined between variables. Tools must thus analyse the model to determine the use to which a variable is put. Similarly to COR, we identify the following classes of variables:

[12]RDF; see `http://www.w3.org/RDF/`
[13]`http://www.cellml.org/specifications`

free a variable with respect to which others are differentiated, e.g. time;

state a variable that is differentiated with respect to a free variable, e.g. the transmembrane potential;

constant a variable that has a constant value set by its initial condition, not by any equation;

computed a variable whose value is computed in a direct fashion (i.e. not via an ODE);

mapped a variable whose value is imported from another component (such variables could equally well be thought of as having the same classification as their source variable, but the distinction is useful from a programming point of view); and

unknown a variable that cannot be put in another category, for whatever reason.

Where models contain only explicit assignment expressions, a simple algorithm may be used to classify variables. The one presented here is essentially the same as that used by COR, and proceeds as follows.

1. Classify all variables as *unknown*.
2. Process the encapsulation hierarchy and connection elements to determine which variables should be classified as *mapped*.
3. Classify all variables with an initial value set as '*maybe constant*'. Note that in a valid model (according to the CellML specification) a variable cannot be both mapped and 'maybe constant'.
4. Now do the following for each assignment expression in the model.
   (a) If the expression represents an ODE, i.e. the left hand side is a derivative, classify the independent variable as *free* and the dependent variable as *state*.

Again we can apply some validity checks. If the state variable doesn't have an initial value, or was previously classified as computed, then the model is invalid. It is also invalid if the free variable was not classified as either free or unknown.

(b) Otherwise, classify the variable assigned to on the left hand side as *computed*. If it was not classified as unknown then the model is invalid.

(c) Recursively process the expression tree forming the right hand side of the assignment, classifying as *constant* any variables which appear and were classified as 'maybe constant'.

We have now described the background to CellML, and the features of the language. Thus far, discussion of the *meaning* of language constructs has been informal. In the next chapter, we define formally what it means to evaluate a CellML model.

3

An Operational Semantics for CellML

In this chapter we define an operational semantics for CellML, which provides us with a formal definition of the language, suitable for reasoning about mathematically. The semantics is given in terms of an abstract data type describing a CellML model (Section 3.1) and an interpreter for models described in this fashion (Section 3.2). Reasons for defining the semantics in this way will be given.

This chapter describes a dynamic *semantics for the meaning of a CellML model when evaluated. The next chapter addresses a static semantics, providing compile-time checks of model validity.*

It must be emphasised that CellML does not specify *how* a model should be simulated (e.g. which ODE solver to use); rather it describes the structure and mathematics of the model, and leaves the choice of solver to the user. On the mathematics side, it thus specifies a vector $\boldsymbol{Y}$ of state variables and the ODE system

$$\frac{\mathrm{d}\boldsymbol{Y}}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{Y}, t). \tag{3.0.1}$$

A solver for such a system basically consists of a loop over time t. The simplest version is Euler's method, which at each time-step t, given a current solution $\boldsymbol{Y}_t$, computes the next solution by

$$\boldsymbol{Y}_{t+\Delta t} = \boldsymbol{Y}_t + \boldsymbol{f}(\boldsymbol{Y}_t, t)\Delta t. \tag{3.0.2}$$

More complex solvers will perform additional calculations during each iteration, for example calculating the right hand side of the system at multiple points during the next time interval, or making use of solutions from several previous iterations. Süli and Mayers (2003) have written a good introduction to this topic.

In order to simulate a CellML model, it must thus be imported into a simulation environment which provides the appropriate numerical algorithms. Given the existence of multiple simulation environments, it is vital that they agree on the *meaning* of a CellML model if exchanging models in this format is to be effective. We thus need a clear semantics of CellML. This has been provided by the specifications (Hedley and Nelson, 2001; Cuellar et al., 2006) available on the `cellml.org` website. These, as with other XML specifications, are lengthy documents written in semi-formal English, describing the structure of CellML documents and the intended behaviour of CellML processing software, and are very thorough. However, as we discuss below, there are limitations to this approach, which we have sought to address by defining a formal semantics for CellML.

With any optimisations we use, it is crucial to be confident that they are correct, in order to have confidence that the results of our simulations are valid derivations from our models. If simulation results differ from empirical data, it is important to know whether the mathematical model or the simulation code is at fault. As simulation develops to the point where it is of direct clinical relevance, reliability of the results will be even more important. Hence we need to be able to show that any transformations we apply to models do not change the results of simulations of the models in any significant way. This also requires a good definition of the meaning of a model, in order to prove that this is invariant under our transformations. In other words, we need a semantics for our model description language that is mathematically tractable.

Since the meaning of a CellML model should be independent of whatever method might be used to simulate the model, we cannot define this meaning in terms of simulation results. This makes our CellML semantics somewhat different in concept from an operational semantics of a traditional programming language, where meaning is defined in terms of execution of the program. We need a meaning which is intrinsic to the model, and the function $\boldsymbol{f}$ provides a natural candidate. Given values for $\boldsymbol{Y}$ and t, evaluation of the *model* is then evaluation of the function $\boldsymbol{f}(\boldsymbol{Y}, t)$. To show that the meaning of a model is unchanged by a transformation, we thus need to show that this evaluation is unchanged for any possible values of $\boldsymbol{Y}$ and t. If this is the case, then any simulation algorithm will give the same result for the transformed model as for the original.

Not all ODE solvers evaluate the function $\boldsymbol{f}$ directly. Some, such as those based on the Rush–Larsen technique (Rush and Larsen, 1978), require the definition of $\boldsymbol{f}$ to be rearranged into particular forms, in order to enable more efficient simulation. However, these techniques do still involve evaluating the expressions within $\boldsymbol{f}$ during the course of the simulation, and so a semantics also based on this principle can still be applied.

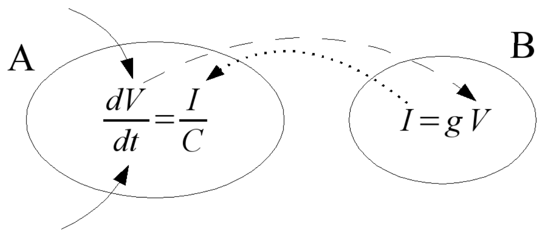
To avoid unnecessary complexity, our semantics focuses on the subset of CellML required for describing the types of mathematical models we are interested in, namely first order systems of ODEs. We also only consider CellML 1.0 models, not CellML 1.1. These restrictions are not onerous, however. The subset we consider is sufficient for the vast majority of models currently available in the `cellml.org` repository, including all the cardiac models. Also, through instantiating all imports, it is possible to convert a CellML 1.1 model into a form which fits with our semantics.

The operational semantics for CellML consists of an abstract data type describing a CellML document (Section 3.1), and a Haskell[1] interpreter for data with this type (Section 3.2). Note that a CellML model is not translated into a Haskell program—there is an interpretive layer. The simple example model of Equation 2.3.1 is encoded as shown in Listing 3.1.

CellML is a declarative language—as we have said above, CellML describes the structure and mathematics of a model, not how it should be simulated. This is analogous to a functional language, in which the focus is on what is to be computed, not how it should be computed. The

[1]Readers not familiar with the Haskell programming language are referred to Appendix A for a brief overview of its features and syntax.

Figure 3.1 The example CellML model in diagrammatic form, with arrows showing the flow of control.



Listing 3.1: The simple example model of (2.3.1) encoded using the Haskell data type for CellML models.

```
example =
  Model "example" units components connections
units =
  [ UDef "ms"
    (SimpleUnits 1 (−3) (BaseUnits "second") 0)]
components =
  [ Component "A" []
    [ VarDecl "time"    "ms"
    , VarDecl "voltage" "volt"
    , VarDecl "current" "ampere"
    , VarDecl "C" "farad"]
    [ Assign (Var "C") (Num 1 "farad")
    , Assign (Ode "voltage" "time")
        (Apply Divide [(Variable "current"),
                       (Variable "C")])]
  , Component "B" []
    [ VarDecl "V" "volt"
    , VarDecl "I" "ampere"
    , VarDecl "g" "siemens"]
    [ Assign (Var "g") (Num 0.3 "siemens")
    , Assign (Var "I")
        (Apply Times [(Variable "g"),
                      (Variable "V")])]]
connections =
  [ VarMap ("A","voltage") ("B","V")
  , VarMap ("B","I") ("A","current")]
```

semantics we have chosen is thus much like a lazy functional language, and so implementing the interpreter in such a language is a natural choice.

Lazy evaluation was chosen in order to avoid having to determine an evaluation order for the expressions making up the model explicitly. While a CellML model is organised into components, these logical groupings are useful from a model reuse perspective, rather than for evaluation—one cannot evaluate each component in turn. In our simple example for instance (see Figure 3.1), values for the voltage V and time t are passed into component A from the evaluator (solid arrows), then the voltage is passed to component B (dashed arrow) in order to compute the current I, which is passed back to A (dotted arrow) to compute the right hand side of the ODE. Real models have an even more complex information flow. While the evaluation order can be determined by a topological sort, it seemed more natural to us to have evaluation performed 'on demand', and implementation in a lazy functional language is well suited to this. (Incidentally, if we are to allow for implicit equations at a later date, the evaluation order becomes even harder to determine, and we expect lazy evaluation to be even more useful. This is discussed further in Section 8.3.)

Compared with the CellML specifications, our interpreter has several benefits. It is both concise and precise, making it easier to grasp conceptually as a whole. The CellML 1.0 specification is a 74 page document; our Haskell interpreter is only 1000 lines of comments and code (much of which is supporting code) and so requires roughly 20 pages when printed at a similar spacing and font size. Being amenable to computer manipulation, it also offers the possibility of being compared automatically against other implementations, although this is not an avenue we have explored as yet. The key advantage for our purposes, however, is that a formal semantics allows us to prove the correctness of optimisations for CellML. In particular, implementing partial evaluation also using Haskell makes the correctness proof for this technique relatively straightforward, as shown in Chapter 5.

Our formal semantics for CellML does have limitations, however. The primary one is that it only gives a semantics for a *subset* of CellML. There are features we have not implemented, such as allowing implicit equations, for example of the form $a+b = c+d$, and hence systems of algebraic equations.[2] Such features are used by very few existing CellML models, and by none of the cardiac models we are concerned with, but their inclusion is necessary if this is to be a complete semantics for CellML. We also do nothing with the metadata included within CellML documents. This may seem innocuous, but it is being proposed to include certain parameters

[2]Simultaneous linear equations, for example.

controlling simulation runs (e.g. the type of numerical algorithm to use) within this metadata, thus enabling more faithful reproduction of published results graphs.

3.1 The model data type

Mathematically, CellML specifies a vector $\boldsymbol{Y}$ of state variables and the ODE system of Equation (3.0.1):

$$\frac{\mathrm{d}\boldsymbol{Y}}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{Y}, t).$$

The data type representing a CellML model encodes the function $\boldsymbol{f}$, and thus the key concept we require is a mathematical expression. For this we use the *MathTree* type, a tree structure which corresponds directly to the MathML encoding of the expression.

```
data MathTree
    = Num Double URef   -- A cn element
    | Bool Bool         -- The result of a relational or logical operator
    | Variable Ident    -- A ci element
    | Apply Operator [MathTree]          -- An apply element
    | Piecewise [Case] (Maybe MathTree)  -- A piecewise element
    | Diff Ident Ident                   -- Diff v1 v2 ≡ dv1/dv2
data Case    -- Piecewise cases: Case condition result
    = Case MathTree MathTree
-- Units references are either to named units,
-- or an anonymous definition
type URef = Either UName Units
```

A few elements of this definition are worth noting.

- *URef* is used to annotate quantities with their physical units. The units model used is explained in Chapter 4, and the use of an **Either** type in Section 5.2.3.
- Boolean values do not exist explicitly in CellML documents;[3] the **Bool** constructor is just used for the results of certain operators.
- *Operator* is a simple enumerated type, with data constructors being capitalised versions of MathML element names.

[3]Although they may in CellML 1.2.

- Piecewise expressions may optionally contain an ‘otherwise’ clause, which is the result of the expression if no case conditions match, hence the **Maybe** *MathTree*.

The result of evaluating an expression can either be a number, or a boolean value. The value space also includes the special value *DynamicMarker* which is used for explicit binding time annotations; see Sections 5.2 and particularly 5.2.4 for further details.

```
data Value
    = Number Double
    | Boolean Bool
    | DynamicMarker
```

The remaining type definitions comprising a full CellML type provide the model structure, representing variables, components, connections, and the model as a whole. The top level datatype for a CellML model is *CellML*:

```
data CellML = Model Ident [UDef] [Component] [Connection]
```

This gives the model a name, and states that it consists of units definitions (see Chapter 4), components, and connections. A component is also identified by name, and may contain further units definitions (scoped locally to the component), variable declarations, and mathematics:

```
data Component = Component Ident [UDef] [VarDecl] [MathAssignment]
```

A variable declaration is simply used to define the units of the variable:

```
data VarDecl = VarDecl Ident UName
```

As explained above, our semantics is restricted to explicit equations, where a variable or derivative is assigned a value, given by some expression (possibly a constant). Thus each expression within a component must be an assignment of an expression to either a variable or a derivative:

```
data MathAssignment = Assign EnvKey MathTree
data EnvKey
    = Var Ident
    | Ode Ident Ident
```

The *EnvKey* type will be discussed further in the next section; for now we note that variables are referred to by name, and a derivative is identified by the names of the dependent and independent variables. Several examples of assignments can be found in Listing 3.1; for instance the statement *Assign* (*Var* ”C”) (*Num* 1 ”farad”) which assigns the constant value 1 to the variable

C.

Finally, to simplify the interpreter implementation, our *Connection* type is directional, thus incorporating the information obtained by analysing the encapsulation hierarchy and variable interfaces (this analysis is performed as part of the syntax conversion from the XML encoding of CellML to our datatype). The second variable is assigned the value of the first. Variables being mapped are identified by a pair giving the name of the component in which they are declared, and the variable name.

```
data Connection = VarMap (Ident , Ident) (Ident , Ident)
```

3.2 The interpreter

In order to assign a meaning to CellML models, we define an interpreter for models defined using the data type in Section 3.1. As we have indicated above, the meaning of a CellML model must be independent of whatever method is used to perform simulations of the model. Our interpreter thus requires two inputs: the model itself (in the form of Equation 3.0.1), and values for the state variables $\boldsymbol{Y}$ and time t. Evaluation of the model is then defined as evaluation of the function $\boldsymbol{f}(\boldsymbol{Y}, t)$. The full code of the interpreter is given in Appendix C; here we describe the main features and explain how they relate to a model's meaning.

At the interpreter's core is the function *eval*, which evaluates expressions defined by a *MathTree* to obtain a *Value*:

```
eval :: Env → MathTree → Value
```

Prior to evaluating any expressions, however, we must first transform the model into an environment (of type *Env*) which is a mapping from variables or ODEs to their defining expressions. This is a more convenient representation for evaluation, and is a common technique when writing interpreters.

Many types of environment can be defined. The general environment concept is of a mapping from keys to values. We use a polymorphic *Environment* type which can be parameterised with types for the keys and values used to give a specific type of environment. The *Environment* type is provided with various functions to perform operations on environments. The *find* function is used to determine the value associated with a given key. It throws an exception if the key is not defined; the related *maybe_find* function instead returns a **Maybe** value, yielding **Nothing** if the key is not found. The function *define* is used in building environments, and adds a new key–value mapping to an environment, yielding a new environment. There is also a function *names* which generates a list of the keys defined in an environment. The types of these functions are as follows.

```
find :: Environment k v → k → v
maybe_find :: Environment k v → k → Maybe v
define :: Environment k v → k → v → Environment k v
names :: Environment k v → [k]
```

The environment type used for a CellML model, *Env*, is defined as follows, with the keys having type *EnvKey* and values type *EnvValue*:

```
type Env = Environment EnvKey EnvValue
data EnvKey = Var Ident | Ode Ident Ident
data EnvValue
    = Expr MathTree
    | Val Value
    | InternalData (VarUnitsEnv, UnitsEnvs, Env)
```

Keys, which represent either variables or derivatives, can thus be bound either to expressions (*Expr MathTree*) or simple values (*Val Value*). To continue with our earlier example, the assignment of 1 to *C* would be represented in a model environment by the mapping from *Var* ”A,C” to *Val* (*Number* 1). The variable name used is explained below. The *InternalData* value is bound to the variable named ”” (which is an illegal identifier for CellML) and is used as a convenient way to pass some extra information between parts of the interpreter and partial evaluator (see also Chapter 5): an environment storing the units of each variable, environments storing the units defined in the model, and an environment containing the initial conditions for simulation.

Given this environment, simulation of a model is then performed by the *run_env* function, which evaluates the right-hand side of each ODE, and defines the results in a new environment, using a helper function *elookup* (defined later) which looks up a key in an environment and evaluates the defining expression within that environment, using *eval*.

```
run_env :: Env → [EnvKey] → Env
run_env model_env derivs
  = foldr eval_deriv empty_env derivs
  where -- Evaluate the RHS of a single ODE
        eval_deriv :: EnvKey → Env → Env
        eval_deriv d env = define env d (Val (elookup model_env d))
```

Here the fold builds up the result environment using *eval_deriv* applied to each derivative in the model in turn. This helper function adds a new binding in the environment, mapping the derivative it is given to the result of evaluating that derivative's definition (i.e. the right-hand side of the relevant ODE) in the model environment. The result is thus an environment binding each derivative in $\frac{\mathrm{d}\boldsymbol{Y}}{\mathrm{d}t}$ to the corresponding *Value* in $\boldsymbol{f}(\boldsymbol{Y}, t)$.

In generating the model environment (using the *load_cellml* function), we perform various canonicalisations, in order to simplify the core of the interpreter.

- Within the environment, components are no longer explicitly represented. We still need to distinguish between variables defined in separate components which have the same name, however, and so all occurrences of variable names are changed to use the 'full name' of the variable, including the name of the component within which it is defined. This 'full name' is defined by the *full_ident* function.

- Units may also be defined within components as well as globally for the whole model, and so a similar transformation is applied to unit names.

- Connections are represented by a simple assignment of one variable to another. The connection *VarMap* (*cname1*, *vname1*) (*cname2*, *vname2*) is thus represented by a binding in the environment of the expression *Expr* (*Variable* (*full_ident cname1 vname1*)) to the key *Var* (*full_ident cname2 vname2*).

- Otherwise clauses of piecewise expressions are converted into an extra case, so that all piecewise expressions have the form *Piecewise cases* **Nothing**. The associated condition is given as **Bool True**, to preserve the expression semantics.

- For any ODE defined, we also add an ‘alias’ to the environment. This alias uses the ‘source’ variable names for both dependent and independent variables, where the source variables are found by following connections until the original definition of the variable is found. Since these sources are unique, if an ODE is referenced in a component different from that in which it is defined, we can thus use the alias to find the definition.

  For example, consider the ODE $\mathrm{d}V/\mathrm{d}t$ in Listing 2.3, defined in the *membrane* component. For this ODE the following alias will also be added to the model environment:

  ```
  define env (Ode "membrane,V" "environment,time")
             (Expr (Diff "membrane,V" "membrane,time"))
  ```

  If the value of $\mathrm{d}V/\mathrm{d}t$ were required in another component, when the use of local variable names fails to find an entry in the environment, the alias will be looked up, and hence the ODE definition found.

Performing static checks such as those described in Chapter 4 also simplifies the interpreter by allowing us to avoid checking for various error conditions. A particularly important static check ensures that there are no cycles (e.g. $a = f(b), b = g(a)$) within the mathematics. This permits use of a simple recursive evaluation strategy for expressions, without needing to check for non-termination due to infinite recursion.

We now proceed to describe the evaluation itself. Evaluation of expressions within an environment is performed by the *eval* function, with the function *apply* defining operator behaviour in terms of standard Haskell functions. While CellML has no user-defined functions, it does have many mathematical operators ‘built-in’, and so *apply* is quite lengthy. A convenience function *elookup* combines the actions of looking up an identifier and evaluating its definition.

The definition of *eval* is simple:

```
eval :: Env → MathTree → Value
-- Constants are easy.
eval env (Num n _)
    = Number n
eval env (Bool b)
    = Boolean b
-- Lookup the variable's definition and evaluate it.
```

```
eval env (Variable v)
    = elookup env (Var v)
-- Lookup the ODE's definition and evaluate it.
eval env (Diff var bvar)
    = elookup env (Ode var bvar)
-- Evaluation of apply depends on the operator.
-- Lazily evaluate the operands.
eval env (Apply operator operands)
    = apply operator (map (eval env) operands)
-- Evaluation of a piecewise expression short-circuits when a True
-- condition is found.
eval env (Piecewise cases Nothing)
    = case foldr ecase Nothing cases of
        Just v  → v
        Nothing → error "fallen off end of piecewise"
    where
      ecase :: Case → Maybe Value → Maybe Value
      ecase (Case cond res) rest = case eval env cond of
            Boolean False → rest
            Boolean True  → Just (eval env res) -- short-circuit
            _ → error ("conditional does not evaluate to a boolean: "
                        ++ show cond)
```

We leave the full definition of *apply* for Appendix C, including just a few key operators here by way of illustration.

```
apply :: Operator → [Value] → Value
apply Plus operands        -- nary addition
    = Number (sum (map (get_num) operands))
apply Minus [operand]      -- unary minus
    = Number (0 - (get_num operand))
apply Minus [a, b]         -- binary minus
    = Number ((get_num a) - (get_num b))
apply Divide [a, b]        -- binary divide
    = Number ((get_num a) / (get_num b))
apply Times operands       -- nary multiplication
    = Number (foldl1 (*) (map (get_num) operands))
apply Exp [operand]        -- unary exponential fn
    = Number (exp (get_num operand))
apply And operands    -- nary logical and
    = Boolean (and (map (get_bool) operands))
apply Lt [a, b]
    = Boolean (get_num a < (get_num b))
```

The definition of *elookup* is straightforward except for the case where we need to look up the definition of an ODE, where we must consider the possibility that it may have been defined in a different component from that in which the reference occurs, and so we must look up the definition using the source variable names instead, to find the alias defined by the canonicalisation process described above.

```
elookup :: Env → EnvKey → Value
elookup env (Ode var bvar)
    = case maybe_find env (Ode var bvar) of
        Just (Expr e) → eval env e
        Just (Val v)  → v
        Nothing → elookup env ode_src
    where ode_src = (Ode (find_src env var) (find_src env bvar))
elookup env key
    = case find env key of
        Expr e → eval env e
        Val v  → v
```

These definitions provide us with a formal, mathematically tractable definition of what it means to evaluate (and hence simulate) a CellML model. The next chapter considers various static checks which may be applied to CellML models, and then in Chapter 5 we use the semantics defined here to prove the correctness of our optimisation of CellML by partial evaluation.

4

Validation of CellML models

The ultimate aim of validation is to prevent incorrect results. Humans will always make errors, and so the more errors that can be automatically detected, the more reliable our system becomes. Detecting errors early is also important—much time will be saved if we can find errors by an analysis of the model, rather than waiting until simulation time, or even for an analysis of the simulation results.

There are a variety of different forms of validation that can be applied to CellML models, with different tools needed to perform each one. The basic levels utilise standard validation tools for XML files, verifying that the given model really does conform to the CellML specification. This is discussed in Section 4.1.

Other checks are not so suited to standard tools. One in particular that we have implemented is to check for dimensional consistency. Errors in this area generally cause wrong results, rather than program failure, so catching them at 'compile time' is especially important. This is the subject of the remaining sections. Section 4.2 introduces the topic, and the conceptual model of units used in CellML is described formally in Section 4.3. The units checking algorithm is given in Section 4.4, and we conclude with some discussion in Section 4.5 placing our algorithm in the context of related work, and considering possible future adaptations:

both to account for possible developments in the CellML language, and whether improvements could be made to the CellML units model itself.

Much of this work has been published (Cooper and McKeever, 2008), including an algorithm for automatically converting between compatible units, which we do not describe here.

4.1 XML validation

There are a variety of standard approaches to validating XML documents, which can therefore also be applied to CellML. They allow us to automatically check many of the rules given in the CellML specifications that define what constitutes a valid CellML model. At the lowest level, an XML parser checks that the file contains well-formed XML. The next level is to compare a CellML document against a *schema* that defines what is allowable content. This verifies that the grammar is correct—that elements and attributes appear in the correct places, and that text content conforms to the appropriate data type. We have written schemas for CellML 1.0 using the RELAX NG[1] and Schematron[2] schema languages.

RELAX NG is designed to address two aspects of XML validation: validating the structure of an XML document, and providing a connection to datatype libraries that validate the content of text nodes and attributes. It has a strong mathematical background, with a theoretical basis similar to regular expressions. A key feature for our purposes is that it permits 'co-occurrence constraints' in which the *value* of one node changes the *content model* of another. This can be used to validate certain aspects of the mathematical content of models. For example, we can ensure that each operator is given the correct number of operands. Our RELAX NG schema is available from the CellML website.

Schematron schemas are more suited to expressing so-called 'business rules' restricting an

[1]`http://relaxng.org/`
[2]`http://www.schematron.com/`

XML document's content. XPath[3] expressions may be used to state more complex relationships between sections of the document. CellML models include many name references: mathematics can refer to variables defined in the enclosing component, units definitions refer to other units definitions, etc. We need to ensure that the target of such a reference exists, and this is done by the Schematron schema.

Other aspects of validation, however, are either difficult or impossible to perform using standard tools, and hence bespoke solutions must be produced. One such aspect is to check for cycles within the data structures implied by a CellML model. The encapsulation hierarchy of components, for instance, should represent a collection of tree structures, as should units definitions. These conditions are trivial to check (e.g. by a topological sort) if the model has been parsed to create graph structures within a program. The same technique can be used to check that the mathematical equations are acyclic, as is required by our CellML semantics (see Section 3.2).

The main contribution of our CellML validation suite, however, is in checking for the consistent use of physical units, and we proceed to discuss this in detail.

4.2 Units and dimensions

Dimensions and physical units are widely used in the sciences, but there is comparatively little support for these concepts in programming languages. Their usefulness for checking the correctness of mathematical models of reality is well known: similarly to the use of types for catching nonsensical programs, we know that an equation cannot be correct if constraints on the dimensions of the quantities involved are not met. For example, if a length and a time are added together, we know that the model must be incorrect in some fashion. Similarly if lengths measured in metres and feet are compared without a suitable conversion then simulations of the model are unlikely to give sensible results.

Such errors may seem trivial, but they are not uncommon. Scientific programs are often

[3] `http://www.w3.org/TR/xpath`

composed of many different components, sometimes written by different authors, using differing units, especially for multi-scale models. Units conversions at the interfaces are thus essential for correct operation. Input and output of quantities with units also requires care. In one instance, a space shuttle was instructed to bounce a laser off a 10,000 mile high mountain, instead of the intended 10,000 feet, and so rolled away from the Earth (Neumann, 1985). Units errors can also be costly—the loss of NASA's Mars Climate Orbiter in 1999 was due to the software failing to convert between imperial and metric units (Isbell and Savage, 1999). The need for automated systems to detect such errors is illustrated in the following quote from Edward Weiler, NASA's Associate Administrator for Space Science.

> "People sometimes make errors. The problem here was not the error, it was the failure of NASA's systems engineering, and the checks and balances in our processes to detect the error. That's why we lost the spacecraft." (Isbell et al., 1999)

Most programming languages do not provide such checks and balances.

Before presenting our approach to solving this problem, we briefly describe the fundamental concepts of a system of units. Every physical *quantity* is considered to exist in some *dimension*, for example length or force. Two quantities may exist in the same dimension, yet have different *units*; for example metres and feet are both units of length. If the relationship between feet and metres is known, it is possible to convert a quantity expressed in one unit to be given in the other.

Dimensions can be either base dimensions, or derived dimensions, which are defined in terms of other dimensions. Dimensions can be combined by multiplication, division, or exponentiation; acceleration is length divided by time squared, for instance. Quantities can be combined in the same way, with the resultant quantity existing in the appropriate combined dimension, and having combined units.

Dimensions and units are in some ways very similar to the programming language concept of *types*. Just as expressions in a strongly-typed programming language must be *well-typed* in order for the program to compile, so mathematical expressions must be *dimensionally consistent*. Rules can be given defining this notion, just as type inference rules define well-typed programs, as we show in Section 4.4. However, several properties necessary for dimension checking cannot be expressed in a conventional type system. For example, rather than dealing with a single type for each expression, a mathematical expression is associated with a dimension, which can be regarded as an equivalence class of units under operations of a units algebra (e.g. $[\mathrm{m}\,\mathrm{s}^{-1}\,\mathrm{s}] = [\mathrm{s}\,\mathrm{m}\,\mathrm{s}^{-1}] = [\mathrm{m}]$). Furthermore, units definitions are not semantically separate from the value space of the programming language: exponents (such as in $[\mathrm{s}^{-1}]$) contain numerical values. Unless exponents are restricted to be constants, a units checker must be capable of actually evaluating expressions. Where the expression cannot be evaluated, the program must either be rejected, or the programmer trusted and the program allowed even though the units cannot be fully checked.

The subject of automatically checking and converting units has been discussed many times already, in a variety of contexts (e.g. Karr and Loveman III, 1978; House, 1983; Männer, 1986; Dreiheller et al., 1986; Baldwin, 1987; Kennedy, 1994; Allen et al., 2004; Wilkinson et al., 2005), although Wilkinson et al. (2005) only checks unit at run time, and Baldwin (1987) and Kennedy (1994) do not support unit conversions. The most fundamental difference, as compared with our work, is the context in which the work is done: all of these consider adding some form of units support to a programming language, as opposed to the modelling language approach we have adopted.

The quality of the work varies considerably, but all the papers mentioned above restrict the kinds of units that can be expressed in some fashion, which we did not want to do since CellML's units model, described in the next section, is very general. One such restricted area is support for exponents on units. In most cases these are only permitted to be constants given in the units definition or declaration. Allen et al. (2004) allow exponents to be given as a `final int`, i.e. an integer fixed at compile time, which allows them to give units to the `power` function; it will be seen that our solution using ideas from partial evaluation is still more general.

4.3 CellML units definitions

The CellML units model was developed with mathematical modellers in mind, and is hence intended to be flexible and intuitive. It allows for base units and units derived in various ways: by scaling, products, etc. as is the case for most units, and also by the use of an offset, as is required for degrees Fahrenheit. The aim is to allow model authors to work in whatever set of units they feel most comfortable, while still ensuring that their models can be integrated with those of other authors using other units.

For the purposes of this exposition, we describe the units model using the abstracted version given in Haskell below. Complete Haskell code can be found in Appendix D.[4]

Definition 4.1 The abstracted CellML units model

```
data Units = ComplexUnits [ComplexUnitsRef]
           | SimpleUnits Multiplier Prefix Units Offset
           | BaseUnits Name
data ComplexUnitsRef = Unit Multiplier Prefix Units Exponent
type Multiplier = Double
type Prefix     = Double
type Exponent   = Double
type Offset     = Double
type Name       = String
```

There are three classes of units in CellML: *base*, *simple* and *complex*.

Base units are defined just with a name, and form the primary units for their base dimension—dimensions are thus implied by the combination of base units, rather than explicitly defined. *BaseUnits* ”metre” would be one example, as the primary units of length.

Simple units are defined as a linear function of another simple or base unit (note that this crucial restriction is not shown in the schema above), and thus represent a scaling of a single base unit, with the *Prefix* and *Multiplier* fields specifying the scaling factor (see below). These are also the only units for which an offset is allowed, e.g. to define the Fahrenheit scale.

Complex units are the product of multiple units. Each unit referenced in the product can be

[4]Appendix A provides a brief overview of the features and syntax of Haskell.

scaled (*Prefix* and *Multiplier* fields) and raised to an arbitrary real power (*Exponent* field).

Both the *Prefix* and *Multiplier* give a multiplicative term in a units definition. This decision was made so that units could be defined in a more natural fashion for modellers, using SI prefixes, with *milli* = -3, *mega* = 6, etc. The overall *multiplicative factor* of a units reference with multiplier m, prefix p and exponent e is defined as $m(10^p)^e$. The multiplicative factor of a units definition is the product of the factors on the units references in the definition.

For both simple and complex units, the new units $[\mathrm{U}]$ are defined in terms of the (basic product of) the constituent units $[\mathrm{u}]$ by the formula

$$[\mathrm{U}] = m[\mathrm{u}] - o,$$

where m is the multiplicative factor of the new units and o is the offset (which is 0 for complex units). For quantities x_{new}, x_{old} measured in the different units, the relationship is

$$x_{new}[\mathrm{U}] = \frac{1}{m}\left[\frac{\mathrm{U}}{\mathrm{u}}\right]x_{old}[\mathrm{u}] + o[\mathrm{U}]. \tag{4.3.1}$$

That we need to use the reciprocal of the multiplicative factor here can be seen if we consider converting from metres to kilometres. The latter has a multiplicative factor of 1000, but 1 metre is 0.001 kilometres. Understanding the behaviour of the offset is a little harder. The intention is for the Fahrenheit scale to be declared as

```
fahrenheit = SimpleUnits (5/9) 0 celsius 32
```

whence 1°F= $(5/9)$°C-32°F, but 1°C is $(9/5 + 32)$°F.

CellML also defines two special units which are worthy of mention here. Firstly, `boolean` is an invented unit used for truth values. This unit is special, in that it is like a base unit, but no units can be derived from it. Secondly, `dimensionless` is defined as a base unit, for quantities that have no dimension (i.e. are plain numbers), or the ratio of two quantities with the same units.

There are several ‘operations’ on units that we use below. These are needed in order to reason about and develop our algorithms. Firstly we consider the question of a canonical form for a units definition, and then in Section 4.3.2 we define the operations which combine units definitions to form new units, thus forming an algebra of units.

4.3.1 Canonicalisation operations

It is possible for multiple units definitions to refer to the same physical units. For example, energy could be given in terms of Joules, or in terms of Newton-metres (expanding the definition of Joule), or in base SI units as $kg\, s^{-2}\, m^2$ or $kg\, m^2\, s^{-2}$. It greatly simplifies the design and analysis of algorithms involving units if they are always expressed in some canonical form, since there is then no need to add similar code in multiple places to cope with variations in the form.

The most obvious canonical form to choose (and one that has been chosen before, e.g. House 1983; Allen et al. 2004) is to express units in terms of products of powers of base units—the last of the forms in the above example. This is a particularly useful choice in that it does not have multiple levels of unit references in a single definition, which removes the requirement of recursing through units definitions from other units algorithms. It allows us to check the dimensional equivalence of two units definitions by a straightforward comparison of their canonical forms, and eases calculating a units conversion expression for a quantity, since only the multiplicative factors and offsets of the canonical form are required. Importantly, this is also the canonical form used in the CellML specification, and our canonicalisation algorithms are based on those given there, but extended to maintain details (such as multiplicative factors) needed for units conversion.

Converting definitions to this canonical form is achieved by the expansion algorithm given below (Algorithm 4.1). However, this algorithm does not give the whole story, since it will keep multiple references to the same base unit. Keeping with the example above, it would expand Joule as $kg\, m\, s^{-2}\, m$. We also present a separate simplification algorithm (Algorithm 4.2) which replaces such multiple references by a single reference, with the appropriate exponent and multiplier.

There are two reasons for keeping simplification as a separate algorithm. The first is that it aids understanding of the canonicalisation process to split it into multiple steps. The second is that simplification is also useful in other contexts—multiple references to the same unit can also occur when multiplying units definitions (e.g. when taking the product of two quantities). We would like the product of $[\mathrm{C\,V^{-1}}]$ and $[\mathrm{V}]$ to be $[\mathrm{C}]$, not $[\mathrm{C\,V^{-1}\,V}]$, for instance.

Algorithm 4.1 Expansion of units definitions

```
expand (BaseUnits n)
  = BaseUnits n
expand (SimpleUnits m p u o)
  = case expand u of
      BaseUnits n
        → SimpleUnits m p (BaseUnits n) o
      SimpleUnits m' p' u' o'
        → SimpleUnits (m*10**p*m'*10**p') 0 u' (o+o'/(m*10**p))
expand (ComplexUnits us)
  = ComplexUnits (expand' us)
  where
    expand' [] = []
    expand' ((Unit m p u e):urefs) = new_urefs ++ expand' urefs
      where
        new_urefs  = prop_mf new_urefs'
        new_urefs' = case expand u of
          BaseUnits n → [Unit m p u e]
          SimpleUnits m' p' u' _ → [Unit m' p' u' 1]
          ComplexUnits urefs' → map prop_e urefs'
            where prop_e (Unit m' p' u' e')
                    = Unit ((m'*(10**p')**e')**e) 0 u' (e*e')
        prop_mf [] = []
        prop_mf ((Unit m' p' u' e'):us)
          = (Unit (m*(10**p)**e * m') p' u' e') : us
```

The expansion algorithm can be hard to follow, so we illustrate and explain with a couple of examples using the following units definitions:

```
kilo   = 3
newton = ComplexUnits [Unit 1 0 kg 1, Unit 1 0 m 1,
                       Unit 1 0 s −2]
kPa    = ComplexUnits [Unit 1 kilo newton 1, Unit 1 0 m −2]
celsius    = SimpleUnits 1 0 kelvin −273.15
fahrenheit = SimpleUnits (5/9) 0 celsius 32
```

Example 1 First let us consider *expand kPa*. We need to apply *expand'* to each of the unit references in its definition. When processing the reference to *newton*, we need to expand this definition too using the same procedure. Each of the unit references in the definition of *newton* is to base units, so *expand* is equivalent to the identity function in this case.

We then map the function *prop_e* onto the unit references of *newton*. This function updates each unit reference to reflect the fact that it occurs within the context of a unit reference, and so may be subject to an extra exponentiation. It does this by raising the multiplicative factor to the appropriate value (*e*) and multiplying the exponent by the same value. For our example, the exponent on the enclosing unit reference is 1, so *prop_e* is equivalent to the identity function.

Next *prop_mf* is applied to the list of unit references from *newton*. This function propagates the multiplicative factor of the enclosing unit reference (to *newton* itself) into the list, by multiplying it into the multiplier of the first unit reference. In our example this sets the multiplier of the *kg* reference to 1000.

The *expand'* function then proceeds to the next unit reference, which is to *m*. *m* is a base unit, so we retain the same reference in the new list. We thus obtain a final answer of

```
kPa_exp = ComplexUnits [Unit 1000 0 kg 1, Unit 1 0 m 1,
                        Unit 1 0 s −2, Unit 1 0 m −2]
```

which illustrates the need for a further simplification step.

Example 2 Next we consider *expand fahrenheit*.

```
expand fahrenheit
  = expand (SimpleUnits (5/9) 0 celsius 32)
  = expand (SimpleUnits (5/9) 0 (SimpleUnits 1 0 kelvin −273.15) 32)
```

Now,

```
expand celsius
  = expand (SimpleUnits 1 0 (BaseUnits "kelvin") −273.15)
  = SimpleUnits 1 0 (BaseUnits "kelvin") −273.15
```

and so

```
expand fahrenheit
  = SimpleUnits (5/9) 0 (BaseUnits "kelvin") (32 −273.15/(5/9))
  = SimpleUnits (5/9) 0 kelvin −459.67
```

Some explanation of Algorithm 4.2 is called for. Simplification is the process of replacing multiple references to the same unit by a single reference. Step 1 defines the form of this

Algorithm 4.2 Simplification of units definitions

For simple or base units, simplification is the identity function. For complex units,

1. for each unique *Units* referenced, replace all the references by a single reference (treat `dimensionless` last):
   (a) sum all the exponents;
   (b) if the result is 0, the reference is to `dimensionless`, otherwise it is to the original *Units*, with the resulting exponent;
   (c) the new *Multiplier* is the product of the multiplicative factors on the original references;
   (d) the new *Prefix* is 0;
2. if `dimensionless` is referenced in the list and has unitary *Multiplier*, then remove the reference;
3. if the list is empty then the result is `dimensionless`, otherwise it is a *ComplexUnits* with the generated list of references, one for each unique *Units*, sorted by unit name.

reference. References in a *ComplexUnits* definition can be thought of as the referenced units being multiplied together to form a new unit. In algebra, when multiplying x^a and x^b the exponents are added, giving x^{a+b}; the same applies here, hence step 1(a).

A dimension with exponent 0 doesn't really exist—a ratio of two time values is a plain number, for example. Hence in step 1(b) `dimensionless` is used where units cancel out.

Steps 1(c) and 1(d) give the multiplicative factor for the new reference; this is contained entirely in the *Multiplier* field for simplicity of the algorithm.

Step 2 is included for readability purposes. When simplifying units such as $[\mathrm{m\,s^{-1}\,s}]$ we think of the result purely as metres, not as metres multiplied by a dimensionless constant 1. There are only two cases where `dimensionless` should be retained in the result. The first is that given in step 3: if no units are referenced other than `dimensionless`, then the resulting units *are* `dimensionless`—a ratio of two time values *is* a plain number. The second is the case where a reference to `dimensionless` exists with a non-unitary multiplier. This occurs in situations such as simplifying $[\mathrm{m\,s^{-1}\,ms}]$, where we need to retain a multiplicative factor of 0.001 in the resulting units definition. Since in the units model presented here such factors can only occur on unit references, a reference to `dimensionless` with the required factor is kept.

As an example, we apply the above algorithm to the expanded *kPa* definition we obtained earlier:

```
kPa_exp = ComplexUnits [Unit 1000 0 kg 1, Unit 1 0 m 1,
                        Unit 1 0 s −2, Unit 1 0 m −2]
```

There is only one instance of multiple references to the same *Units*, namely those to *m*, with exponents 1 and -2. Summing these gives -1, and the product of the multiplicative factors is 1, so we replace the two references by the single reference *Unit* 1 0 m -1. There are no references to `dimensionless`, and the list is non-empty, so the result is

```
kPa_exp_simp = ComplexUnits [Unit 1000 0 kg 1, Unit 1 0 m −1,
                             Unit 1 0 s −2]
```

4.3.2 Basic operations of a units algebra

Dimensional equivalence

This is an equivalence relation between units which holds if the units have the same dimensions, e.g. they are both times, or both areas. This is important, since many mathematical operators require operands having the same *dimension*, rather than identical units—we can add any lengths, no matter what units they are measured in.

Dimensional equivalence of two units definitions is checked by Algorithm 4.3.

Algorithm 4.3 Checking dimensional equivalence

```
1. Expand and simplify each definition (to express each units definition in canonical form
   as a product of powers of base units, as described above).
2. Extract two lists of the form (BaseUnitsName, Exponent).
3. Lexicographically sort each list by BaseUnitsName (to enable the comparison in the next
   step).
4. Return true iff the lists are equal.
```

Let us illustrate this process by comparing Joules and kiloPascals. First we define some units:

```
kilo   = 3
newton = ComplexUnits [Unit 1 0 kg 1, Unit 1 0 m 1,
                       Unit 1 0 s −2]
joule  = ComplexUnits [Unit 1 0 newton 1, Unit 1 0 m 1]
kPa    = ComplexUnits [Unit 1 kilo newton 1, Unit 1 0 m −2]
```

where *kg*, *m* and *s* are the usual SI base units for mass, length, and time, respectively. Applying step 1, we obtain the canonical forms

```
joule_can = ComplexUnits [Unit 1 0 kg 1, Unit 1 0 m 2,
                          Unit 1 0 s −2]
kPa_can   = ComplexUnits [Unit 1000 0 kg 1, Unit 1 0 m −1,
                          Unit 1 0 s −2]
```

Extracting lists as per step 2 gives us

```
joule_list = [("kg", 1), ("m", 2), ("s", −2)]
kPa_list   = [("kg", 1), ("m",−1), ("s", −2)]
```

which are already sorted, so step 3 does nothing. Finally, we see that these lists are not equal, since the exponents for "m" differ, so Joules and kiloPascals are not dimensionally equivalent, as we would expect.

Multiplication

The operator $\otimes$ represents multiplication of units, with simplification where appropriate. For example, $m\,s^{-1} \otimes s = m$. This is implemented according to Algorithm 4.4.

Algorithm 4.4 Multiplication of units definitions

1. Convert both operands to *ComplexUnits*:
   - *SimpleUnits m p u o* $\mapsto$ *ComplexUnits* [*Unit m p u* 1],
   - *BaseUnits n* $\mapsto$ *ComplexUnits* [*Unit* 1 0 (*BaseUnits n*) 1].
2. Simplify the *ComplexUnits* formed from the concatenation of the two lists of units references.

The conversion from *SimpleUnits* to *ComplexUnits* deserves further comment. In particular, note that the offset field is dropped in this conversion. This demonstrates that when units defined with respect to some reference point are combined with other units, the value of the reference point is no longer important. In terms of measurement scales, we are only concerned with the distance between two points on the scale, not their distance from the origin.

Exponentiation

Units can also be raised to a power; this operation is represented in the same way as standard exponentiation (e.g. s^2). To raise a units definition to the power *e* we use Algorithm 4.5, which simply alters the *Exponent* and *Multiplier* fields, converting the units into *ComplexUnits* if needed in order to do so.

Algorithm 4.5 Exponentiation of units definitions

```
exponentiate (BaseUnits n) e =
  ComplexUnits [Unit 1 0 (BaseUnits n) e]
exponentiate (SimpleUnits m p u o) e =
  ComplexUnits [Unit (m**e) p u e]
exponentiate (ComplexUnits urefs) e =
  ComplexUnits (map expo urefs)
   where
     expo (Unit m p u e')
          = Unit (m**e) p u (e'*e)
```

4.4 Units checking

Below we present type-inference style rules for performing units checking of mathematical expressions. These define dimensional consistency of an expression in much the same way as we could define whether it was well-typed, by giving assertions about the units of expressions in terms of assertions about the units of their subexpressions.

Therefore the units checking algorithm processes an expression tree in a bottom-up fashion, annotating nodes with their units as it progresses. Leaf nodes (numbers and variables) are all explicitly annotated with their units in the model. For compound expressions, the algorithm must select the rule whose conclusion matches the form of the expression, and check that the subexpressions have units matching the premise of the rule. If they do, the expression can be annotated with the units specified by the conclusion, which will often be derived from the units of the subexpressions, using the operations given in Section 4.3.2. It is these operations, and the fact that units definitions can contain elements from the value space, that primarily distinguish this algorithm from type checking. The latter issue will be discussed in Section 4.4.2.

Firstly we introduce some notation. Subexpressions are given as e_1, e_2, etc. Greek letters are used for qualifiers such as the order of a derivative, *f* stands for a function, and *op* for an operator.

We write $\mathrm{e}_1 :: u$ to say "e_1 has units dimensionally equivalent to u". That is, u stands for a whole class of physical units, rather than a single unit. This allows us to write the rules in a more standard style, since most operators merely require operands with dimensionally equivalent units, not identical units. The choice of which unit from the equivalence class to use for an expression is left as arbitrary here; it is important in the context of units conversion however (Cooper and McKeever, 2008).

4.4.1 Inference rules

We first present all the rules of the units checking algorithm, and then illuminate them with a few examples.

Addition (considered as an n-ary operator):

$$\frac{\mathrm{e}_1 :: u \ldots \mathrm{e}_n :: u}{\mathrm{e}_1 + \ldots + \mathrm{e}_n :: u} \tag{4.4.1}$$

Subtraction (binary operator):

$$\frac{\mathrm{e}_1 :: u \quad \mathrm{e}_2 :: u}{\mathrm{e}_1 - \mathrm{e}_2 :: u} \tag{4.4.2}$$

Unary minus:

$$\frac{\mathrm{e}_1 :: u}{-\mathrm{e}_1 :: u} \tag{4.4.3}$$

Multiplication (considered as an n-ary operator):

$$\frac{\mathrm{e}_1 :: u_1 \ldots \mathrm{e}_n :: u_n}{\mathrm{e}_1 \times \ldots \times \mathrm{e}_n :: u_1 \otimes \ldots \otimes u_n} \tag{4.4.4}$$

Division:

$$\frac{\mathrm{e}_1 :: u \quad \mathrm{e}_2 :: v}{\mathrm{e}_1/\mathrm{e}_2 :: u \otimes v^{-1}} \tag{4.4.5}$$

Logical operators:

$$\frac{\mathrm{e}_1 :: \texttt{boolean} \ldots \mathrm{e}_n :: \texttt{boolean}}{\mathrm{e}_1 \; \mathit{op} \; \ldots \; \mathit{op} \; \mathrm{e}_n :: \texttt{boolean}} \tag{4.4.6}$$

Relational operators:

$$\frac{\mathrm{e}_1 :: u \quad \mathrm{e}_2 :: u}{\mathrm{e}_1 \; \mathit{op} \; \mathrm{e}_2 :: \texttt{boolean}} \tag{4.4.7}$$

The abs, floor and ceiling functions:

$$\frac{\mathrm{e}_1 :: u}{f \, \mathrm{e}_1 :: u} \tag{4.4.8}$$

The exponential function, natural logarithm, factorial function, and trigonometric functions:

$$\frac{\mathrm{e}_1 :: \texttt{dimensionless}}{f \, \mathrm{e}_1 :: \texttt{dimensionless}} \tag{4.4.9}$$

Logarithms:

$$\frac{\mathrm{e}_1 :: \texttt{dimensionless} \quad \beta :: \texttt{dimensionless}}{\log_\beta(\mathrm{e}_1) :: \texttt{dimensionless}} \tag{4.4.10}$$

Powers:

$$\frac{\mathrm{e}_1 :: u \quad \mathrm{e}_2 :: \texttt{dimensionless}}{\mathrm{e}_1^{\mathrm{e}_2} :: u^{\mathrm{e}_2}} \tag{4.4.11}$$

Roots:

$$\frac{\mathrm{e}_1 :: u \quad \delta :: \texttt{dimensionless}}{\sqrt[\delta]{\mathrm{e}_1} :: u^{1/\delta}} \tag{4.4.12}$$

Derivatives:

$$\frac{\mathrm{e}_1 :: u \quad \mathrm{e}_2 :: v \quad \delta :: \texttt{dimensionless}}{\frac{\mathrm{d}^\delta \mathrm{e}_1}{\mathrm{d}\mathrm{e}_2^\delta} :: u \otimes v^{-\delta}} \tag{4.4.13}$$

Piecewise:

$$\frac{\mathrm{c}_1 :: \texttt{boolean} \ldots \mathrm{c}_n :: \texttt{boolean} \quad \mathrm{e}_0 :: u \ldots \mathrm{e}_n :: u}{\text{if } \mathrm{c}_1 \text{ then } \mathrm{e}_1 \text{ elif } \ldots \text{ elif } \mathrm{c}_n \text{ then } \mathrm{e}_n \text{ else } \mathrm{e}_0 :: u} \tag{4.4.14}$$

Examples

Consider the expression $2\,\mathrm{m} + 3\,\mathrm{m}$. This matches the form of the conclusion to rule (4.4.1), with $\mathrm{e}_1 = 2\,\mathrm{m}$ and $\mathrm{e}_2 = 3\,\mathrm{m}$. Both e_1 and e_2 thus have units dimensionally equivalent to (and indeed equal to) metres, and so $2\,\mathrm{m} + 3\,\mathrm{m} :: \mathrm{m}$.

Alternatively, suppose the expression $4\,\mathrm{kg} < 6\,\mathrm{km}$ is to be checked for consistent use of units. This is an application of a relational operator, and so by rule (4.4.7) we again require the operands to have dimensionally equivalent units. However, $4\,\mathrm{kg} :: \mathrm{kg}$ whereas $6\,\mathrm{km} :: \mathrm{m}$, which are in different dimensions (mass and length, respectively), and so the premise does not hold. The expression is thus not dimensionally consistent.

As a third example, consider the more complex expression $\mathrm{e} = 5\,\mathrm{m}/(10\,\mathrm{s})^2$. This expression matches the conclusion of rule (4.4.5), with $\mathrm{e}_1 = 5\,\mathrm{m}$, and $\mathrm{e}_2 = (10\,\mathrm{s})^2$. To determine the units of e we thus need to first determine the units of the subexpression e_2, using rule (4.4.11). Now $10\,\mathrm{s} :: \mathrm{s}$ and $2 :: \texttt{dimensionless}$, and so the premises are satisfied and $\mathrm{e}_2 :: \mathrm{s}^2$. Thus the units of e are given by $\mathrm{m} \otimes (\mathrm{s}^2)^{-1} = \mathrm{m\,s}^{-2}$.

Finally, consider the equation for *beta_n* taken from the modified Hodgkin–Huxley equations of Listing 2.3:

$$\beta_n = 0.125\,\mathrm{ms}^{-1} e^{\frac{V+75\,\mathrm{mV}}{80\,\mathrm{mV}}}$$

where $V :: \mathrm{m}V$ and $\beta_n :: \mathrm{ms}^{-1}$. The process of checking the units of this expression can best be represented by the following derivation tree. Working from axiomatic premises, we see that the appropriate rule can be satisfied at each step, and so the whole expression has consistent units.

$$\dfrac{\overline{\beta_n :: \mathrm{ms}^{-1}} \qquad \dfrac{\overline{0.125\,\mathrm{ms}^{-1} :: \mathrm{ms}^{-1}} \qquad \dfrac{\dfrac{(4.4.1)\,\dfrac{\overline{V :: \mathrm{mV}}\;\overline{75\,\mathrm{mV} :: \mathrm{mV}}}{V + 75\,\mathrm{mV} :: \mathrm{mV}} \qquad \overline{80\,\mathrm{mV} :: \mathrm{mV}}}{(V+75\,\mathrm{mV})/80\,\mathrm{mV} :: \texttt{dimensionless}}\,(4.4.5)}{e^{(V+75\,\mathrm{mV})/80\,\mathrm{mV}} :: \texttt{dimensionless}}\,(4.4.9)}{0.125\,\mathrm{ms}^{-1}e^{(V+75\,\mathrm{mV})/80\,\mathrm{mV}} :: \mathrm{ms}^{-1}}\,(4.4.4)}{\beta_n = 0.125\,\mathrm{ms}^{-1}e^{(V+75\,\mathrm{mV})/80\,\mathrm{mV}}}\,(4.4.7)$$

4.4.2 Partial evaluation and units checking

As we have already mentioned, units checking is made more complex than traditional type checking by the fact that units definitions can contain items from the value space, rather than being entirely separate. This is seen above in the rules for logarithms, powers, roots and derivatives ((4.4.10)–(4.4.13)), where the units of the whole expression contain values (β, e_2 and δ), rather than 'just' units definitions. A fully general units checker is impossible to write, since it would need to be able to evaluate arbitrary expressions in order to determine suitable units for the cases mentioned, but not all expressions are computable in a programming language (e.g. in the case of an endless loop or recursion) and not all values are known at compile time. We must therefore conservatively approximate the division between those values computable by the units checker, and those unknown until program run time.[5]

Much research has been done on this issue, as part of partial evaluation (Jones et al., 1993). Deciding whether a given expression is computable at compile time or not is done by a *binding time analysis*. Since we have a partial evaluator for CellML (see Chapter 5) we use this to evaluate required values where possible.

If the binding time analysis determines that these values cannot be computed, then the expression is marked as having unknown units, and units checking 'fails'. Note that even in this case we are still able to translate the model to code and run it, but we have lost the extra checks that units checking gives us, and we will not be able to perform units conversion where the failed expression is involved, which could lead to wrong results if a conversion is, in fact, needed. This is a different approach from standard partial evaluation, where the evaluation of expressions marked as dynamic is simply deferred until program run time. One could imagine a more complex units checking compiler adding a run time units checking framework to the generated executable if it was unable to determine all the units at compile time. In practice, we believe this would be more trouble than it is worth, since such cases are unlikely to occur.[6]

[5]Exact determination of the division is impossible, as this would imply a solution to the halting problem.

[6]One case where exponents might be dynamic values is if the space dimensionality of the physical problem is a run time parameter. As an easy workaround we suggest to run the units checking multiple times, once for each possible value of the parameter.

4.5 Discussion

We have presented above our approach to automatically validating certain properties of CellML models. This assists model authors (and users) in catching errors at an early stage, thus saving time. The checks described have focused on ensuring that models do conform to the CellML specification.

Our main contribution is in checking for the consistent use of physical units. Partial evaluation techniques have been leveraged in order to handle quantities raised to arbitrary powers robustly. A recent publication (Cooper and McKeever, 2008) describes how to extend this work to perform automatic conversion between quantities measured in different but dimensionally equivalent units.

There are aspects of dealing with units addressed in other papers that we have not covered in our work thus far. One of these is the question of *inferring* a ‘units signature’ for an expression. Kennedy (1994) presents an inference algorithm along the lines of ML type inference. Since in CellML all constants and variables are explicitly annotated with their units, we have not needed to make any inferences, but this would be an interesting area to examine, especially in conjunction with the incorporation of user-defined functions. These may well be included in a future version of CellML; such an extension has more implications for units conversion than for units checking, however, so we refer the reader to our paper for further discussion (Cooper and McKeever, 2008).

Another area we have not addressed is the theoretical underpinning of our units model. Allen et al. (2004) present a dimensional algebra based on a free abelian group, whereas Kennedy (1994) defines a formal type system for dimensions. To deal satisfactorily with inferring units we would require some such basis for our work as well. Kennedy notes that a dependent type system is required in order to handle values in units definitions (Kennedy, 1994).

The CellML units model is complex, since it attempts to allow for any sorts of units that modellers might wish to use. As we discovered whilst doing this work, this can make it confusing to work with. One interesting issue is that the units model does not distinguish explicitly between °C (i.e. a temperature measurement in degrees Celsius) and C° (i.e. the difference between two such measurements). This can make certain expressions ambiguous: for example, twice 2°C could refer to $550.3\,\mathrm{K}$ or a temperature difference of $4\,\mathrm{K}$. To an extent the distinction is made implicitly, in that °C will always be expressed using *SimpleUnits* with an offset, whereas C° will generally be expressed using *ComplexUnits*, where the offset is dropped, since temperature differences are usually used in a context where they are combined with other units (for example, a temperature gradient might be measured in $\mathrm{C}^{\circ}\,\mathrm{m}^{-1}$). To avoid this complexity, SBML has decided not to use units with offsets at all, and indeed as of the time of writing no models in the `cellml.org` repository use units with offsets. An alternative solution would be to follow Allen et al. (2004) in explicitly modelling scale readings as separate entities from other quantities.

Another shortcoming, not so much in the units model itself as in the XML encoding of it, is also demonstrated by the Fahrenheit definition:

```
fahrenheit = SimpleUnits (5/9) 0 celsius 32
```

The multiplier $5/9$ cannot be exactly represented in decimal form, and so cannot be given exactly in a CellML model. There are two approaches which could be taken to resolve this. One is to add a ‘divisor’ field, since $9/5$ can be written exactly. A more general solution would be to allow the multiplier to be given by a mathematical expression encoded in MathML.

To achieve generality, CellML allows exponents in units definitions to be any real number. This has disadvantages, however, due to the inexact nature of floating point arithmetic. Comparing two units for equality (or even dimensional equivalence) then involves comparing two floating point values, and hence the test cannot be exact. Since there are no ‘real world’ units which require such general exponents, other work (House, 1983, p. 368) favours using only rational powers, thus allowing exact comparisons to be made with no practical loss in expressiveness.

Finally, we note again that there are two different senses of the word ‘validation’ in the context of CellML. We have discussed checking that a model conforms to the definition of what constitutes a CellML document (as given in the specifications). The sense which is probably more familiar to modellers and physiologists is to validate a mathematical model against experimental data, to determine to what extent the model matches reality (and can therefore elucidate reality). This is closely related to the task of model curation—annotating models in the repository to indicate what a user may expect from them.

At the most basic level of CellML model curation, a given CellML encoding of a mathematical model can be assigned one of four curation levels.

Level 0: the model has been implemented, but has *not yet* been through the process of curation.

Level 1: the model has been implemented and corrected, if necessary, to *accurately represent* the published model.

Level 2: the model has been implemented and corrected, if necessary, to *accurately reproduce* the published results.

Level 3: the model has been implemented and corrected, if necessary, to *satisfy* domain specific biophysical constraints (e.g. conservation of mass and charge, or thermodynamic constraints).

Potentially, a CellML model may be assigned multiple curation levels, but historically at least, CellML models which accurately represent a model as it was published (level 1) will not satisfy the requirements for level 2. With recent tool developments to assist modellers, it is hoped that new models being developed will satisfy curation levels 1–3 with a single version of the CellML model.

Having now considered correctness of the models themselves, we next turn our attention to simulation efficiency. The next two chapters introduce two separate optimisation techniques, showing how they may be applied to CellML, and proving that correctness of simulations is not adversely affected.

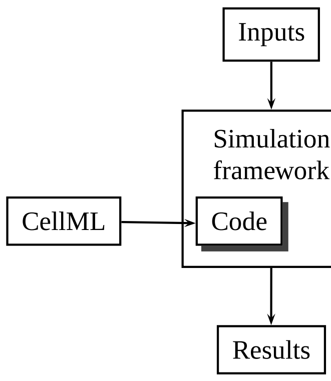
5 Partial Evaluation

CellML provides us with a portable and easily understood format for cell models, but as with computing in general this comes at the expense of efficiency. Again pursuing our analogy of treating CellML like a programming language, just as we were able to check certain properties of CellML models automatically in the previous chapter, there is also scope for automatically transforming CellML models to enable more efficient simulation.

In this chapter, we consider the use of partial evaluation (PE; Jones et al. 1993) for automatically optimising the simulation of cardiac ionic cell models. The technique is explained in general terms in Section 5.1. We have designed a partial evaluator specifically for CellML, which hence contains some unique features. This is described in Section 5.2.

Crucially, partial evaluation can also be placed on a solid theoretical basis, and the correctness of our partial evaluator is proven in Section 5.3. We conclude the chapter with some discussion in Section 5.4, which includes a comparison of two implementations of our partial evaluation techniques. Experimental results of applying this optimisation to a sample of models will be presented in Chapter 7.

Figure 5.1 CellML models are typically translated into computer code and compiled in order to be simulated within a simulation environment.



5.1 Introduction

Writing a separate program to simulate each type of cell model, coupled tightly to the simulation environment, and optimising by hand, can produce very efficient code, and this is thus the route that has traditionally been taken since computation time is at a premium. This has the disadvantages that much work is required for each new cell model, the simulation software becomes very hard to maintain, and confidence in the correctness of the simulations can be low.

On the other hand, if a simulation environment must interpret a large CellML file at every time step, this introduces a large computational overhead. For this reason it is common to convert the CellML model into code written in some programming language prior to simulation, as illustrated in Figure 5.1. This is a substantial improvement, but a good model will be written in an abstract style to enable easy comprehension by human readers, with many small components. A faithful translation will thus produce an inefficient program, just as there is a cost associated with a modular, high-level programming style. Often a natural ‘flattening’ transformation is applied,[1] which places all the mathematics within a single method, renaming variables to indicate the original ‘owning’ component. This will give some improvement, and one might suppose that using an optimising compiler would remove further inefficiencies. However there

[1]For example COR (Garny et al., 2003b) uses such a transformation; CESE (Missan and McDonald, 2005) does not.

are optimisations that we would like to apply that a compiler will not perform.

A compiler is designed to process many different programs, solving many different problems. Hence any optimisations it applies must necessarily be general, applicable to a large class of programs. Specific problems may be amenable to optimisations that do not apply to other problems, and we would not expect a general compiler to apply these. Also, a compiler will only apply optimisations after the CellML model has been converted into program code. As we shall see, there are advantages to performing optimisations earlier, directly transforming the CellML model. Notably, this allows such domain specific optimisations to interact.

In this chapter we discuss the class of optimisations known as staging transformations. Consider a system of ODEs, such as is found in a cell model. To simulate the model we need to solve the system, which involves a loop over time, with the 'right hand side' of the system being evaluated at least once at each time step. Some of the computations performed at each time step will be identical, and it would thus make sense to perform these computations only once, before the simulation is started—they are moved to an earlier stage of the whole process. Various simple examples are found in Listing 2.3, such as the division by the membrane capacitance *Cm* in **ode**(V, *time*), or the computations of the reversal potentials *E_Na*, *E_K* and *E_L*. This optimisation especially applies when the cell model is parameterised, for example to specify a version of the model, or some external conditions. Many of the parameters will not change during the course of the simulation, and so static computations depending only on fixed parameters should be performed only once. Since ODE solver loops are often large and complex, a compiler will generally not perform this sort of optimisation for us to the degree we desire. Also, if these transformations are left to the compiler, any other optimisations we may wish to perform must be done beforehand, not afterwards. We thus need to use a *partial evaluator* instead.

A partial evaluator is an automatic tool that pre-computes parts of a program known at compile time, producing a new *specialised* program. This specialised program will produce the same output as the original, when given equivalent[2] input. Deciding which computations only

[2]Inputs known at compile time may be given to the partial evaluator; these are then not given to the specialised

depend on the (incomplete) known data and can safely be performed early is undertaken by a *Binding Time Analysis* (BTA), which classifies each (sub-)expression as *static* or *dynamic*. The former are certain to be computable at compile time, whereas the latter may not be—the name comes from the fact that they may depend on variables which change dynamically at run time, such as state variables or the simulation time.

As a (trivial) example, consider this program fragment which computes integer powers:

```
power base exponent =
    if exponent == 0 then 1
                     else base * (power base (exponent - 1))
```

We can partially evaluate the function *square*:

```
square x = power x 2
         = x * (power x 1)
         = x * x * 1
```

A specialised program would thus include the expression $x*x*1$ in place of calls to *square x*; the former will execute faster than the latter.

There are other instances where the techniques of partial evaluation have been applied to scientific computations. Anderson has specialised ray tracers to a specific scene (Andersen, 1995). He noted a positive interaction between partial evaluation and an optimising compiler: program simplifications as a result of partial evaluation allowed the compiler to make further optimisations, for example algebraic simplifications and better register use. Partial evaluation has also been applied to programs solving the 9-body gravitational attraction problem (Berlin and Weise, 1990; Berlin, 1990) with considerable success—speed increases of between 7 and 90 times have been reported. Finally work has been done on using partial evaluation to accelerate the simulation of digital circuits (Weise and Seligman, 1992); this shows the use of partial evaluation in an object-oriented context. We have not found any instances where the techniques have been applied to cardiac modelling however, so this is a new application area.

We show how partial evaluation can be applied to CellML models, producing a new model in which all possible computation has been performed, but that will give the same results as the

program.

original model when simulated. The rules governing this transformation are presented in the next section.

5.2 Partial evaluation of CellML

We actually have two implementations of our partial evaluator for CellML. One, written in Python[3] produces a new CellML document, and for pragmatic reasons is the one used for applying the optimisations (see also Section 5.4 and Cooper and McKeever, 2007). The other, written in Haskell, we present here in order both to express the ideas clearly and to prove the correctness of the optimisation technique. Its main output is an environment representing a model, as described in Section 3.2, rather than a CellML document (although it is straightforward to generate the latter from the former). Key portions of the Haskell code are included within the text, and the full code can be found in Appendix E.

The input to the partial evaluator is primarily a CellML model. Automatic analysis of the model (see Section 2.3.2) can determine which variables are state or free variables (and hence dynamic) and which are parameters (assumed to be static). However, the partial evaluator also takes an environment mapping variables to values, in order to specify those parameters as explicitly being dynamic. This serves two purposes. Firstly, scientists may wish to observe how the values of certain computed variables vary during the course of a simulation; these variables must therefore be retained in the specialised model. Secondly, scientists may wish to modify the values of some parameters manually during the course of a simulation, for instance to model the introduction of a drug; this is known as computational steering.

A unique feature of our partial evaluator is its support for physical units. Every quantity in the specialised model will be annotated with the units of the expression from which it was computed, and appropriate definitions of these units will be added to the units environment of the model. Thus if the input model is dimensionally consistent, the specialised model will be also. This feature is discussed further in Section 5.2.3.

[3]`https://chaste.ediamond.ox.ac.uk/cellml/`

Partial evaluation is conceptually performed in two phases: binding time analysis (Section 5.2.1) followed by evaluation and reduction of expressions (Section 5.2.2). The former determines whether each node in each expression tree in the model is static or dynamic. The latter evaluates static expressions to obtain their value, and *reduces* dynamic expressions by evaluating static subexpressions. Section 5.2.4 discusses how the phases are combined to partially evaluate a model.

We make two key assumptions about the input CellML model:

1. the dependency graph of the equations is acyclic; and

2. the model is 'valid'.

Both of these assumptions may be checked automatically. In other words, we shift error checking to a pre-processing stage so it does not obscure the key ideas. The second of these assumptions means that the mathematics does yield a boolean value where a boolean is expected, etc. The first ensures that our recursions will terminate. For example, BTA is simply a post-order traversal of the expression tree, processing dependencies by looking up definitions in the environment, hence could only fail to terminate if there was a cycle.[4]

5.2.1 Binding time analysis

BTA is performed primarily by the *bta* function, which analyses a single expression. There is a related function, *bta_key*, which analyses the definition of an identifier within the model environment, using *bta* if the definition is an expression; it is in turn used by *bta* to analyse variable and ODE lookups. The outcome of this analysis is a *partition* of the model environment into static and dynamic portions:

```
data BindingTime = Static | Dynamic

partition :: Env → (Env, Env)
partition env = foldr f (empty_env, empty_env) (names env)
```

[4]It could also fail if the function was not defined for all terms in the abstract data type; our implementation of BTA is total, however.

```
  where
    f :: EnvKey → (Env, Env) → (Env, Env)
    f k (envs, envd) =
      case bt of
        Static  → (define envs k v, envd)
        Dynamic → (envs, define envd k v)
      where
        v = find env k
        bt = bta_key env k
```

The *bta* function is straightforward, with most of the work delegated to helper functions, discussed below. The only cases dealt with directly are for constants, which are static.

```
bta :: Env → MathTree → BindingTime
bta env (Num _ _) = Static -- constants are always static
bta env (Bool _)  = Static -- constants are always static
bta env (Variable v) -- look up the definition and analyse that
  = bta_key env (Var v)
bta env (Diff v1 v2) -- analyse the ODE definition
  = bta_key env (Ode v1 v2)
bta env (Apply operator operands)
  = bta_apply env operator operands
bta env (Piecewise cases Nothing)
  = bta_piecewise env cases
```

The main feature of the *bta_key* helper function is that it checks to see whether variables have been annotated as dynamic, either automatically (in the case of state and free variables) or explicitly by the user. It does so by testing whether they are defined in a 'dynamic environment' *dyn_env* included as part of the internal data[5] within the environment.

```
bta_key :: Env → EnvKey → BindingTime
bta_key env (Var "") = Static -- arbitrary choice
bta_key env key
  = case maybe_find dyn_env key of
      Just _  → Dynamic -- state or free variable, or user annotated
      Nothing → case find env key of
                  Expr t → bta env t -- analyse the defining expression
                  Val _  → Static    -- constants are static
  where (InternalData (_,_,dyn_env)) = find env (Var "")
```

The binding time of an operator application is, in general, the maximum of the binding times of the operands (the ordering on the *BindingTime* type means that *Static* $<$ *Dynamic*). If any operand is dynamic, we do not (in most cases) have enough information to evaluate the whole expression, and thus it too must be dynamic. There are exceptions to this, however. Our partial

[5]See also Section 3.2.

evaluator is partially online, which means that it evaluates certain static expressions during the binding time analysis phase in order to determine a better partition, with more expressions annotated as static. The subsidiary function *bta_short_circuit* is used to short-circuit the analysis of operators such as *And* and *Or*. It is applied to the list of operands, and analyses only as many as are required to determine a binding time for the whole expression. This is done by evaluating static operands (we show that this is safe in Theorem 5.2), checking with the supplied predicate whether evaluation of the whole expression would short-circuit at this point. If a dynamic operand is encountered, then it cannot be evaluated and so the whole expression is dynamic.

```
bta_apply :: Env → Operator → [MathTree] → BindingTime
bta_apply env And operands -- short-circuit if static operand is False
  = bta_short_circuit env (not . get_bool) operands
bta_apply env Or operands  -- short-circuit if static operand is True
  = bta_short_circuit env get_bool operands
bta_apply env _ operands   -- general case
  = maximum (map (bta env) operands)

bta_short_circuit :: Env → (Value → Bool) → [MathTree] → BindingTime
bta_short_circuit env pred (t:ts)
  = if bta env t == Static
      then if pred (eval env t) then Static
                                else bta_short_circuit env pred ts
      else Dynamic
bta_short_circuit env pred [] = Static
```

A similar short-circuiting is applied to piecewise expressions, with evaluation of initial static conditions. If such a condition is false, we can ignore that case; if it is true, then that case is the only one we need consider.

```
bta_piecewise :: Env → [Case] → BindingTime
bta_piecewise env (Case cond res : cs)
  = if bta env cond == Static
      then if get_bool (eval env cond) then bta env res
                                       else bta_piecewise env cs
      else Dynamic
bta_piecewise env [] = Static
```

5.2.2 Reduction of dynamic expressions

The workhorse of the partial evaluator is the *reduce* function, which performs PE on a single expression, evaluating static subexpressions within the static portion of the model environment and replacing them by constants (using the subsidiary function *eval_to_expr*). We first show its definition, then explain the key features.

```
reduce :: Env → MathTree → MathTree
reduce env expr
  = case bta env expr of
      Static  → let (_,e) = eval_to_expr expr in e
      Dynamic → reduce' expr
  where
    -- Determine expression binding times
    (env_s, env_d) = partition env

    -- Evaluate a static expression to get a constant expression
    eval_to_expr :: MathTree → (Value, MathTree)
    eval_to_expr e
      = let v = eval env_s expr in
          (v, case v of
                Number n  → Num n (Right (eval_units_in env expr))
                Boolean b → Bool b
          )

    -- Reduce an expression known to be dynamic
    reduce' (Variable var)
      = reduce_lookup (Var var) (Variable var)
    reduce' (Diff v1 v2)
      = reduce_lookup (Ode v1 v2) (Diff v1 v2)
    reduce' (Piecewise cases Nothing) -- short-circuit static conditions
      = f cases
      where
        f allcs@(Case cond res : cs)
          = if bta env cond == Static
              then case eval env_s cond of
                     Boolean True  → reduce env res
                     Boolean False → f cs
              else Piecewise (map (rcase env) allcs) Nothing
        rcase env (Case cond res) = Case cond' res'
          where cond' = reduce env cond
                res'  = reduce env res
    reduce' (Apply And operands) -- short-circuit if static operand False
      = short_circuit And (not . get_bool) operands
    reduce' (Apply Or operands)  -- short-circuit if static operand True
      = short_circuit Or get_bool operands
    reduce' (Apply Divide [n, d]) -- convert divide-by-static to times
      = if bta env d == Static
          then reduce env (Apply Times [n, Apply Divide [one, d]])
          else Apply Divide (reduce_list [n, d])
      where one = Num 1 (Left (full_ident ".model" "dimensionless"))
    reduce' (Apply op operands) -- reduce operands
      = Apply op (reduce_list operands)

    reduce_list = map (reduce env)
```

```
short_circuit :: Operator → (Value → Bool) → [MathTree] → MathTree
short_circuit op pred (t:ts)
  = if bta env t == Static
      then if pred val then e -- never happens as expr is dynamic
                       else short_circuit op pred ts
      else Apply op (reduce_list (t:ts))
  where (val, e) = eval_to_expr t

reduce_lookup key key_as_expr
  = if may_instantiate_key env key
    then reduce env e -- instantiate reduced definition
    else key_as_expr  -- retain lookup
  where Expr e = find env_d key
```

Several aspects of this function deserve further discussion. Firstly, environment lookups are handled by the *reduce_lookup* function. This uses the function *may_instantiate_key* to determine whether to instantiate the (reduced) definition of the variable or ODE in place of the lookup, or whether to leave the lookup in place. This is used to prevent code duplication, and will be examined further in Theorem 5.5. We only note that the definition will be an *Expr*, since if not then the expression would be static.

Parallelling *bta_short_circuit* there is another subsidiary function *short_circuit* defined within *reduce*, which allows us to discard unneeded operands where short-circuiting is possible, and yet the expression as a whole is dynamic. Any initial static operands are discarded (since they must evaluate to give **False** under the predicate) and the remaining operands reduced.

Short-circuiting also takes place in the reduction of piecewise expressions, with initial static conditions being evaluated much as was done during BTA. Note that since the whole expression is dynamic, there must be at least one condition that is either dynamic or evaluates to **True**. The *rcase* helper function reduces any remaining cases by reducing the condition and result separately.

Our partial evaluator performs a further optimisation by converting divisions where the denominator is static into multiplications, which can be evaluated more efficiently at run-time. A compile-time computation is performed to determine the reciprocal of the denominator, and a run-time multiplication of the numerator with this reciprocal is generated.

We also define a related function *reduce_key* which reduces the definition of an identifier. There is an important difference in the type of this function, however: whereas *reduce* produces a new expression, *reduce_key* wraps either an expression or a plain value in an *EnvValue*. The use of this will be seen in Section 5.2.4.

```
reduce_key :: Env → EnvKey → EnvValue
reduce_key env k
  = case find env k of
      Expr t → Expr (reduce env t)
      value  → value
```

5.2.3 Units and partial evaluation

It was seen in the definition of *reduce* that whenever a static expression was evaluated to a number, the number was also annotated with the units definition associated with the original expression, in the line

```
    Number n   → Num n (Right (eval_units_in env expr))
```

The function *eval_units_in* uses the algorithms of Chapter 4 to determine the units of the given expression. In doing so it makes use of two environments contained in the *InternalData* within *env*. The first, of type *VarUnitsEnv* = *Environment Ident* (*UName*, *Units*) maps (the full name of) each variable to the units it is explicitly given in the CellML model. The second, of type *UnitsEnvs* = *Environment Ident UnitsEnv* = *Environment Ident* (*Environment UName Units*) stores the units defined within the model, grouped according to the component in which they were defined. This latter is used to determine the units associated with constant expressions, using the *lookup_units* function to search first definitions in the local component, then those at the whole model scope, then the built-in definitions.

```
lookup_units :: UnitsEnvs → Ident → UName → Units
lookup_units envs cname uname
  = case maybe_find (component_units envs cname) uname of
        Just u   → u
        Nothing → case maybe_find (model_units envs) uname of
                        Just u   → u
                        Nothing → find standard_units uname

model_units :: UnitsEnvs → UnitsEnv
model_units uenvs = find uenvs ".model"
```

```
component_units :: UnitsEnvs → Ident → UnitsEnv
component_units uenvs cname
  = if cname == "" then empty_env
    else case maybe_find uenvs cname of
      Just env → env
      Nothing  → error (show cname ++ " is not a component")
```

This is sufficient to associate the proper units *definition* with the constant, but the units environment has been left unchanged. However, a CellML model cannot contain a units definition inline within the mathematics, but must refer by name to units defined elsewhere in the file. This detail is taken care of later by the *define_pe_units* function, after all expressions have been completely reduced, so that reduction of expressions may be performed locally without needing to modify the environment. We thus see why an **Either** type was used for units references: in a model, either before or after PE has been performed, such references must be to unit *names*, but during PE it is convenient to refer anonymously to unit *definitions*.

The *define_pe_units* performs a kind of fold over the environment representing a model, processing each expression tree defined within the environment using the *def_units_expr* function. This in turn recurses through the expression examining each units reference, and where reference is made to anonymous units the function *define_units* is used to obtain a name instead, and a (possibly updated) units environment. The *define_units* function uses equality on the *Units* type to determine if a definition is already present in the environment. Where it is, the name to which that definition is bound is used; where not present, the definition is added to the environment (in that portion for definitions global to the model) with a unique name of the form "$\_\_\_i$", where i is the first integer such that the resulting name is not already used.

We give here code for three of the key functions; full details are in Appendix E.

```
define_pe_units :: Env → Env
define_pe_units env
  = foldr_expr_key f env env
  where
    f :: MathTree → EnvKey → Env → Env
    f expr key env' = define env'' (Var "") idata'
     where
      (InternalData (vuenv, uenvs, dyn_env)) = find env' (Var "")
      idata' = InternalData (vuenv, uenvs', dyn_env)
```

```
        env'' = define env' key (Expr expr')
        (uenvs', expr') = def_units_expr uenvs expr

def_units_expr :: UnitsEnvs → MathTree → (UnitsEnvs, MathTree)
def_units_expr uenvs (Num x uref)
  = case uref of
      Left uname  → (uenvs, Num x uref)
      Right units → (uenvs', Num x (Left uname'))
       where
        (uname', uenvs') = define_units uenvs units
def_units_expr uenvs (Apply op operands)
  = (uenvs', Apply op operands')
  where (uenvs', operands') = map_foldr def_units_expr uenvs operands
def_units_expr uenvs (Piecewise cases Nothing)
  = (uenvs', Piecewise cases' Nothing)
  where (uenvs', ts) = map_foldr def_units_expr uenvs (cases2list cases)
        cases' = list2cases ts
def_units_expr uenvs leaf = (uenvs, leaf)

define_units :: UnitsEnvs → Units → (UName, UnitsEnvs)
define_units uenvs units
  = case units_defined uenvs units of
      Just uname → (uname, uenvs)
      Nothing    → (uniq_uname,
                     define uenvs ".model"
                            (define menv uniq_uname units))
  where menv = model_units uenvs
        uniq_uname = uniq_key menv
```

There is one other units-related transformation needed when performing PE. Partial evaluation changes the component structure of the model, moving all mathematics to be situated within a single component, and removing all the original components. Any units definitions within these components must thus also be moved. This is accomplished by the *move_component_units* function, which is applied as a post-processing step on the model environment after PE. It alters all units definitions to be global to the model, and updates name references within mathematics to reflect this.

Note that after PE, all named units references to component-level definitions use a full name of the form *full_ident component_name units_name*, so we can use that name in defining these units within the model-global environment, and there are unlikely to be naming conflicts with units already defined there. We still include a test for naming conflicts, however, to ensure they don't occur. If there is a conflict, we iterate adding '_' to the end of the name until there is no longer a conflict.

This may result in duplicate definitions, in the sense that the same units are defined multiple times but with different names; however this will only occur if it was also the case in the original model, which is reasonable.

```
move_component_units :: Env → Env
move_component_units env
  = modify_exprs (modify_leaves rename_urefs) env'
  where
    InternalData (vuenv, uenvs, dyn_env) = find env (Var "")
    idata' = InternalData (vuenv, new_uenvs, dyn_env)
    env' = define env (Var "") idata'
    new_uenvs = define standard_uenvs ".model" new_model_env
    (new_model_env, renamed)
        = foldr_env do_comp (model_units uenvs, empty_env) uenvs

    rename_urefs :: MathTree → MathTree
    rename_urefs (Num n (Left uname))
        = case maybe_find renamed uname of
            Just new_name → Num n (Left new_name)
            Nothing       → Num n (Left uname)
    rename_urefs leaf = leaf

    do_comp :: Ident → UnitsEnv → (UnitsEnv, RenameEnv)
            → (UnitsEnv, RenameEnv)
    do_comp cname c_uenv (uenv, renames)
      = if head cname == '.'
        then (uenv, renames)
        else foldr_env (do_udef cname) (uenv, renames) c_uenv
    do_udef :: Ident → UName → Units → (UnitsEnv, RenameEnv)
            → (UnitsEnv, RenameEnv)
    do_udef cname uname units (uenv, renames)
      = if new_name == full_name
        then (new_env, renames)
        else (new_env, define renames full_name new_name)
      where full_name = full_ident cname uname
            new_name = (head . dropWhile used)
                       (iterate (++"_") full_name)
            used name = isJust (maybe_find uenv name)
            new_env = define uenv new_name units
```

5.2.4 Applying PE to a model

In Chapter 3, simulation of a model was defined in terms of the function *run_env*, which acts on an environment representing the model, rather than on the *Model* type directly. It also requires an environment containing values for the state variables and time, in order to evaluate each ODE at that input point in the domain of the model function $\boldsymbol{f}$. Similarly therefore, partial evaluation of a model is defined in terms of the function *reduce_env*, which applies PE to each expression in the model environment, generating a new environment which may be passed to *run_env*.

As with *run_env*, *reduce_env* takes as input two environments and a list of keys, this last specifying the ODEs which compose $\boldsymbol{f}$. The first environment is the model environment, just as would be passed to *run_env*. The second environment contains values for any dynamic variables, and is thus known as *dyn_env*. It defines the state variables $\boldsymbol{Y}$ and time t, as for *run_env*, but may also contain definitions of other variables which the user has explicitly annotated as dynamic. These definitions are considered to replace those given in the model unless they are given by the special value *DynamicMarker*, so that users may also use this environment to alter parameter values at run time.

```
reduce_env :: Env → [EnvKey] → Env → Env
reduce_env model_env derivs dyn_env
  = (move_component_units . define_pe_units)
          (rec_reduce_derivs model_env)
  where idata = find model_env (Var "")
        reduce_derivs env
          = foldr (reduce_deriv env) init_env derivs
        init_env = filter_env real_value dyn_env
          where real_value _ (Val DynamicMarker) = False
                real_value _ _ = True
        rec_reduce_derivs env
          = if has_instantiable_key new_env
               then rec_reduce_derivs new_env
               else new_env
          where new_env = define
                   (head (dropWhile has_undefined_var
                          (iterate (add_reduced_definitions env)
                                   (reduce_derivs env))))
                   (Var "") idata

        reduce_deriv :: Env → EnvKey → Env → Env
        reduce_deriv menv d env = define env d (reduce_key menv d)
```

There are several auxiliary functions defined here. The most important is *reduce_deriv*, which uses *reduce_key* (and hence *reduce*; see Section 5.2.2) to apply PE to the definition of a single ODE. The function *reduce_derivs* uses this to add reduced definitions of each ODE to the dynamic environment.

If we were not concerned with code duplication—if the definitions of variables or ODEs were always instantiated at the point of lookup and reduced—this would be all that is required. Indeed in the proof of Theorem 5.4 we assume that this is the case. Our full implementation shown here, however, does *not* duplicate code, and this is proved in Theorem 5.5. In the previous section we noted the function *may_instantiate_key* which restricts when definitions may be instantiated. The other part of the solution is *rec_reduce_derivs*, which repeats PE if there are any references remaining which may be instantiated (*has_instantiable_key* uses *may_instantiate_key*), any ensures that any expressions which are used but not instantiated at points of use are still reduced and defined within the model environment. Definitions of the functions involved will be seen in the proof of Theorem 5.5.

Finally we note that while conceptually there are two phases to partial evaluation, in our implementation these are not separated. The *bta* function is called whenever a binding time is required by *reduce*, and *partition* is also called by *reduce* multiple times. This is clearly inefficient, but does not affect correctness. The approach was chosen for the sake of simplicity, to avoid passing binding time information separately to the *reduce* function, or defining a different type for expressions annotated with a binding time.

5.3 Proof of correctness

Consider again the ODE system model given in Equation (3.0.1):

$$\frac{\mathrm{d}\boldsymbol{Y}}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{Y}, t).$$

For the correctness proof, we wish to show that for any ODE solver, simulation of the partially evaluated model will produce the same results as simulating the original model, when given the same inputs (initial conditions and simulation duration), i.e.

$$\begin{aligned}&\forall\ i \in \mathit{inputs},\ c \in \mathit{CellMLmodels}\\ &\quad \mathit{solver}(c, i) = \mathit{solver}(\mathcal{PE}\ c, i).\end{aligned}$$

As we discussed in Chapter 3, since it is impossible to analyse every potential algorithm for solving ODEs directly, we instead note that all solvers must interact with the model through evaluations of the function $\boldsymbol{f}$. Hence if we can show that evaluation of $\boldsymbol{f}$ is unchanged under partial evaluation, then the desired result follows. To use the terminology of our interpreter and partial evaluator developed here, suppose that env_f is the model environment representing the function $\boldsymbol{f}$, and let *derivs* be the list of *EnvKey*s representing the derivatives in $\mathrm{d}\boldsymbol{Y}/\mathrm{d}t$. We then need to prove that for all models of the form (3.0.1) and all input environments *inputs*,

$$\begin{gathered} \mathit{run\_env}\ \mathit{env}_f\ \mathit{derivs}\ \mathit{inputs} = \\ \mathit{run\_env}\ (\mathit{reduce\_env}\ \mathit{env}_f\ \mathit{derivs}\ \mathit{inputs})\ \mathit{derivs}\ \mathit{inputs} \end{gathered}$$

We start by showing that the BTA produces a ‘suitable’ division of the environment representing a model into static and dynamic parts, such that evaluating a static expression within only the static environment is the same as evaluation within the whole environment (Theorem 5.2). We then turn to the partial evaluation itself, and proceed in four stages.

Theorem 5.3 Having shown that evaluation of static expressions by the partial evaluator is ‘safe’, in that evaluation will succeed if evaluation in the full model would succeed, we can then show that evaluation of a reduced expression, in which all static portions have been already evaluated, is the same as evaluating the original expression within the same environment. This shows that partial evaluation of a single expression does not change its meaning.

Theorem 5.4 We then show that, when using a somewhat simplified core of the partial evaluator, partial evaluation of a whole model does not change its meaning. This proves the correctness of the core, and the two remaining theorems deal with the extra features present in the full implementation.

Theorem 5.5 The core implementation considered in Theorem 5.4 duplicates code—it always instantiates the definition of a variable or ODE at the point of lookup. It only requires minor changes to correct this, however. The first is to instantiate defining expressions only if the lookup is performed in precisely one location within the model. The second is to repeat PE on the reduced model if it contains any definitions that are used only once, and ensure that definitions which are used *more* than once are included (reduced) within the reduced model. We show that these modifications both maintain correctness and ensure no code is duplicated.

Theorem 5.6 Finally, we show that appropriate units definitions are added to the reduced model, and that all quantities in the model have a reference to the appropriate units definition.

In this chapter, some of the proofs are only given in outline form, in order to put across the main ideas without detracting from the flow of the text. Further details are included in Appendix B. For brevity, we also frequently ignore the fact that CellML *Value*s are not Haskell values, and do not write in the (un)wrapping functions. For instance,

```
apply And operands = Boolean (and (map get_bool operands)),
```

but in the proof here we would write

```
apply And operands = and operands.
```

The proof mixes Haskell and mathematical notation to some extent.

- $k \in env$, where $k :: EnvKey$, means that the key k is defined in the environment *env* and maps to some unspecified value.
- $env_s \oplus env_d$ is a sum operation on environments. The resulting environment contains all the definitions in both env_s and env_d. For the sake of definiteness, if env_s and env_d both define the same key, the value from env_s is chosen.
- We also talk of an expression *expr* being within an environment *env*. This means that $\exists k \in env$ such that $find\ env\ k = Expr\ t$ and *expr* is a subtree of t (possibly the whole tree).

We begin the proof with a lemma stating some basic properties of the partition produced by the binding time analysis.

Lemma 5.1 The binding time analysis of a CellML model produces a partition of the environment into static and dynamic portions. Formally, if $(env_s, env_d) = partition\ env$,

(i) $\forall k \in env_s, find\ env_s\ k = find\ env\ k$;

(ii) $\forall k \in env_d, find\ env_d\ k = find\ env\ k$;

(iii) $bta\_key\ env\ k = \mathrm{S} \Leftrightarrow k \in env_s$;

(iv) $bta\_key\ env\ k = \mathrm{D} \Leftrightarrow k \in env_d$;

(v) $\forall k \in env$, either $k \in env_s$ or $k \in env_d$ (but not both).

Proof. All of these follow straightforwardly from the definition of *partition*. □

Theorem 5.2 Evaluation of a static expression can be performed solely within the static portion of the model environment. Formally, for any environment *env* and expression *expr* within it,

$$bta\ env\ expr = S \Rightarrow eval\ env_s\ expr = eval\ (env_s \oplus env_d)\ expr$$

where

$$(env_s, env_d) = partition\ env.$$

Note that by Lemma 5.1 $env = env_s \oplus env_d$.

Proof. The proof proceeds by structural induction on the form of *expr*, and details are given in Appendix B.1. The key requirement is to show that variable or ODE lookups can always be satisfied by the static portion of the environment. There is also some work required to handle the cases where short-circuiting is employed. □

Theorem 5.3 Partial evaluation of a single expression does not change its meaning: evaluation of a reduced expression is the same as evaluating the original, within the same environment. Formally, for any environment *env* and expression *expr* within it,

$$eval\ env\ expr = eval\ env\ (reduce\ env\ expr).$$

Proof. We first consider the case of a static expression. This is evaluated by *reduce* within the static portion of the environment, to ensure that all required variables are defined:

$$
\begin{array}{ll}
 & \mathit{eval}\ \mathit{env}\ (\mathit{reduce}\ \mathit{env}\ \mathit{expr}) \\
= & \quad \{\text{definition of } \mathit{reduce}\} \\
 & \mathit{eval}\ \mathit{env}\ (\mathit{wrap}\ (\mathit{eval}\ \mathit{env}_s\ \mathit{expr})) \\
= & \quad \{\text{Theorem 5.2}\} \\
 & \mathit{eval}\ \mathit{env}\ (\mathit{wrap}\ (\mathit{eval}\ \mathit{env}\ \mathit{expr})) \\
= & \quad \{\text{evaluation of a constant}\} \\
 & \mathit{eval}\ \mathit{env}\ \mathit{expr}
\end{array}
$$

The *wrap* function simply wraps a *Value* into a constant expression (*Num* or **Bool**), and hence *eval* just performs unwrapping. In the actual code, the behaviour of *wrap* is implemented by the *eval_to_expr* function.

For dynamic expressions we proceed by structural induction on the form of *expr*. Note that since constants are always static and *expr* is dynamic, there are only four cases to consider. In each case we assume that $(\mathit{env}_s, \mathit{env}_d) = \mathit{partition}\ \mathit{env}$.

i. Case $\mathit{expr} = \mathit{Variable}\ v$:

   This case is handled by the *reduce_lookup* function. The behaviour differs depending on the result of *may_instantiate_key*. For the present, we consider a simplified version where the result is always **True** unless the key was provided in the input *dyn_env*, i.e. it is a state variable, time variable, or explicitly annotated as dynamic by the user.

   If *may_instantiate_key* gives **False**, *reduce* leaves *expr* unchanged and the result is trivial. Otherwise, by Lemma 5.1(iv) $\mathit{Var}\ v \in \mathit{env}_d$, and by the definition of *bta_key* and Lemma 5.1(ii) we also have that $\mathit{find}\ \mathit{env}_d\ (\mathit{Var}\ v) = \mathit{Expr}\ e$, where e is dynamic. Thus,

$$
\begin{array}{ll}
 & \mathit{eval}\ \mathit{env}\ (\mathit{reduce}\ \mathit{env}\ \mathit{expr}) \\
= & \quad \{\text{definition of } \mathit{reduce}\} \\
 & \mathit{eval}\ \mathit{env}\ (\mathit{reduce}\ \mathit{env}\ e) \\
= & \quad \{\text{IH on } e\} \\
 & \mathit{eval}\ \mathit{env}\ e \\
= & \quad \{\text{Lemma 5.1(ii) and definition of } \mathit{elookup}\} \\
 & \mathit{elookup}\ \mathit{env}\ (\mathit{Var}\ v) \\
= & \quad \{\text{definition of } \mathit{eval}\} \\
 & \mathit{eval}\ \mathit{env}\ \mathit{expr}
\end{array}
$$

ii. Case *expr* = *Ode* v_1 v_2:

   Since this case is also handled by *reduce_lookup*, the argument is identical to that above.

iii. Case *expr* = *Apply operator operands*:

iv. Case *expr* = *Piecewise cases* **Nothing**:

   The details for these cases are given in Appendix B.2. The short-circuiting requires some fairly lengthy analysis, but it is not difficult. □

Theorem 5.4 Partial evaluation of a model does not change its meaning. As mentioned at the start of Section 5.3, this involves showing that evaluation of $\boldsymbol{f}(\boldsymbol{Y}, t)$ is unchanged for any possible values of $\boldsymbol{Y}$ and t. Formally, for any CellML model *model* and environment *initenv* associating values with $\boldsymbol{Y}$ and t,

$$\textit{run\_cellml model initenv} = \textit{reduce\_and\_run\_cellml model initenv}.$$

Proof. The only difference between the two sides in this equality is the environment which is used to run the model in the function *run_env*—*run_cellml* uses that given by *load_cellml* directly, whereas *reduce_and_run_cellml* applies partial evaluation to it first using *reduce_env*.

Here we prove the theorem for the simpler case where code duplication is not considered, and defer consideration of the full partial evaluator for Theorem 5.5. Thus, as was seen in the proof of Theorem 5.3, all environment lookups have the defining expression instantiated unless a definition is given in the input environment *initenv*. Also, the definition of *reduce_env* is simpler:

```
reduce_env :: Env → [EnvKey] → Env → Env
reduce_env model_env derivs initenv
  = define_pe_units (reduce_derivs model_env)
  where idata = find model_env (Var "")
        reduce_derivs env
            = define (foldr (reduce_deriv env) initenv derivs)
                     (Var "") idata
        reduce_deriv menv d env
            = define env d (reduce_key menv d)
```

Recall that *run_env* is defined as

```
run_env model_env derivs
  = foldr eval_deriv empty_env derivs
  where eval_deriv d env
          = define env d (Val (elookup model_env d))
```

We need to prove that

$$\begin{gathered}\textit{run\_env model\_env derivs initenv} = \\ \textit{run\_env}\ (\textit{reduce\_env model\_env derivs initenv})\ \textit{derivs initenv}\end{gathered}$$

where *derivs* is the list of *EnvKey*s representing the derivatives in the model, and *model_env* is the environment generated from *model*.

Since both *run_env* and *reduce_env* are essentially folds processing each ODE in turn, the result follows by repeated application of Theorem 5.3. We can't apply it directly, since evaluation of each reduced ODE takes place in the reduced model environment, and Theorem 5.3 requires evaluation of the reduced expression to take place in the same environment as evaluation of the whole expression. However, on examining the proof we see that the environment used for evaluation only matters when environment lookups are performed. But using the definition of *reduce_env* given here, the only lookups remaining in a reduced expression are satisfied by *initenv*, and thus have the same definition in both *model_env* and the reduced environment. □

Theorem 5.5 When using the full PE implementation, there is no duplication of code in the reduced model. That is, if the original model contains a binding of some dynamic variable (say x) to an expression (say *Expr e*, where *e* is not an environment lookup, i.e. of the form *Variable v* or *Diff* v_1 v_2), and x is used in more than one location in the reduced model, then the reduced model will contain a binding of x to a reduced *Expr e*, and lookups of x will be retained in the reduced model, rather than its definition being instantiated.

While we refer to x as a variable in the statement above, this applies equally if x is a derivative.

Proof. There are two facets to this proof. The first is to show that definitions will not be instantiated if this would lead to code duplication, which is the purpose of the *may_instantiate_key* function. The second is to show that where a lookup is retained, an appropriate definition is also present.

The definition of a key may be instantiated if either (a) this is the only lookup of the key in the whole model; or (b) the key's definition means it is just an alias for another key. An example of this latter case is the key being defined by *Expr* (*Variable v*). There is also another criterion used by *may_instantiate_key*: the key must not be explicitly marked as dynamic by the user (i.e. must not be provided in the input environment; this includes the case where it is a state variable or time); this is tangential to a discussion of code duplication, however.

```
may_instantiate_key :: Env → EnvKey → Bool
may_instantiate_key env key
  = not annotated && (used_once || alias)
  where
    InternalData (_, _, dyn_env) = find env (Var "")
    annotated = isJust (maybe_find dyn_env key)
    used_once = check_usage env key == 1
    alias  = case find env key of
      Expr (Variable _) → True
      Expr (Diff _ _)   → True
      _                 → False
```

The *check_usage* function simply counts how many times the key is looked up in expressions within the given environment. With these definitions it is easy to see that expressions are never duplicated. We thus turn our attention to showing that the reduced model environment contains all the necessary definitions.

The key part of the full *reduce_env* implementation is the function *rec_reduce_derivs*, which we reproduce here for ease of reference.

```
rec_reduce_derivs env
  = if has_instantiable_key new_env
      then rec_reduce_derivs new_env
      else new_env
  where new_env = define
          (head (dropWhile has_undefined_var
                    (iterate (add_reduced_definitions env)
                             (reduce_derivs env))))
          (Var "") idata
```

```
has_instantiable_key env
  = foldr (||) False (map (may_instantiate_key env) (lookups env))
```

```
has_undefined_var env
  = not (used_set `Set.isSubsetOf` def_set)
  where def_set  = Set.fromList (names env)
        used_set = Set.fromList (lookups env)

add_reduced_definitions from_env to_env
  = foldr add_def to_env (Set.toList (Set.difference used_set def_set))
  where def_set  = Set.fromList (names to_env)
        used_set = Set.fromList (lookups to_env)
        add_def key env = define env key (reduce_key from_env key)
```

The function *reduce_derivs* is used to perform a single run of the PE algorithm, reducing each ODE in the model, and then *add_reduced_definitions* is iterated to ensure that any environment lookups are defined in the reduced model, and their definitions reduced. PE is repeated while the *has_instantiable_key* test (which uses *may_instantiate_key*) holds. This is because reduction of an expression may remove environment lookups, for instance if they occur in a subexpression that we can show statically will never be evaluated, and thus may reduce the number of lookups of a particular key to one, hence allowing an instantiation that was previously considered impossible.

In order to show that the reduced model environment includes all the required definitions, we therefore need to prove the following three claims.

i. Iteration of *add_reduced_definitions* terminates.

ii. Recursion of *rec_reduce_derivs* terminates.

iii. Evaluation of ODEs in the reduced model is equivalent to evaluation in the original model.

The first two points show how progress is made towards completion, and prove that the partial evaluation process will not fail to terminate. The last point proves that the model semantics are preserved.

i. Iteration of *add_reduced_definitions* terminates.

   Let env_i be the i^{th} member of

$$\textbf{iterate}\ (add\_reduced\_definitions\ env)\ (reduce\_derivs\ env)$$

where *env* is the environment passed to *rec_reduce_derivs*. Since we are assuming the model is valid, all lookups from *env* must be defined in *env*, and hence all lookups from any env_i must be defined in *env* (this can be shown by a simple induction, using this paragraph to perform the inductive step). The iteration continues until some env_i has no undefined variables. By properties of **iterate**, env_{i+1} will include all the definitions in env_i. If env_i does have an undefined variable, then by properties of sets a definition not present in env_i will be copied (reduced) from *env* to env_{i+1}. Hence the size of the env_i increases monotonically, and is bounded above by the size of *env*, so the iteration must terminate.

ii. Recursion of *rec_reduce_derivs* terminates.

We show that the number of lookups in the environment at each recursive call is strictly decreasing; since the number cannot go below zero the recursion must thus terminate. As we have shown above, expressions are never duplicated, and the total number of expression tree nodes in the environment cannot increase from one recursive call to the next, hence the number of lookups cannot increase. Also, whenever a recursive call is made, *has_instantiable_key new_env* must be **True**, hence *reduce* will instantiate at least one defining expression, thus reducing the number of lookups by at least one.

iii. Evaluation of ODEs in the reduced model is equivalent to evaluation in the original model.

This requires us to prove that for any expression *expr* within the original model environment env_m, with corresponding reduced environment env_r,

$$eval\ env_m\ expr = eval\ env_r\ (reduce\ env_m\ expr).$$

For the most part the proof proceeds as for Theorem 5.3, since the environments used for evaluation and reduction only matter when lookups are performed. Hence we need only consider the cases for *Variable v* and *Ode* v_1 v_2. We will prove the *Variable v* case; that for ODEs is essentially identical.

Therefore, suppose that $expr = Variable\ v$, and let $k = Var\ v$. If k is static the result is trivial, since then

$$reduce\ env_m\ expr = wrap\ (eval\ env_m\ expr)$$

and evaluation of a constant does not depend on the environment. Assume therefore that k is dynamic. Since the model is valid, $k \in env_m$, and it must be bound either to a run-time parameter or a dynamic expression. In the former case the result is also trivial, since then *reduce* leaves *expr* unchanged and k is defined in the initial input environment, and thus has the same definition in both env_r and env_m.

The final case to consider is thus that of k bound to a dynamic expression e in env_m, so that $find\ env_m\ k = Expr\ e$. If at any point in the recursion of *rec_reduce_derivs* k is an instantiable key, then we have that

$$\begin{array}{ll}
 & eval\ env_r\ (reduce\ env_m\ expr) \\
= & \quad \{\text{definition of } reduce\} \\
 & eval\ env_r\ (reduce\ env_m\ e) \\
= & \quad \{\text{IH on } e\} \\
 & eval\ env_m\ e \\
= & \quad \{\text{definition of } elookup\} \\
 & elookup\ env_m\ k \\
= & \quad \{\text{definition of } eval\} \\
 & eval\ env_m\ expr
\end{array}$$

Otherwise, *reduce* retains the lookup of k, and since *has_undefined_var* is false for env_r we must have $k \in env_r$. From the definition of *add_reduced_definitions* we have that

$$find\ env_r\ k = reduce\_key\ env_m\ k \tag{5.3.1}$$

and so

$$\begin{array}{ll}
 & eval\ env_r\ (reduce\ env_m\ expr) \\
= & \quad \{\text{definition of } reduce\} \\
 & eval\ env_r\ expr \\
= & \quad \{\text{definition of } eval\} \\
 & elookup\ env_r\ k \\
= & \quad \{\text{definition of } elookup\text{, (5.3.1), and definition of } reduce\_key\} \\
 & eval\ env_r\ (reduce\ env_m\ e) \\
= & \quad \{\text{continuing as above}\} \\
 & eval\ env_m\ expr
\end{array}$$

□

We have now shown that partial evaluation of a model will not change the results of simulations, and that no code duplication will occur in a specialised model. Such requirements are standard for partial evaluation of any language, however. CellML, being a modelling language concerned with the real world, also includes units information, and as we saw in Chapter 4 it is important that the use of units within a model is consistent. We must thus show that our partial evaluator maintains this consistency, and this is the subject of the next theorem.

Theorem 5.6 Appropriate units definitions are added to the reduced model. That is, for any constant c in the reduced model with units u (which are not standard CellML units), the units environment *env*$_u$ of the reduced model contains a binding of a name n_u :: *UName* to the units definition u :: *Units*, and the units of c in the reduced model are given as n_u (i.e. a named reference, rather than an anonymous reference to u). Further, n_u is the only name bound to u in the units environment (assuming that the original model did not contain duplicate definitions).

Proof. The units environment is contained within the 'internal data' part of a model environment. Units references within expressions have type **Either** *UName Units*, and as such are either by name to units defined in the environment, or to 'anonymous' units definitions given explicitly. During partial evaluation, when a constant is created, the units field is filled in with the units of the expression as an 'anonymous' reference, as explained in Section 5.2.3. This avoids the need for PE to change the environment as well as individual expressions. After PE is complete, the *define_pe_units* function is used to add these anonymous units to the units environment and change the references within constants to use names.

Under certain assumptions, including correctness of the *foldr_expr_key* and *map_foldr* functions, this theorem is straightforward to prove informally. The key component is the *define_units* function. It uses the equality relationship defined on the *Units* datatype to avoid adding multiple definitions of u to the units environment, so we require this to actually capture equality of definitions, rather than being an equivalence relation. As discussed at the end of Chapter 4, due to the use of floating point arithmetic in comparing definitions this is not absolutely ensured, but the assumption is unlikely to fail.

If a definition equal to u is found, *define_units* returns the name n_u already bound to u, and leaves the units environment unchanged. Otherwise, it uses *uniq_key* to generate a unique key n_u such that $n_u \notin env_u$, and returns both n_u and the new units environment $env_u \oplus \{n_u \mapsto u\}$, with the new binding contained in the 'model global' portion of env_u. The correctness of *uniq_key*, shown below, follows directly from properties of sets.

```
uniq_key :: UnitsEnv → UName
uniq_key uenv
  = head (dropWhile used (map num2uname [0..]))
  where names_set = Set.fromList (names uenv)
        num2uname n = "___" ++ show n
        used n = Set.member n names_set
```

Finally, *def_units_expr* is used to process each constant within the reduced model, and uses the results of *define_units* to update the units reference as needed. □

With these results, we can thus have confidence that the partial evaluation algorithm described in this chapter will not change the results of simulations of models. Partial evaluation therefore provides us with a reliable, provably correct optimisation of mathematical models described in CellML.

5.4 Discussion

We have presented an application of the optimisation techniques of partial evaluation to the domain specific modelling language CellML, and proved it correct. Due to the unusual nature of CellML, our partial evaluator contains features not seen in those for other languages. We believe the integration of physical units to be unique. Also, since we have no need to self-apply PE, we chose a partially online strategy to obtain some further small improvements.

One issue that has not yet been addressed is the impact on our proof of floating point arithmetic. We have assumed that all PE-time calculations have been of sufficient precision to not unduly affect the accuracy of simulations. This is a reasonable assumption for two reasons. Firstly, the precision is much greater than that used for parameter values. Secondly and more importantly, as we shall see in the next chapter, greater errors do not affect simulation accuracy.

The proof presented here has shown the reliability of the Haskell implementation of PE for CellML, and hence of the essential PE algorithm. However, as noted in Section 5.2, in practice we use a Python implementation for transforming CellML models. Our proof as given here does not directly address the reliability of that implementation. However, we are able to compare the results of the two implementations (utilising a function which converts a model environment back to an instance of the *CellML* type), and the fact that the Haskell implementation has been proven correct then allows us to verify the output of the Python implementation on each model in our sample. This gives us an additional level of confidence above that obtained by unit testing and comparing simulation results.

The comparison mentioned above is not, unfortunately, exact. This is due to the fact that the Python implementation has a better algorithm for adding units definitions to the reduced model. It uses a different data structure for units, with references done by name rather than including the reference unit's definition directly (as is done with the *ComplexUnitsRef* type for instance; see Section 4.3). This then allows it to generate more human-readable names for definitions, based on these named references.

The primary reasons for having a Python implementation, rather than just using Haskell, are pragmatic. Whilst our initial approach to partially evaluating CellML used Haskell, specifically the Haskell XML Toolbox, to convert CellML into Scheme (whence an existing partial evaluator could be used), this was found to be difficult to work with. A decision was thus made to start from scratch in implementing PE for the CellML language directly, and Python was chosen as being a good fit both for our experience and that of the intended user community—Python has gained significant popularity particularly among bioinformaticians, and it is also used for pre-existing CellML-related software, notably the model repository; other CellML tools are also written using the object-oriented paradigm. It was thus felt that other developers would be more able to contribute to and use a Python-based tool.

The Python implementation was designed with both extensibility and performance in mind.

It essentially operates directly on the XML tree (using a data binding library), and endeavours to do something sensible with any data or metadata that it is not transforming. Since CellML is designed to be extensible, this flexibility is useful. It also makes much use of 'bookkeeping' state to avoid recomputation of data during the validation and optimisation process.

However, the Python implementation has definite shortcomings. The PE algorithm is obscured by the 'plumbing' introduced by the data binding library, the heavy use of state, and the object oriented design of the system. Python, being a dynamically typed imperative language, also lacks the desirable features of Haskell that facilitate program proofs. Hence when we considered proving the correctness of our techniques, we returned to a Haskell implementation. We have also found that the experience of developing two essentially independent implementations has been of assistance in refining the techniques.

With hindsight, we can see benefits to having the primary implementation in Haskell, and this may be the direction we take in the future. With suitable 'meta' functions for processing the data types involved, model transformations can be described very concisely and elegantly. Haskell's pattern matching also provides us with an easy way to apply transformations only to expressions having a certain form. There may be scope for using automatic theorem provers to analyse the correctness of optimisations. These considerations suggest that Haskell would be a better choice for developing a more general model transformation framework.

In Chapter 7 we will consider the question of effectiveness, with experimental results of applying this optimisation to a sample of models. Firstly, however, we look at another technique for optimising cardiac simulations, and its application to CellML.

6

Lookup Tables

In this chapter we consider a further optimisation technique which may be applied to enable more efficient simulation of CellML models. Lookup tables have long been used in cardiac simulations, with the modifications required being manually applied to hand-coded implementations of models. We have automated the technique for models described in CellML.

Section 6.1 describes how the technique works, and also explains how the analysis of when a lookup table may be used can be automated. This is followed in Section 6.2 with a brief discussion of how the use of lookup tables affects code generation.

The remainder of the chapter is devoted to our other contribution in this area—an analysis of the error introduced by this optimisation. Since the use of lookup tables involves approximating certain expressions within the model, we cannot prove that the meaning of the model is unchanged as we did for partial evaluation in the previous chapter. Instead we must show that the error introduced, and then accumulated over the course of a simulation, is within a tolerance level.

6.1 Introduction

A further optimisation which may be applied to CellML models is the use of lookup tables to precompute the values of expressions that would otherwise be repeatedly calculated.

This technique works because several expressions in most cardiac cell models contain only one dependent variable: the transmembrane potential. For example, the following occurs in the LR91 model (Luo and Rudy, 1991):

$$\beta_m = 0.08e^{\frac{-V}{11}}. \tag{6.1.1}$$

Under physiological conditions the transmembrane potential V usually lies between $-100\,\mathrm{mV}$ and $50\,\mathrm{mV}$, and so a table T can be generated of precomputed values of β_m for potentials within this range. Then, given any transmembrane potential within the range, a value for β_m can quickly be computed using linear interpolation between two entries (T_i and T_{i+1}) of the lookup table:

$$\beta_m = T_i + \frac{(T_{i+1} - T_i)(V - V_i)}{V_{i+1} - V_i}, \tag{6.1.2}$$

where V_j is the voltage used in computing T_j. If V lies outside the range, this can either be considered an error condition, or the original equation can be used.

Another technique sometimes described as using lookup tables is memoization (Michie, 1968). This is a computer science technique in which the return values of function calls are automatically cached, and if the function is called again with the same arguments then the saved result is used, thus avoiding recomputation of the function. This is not the same as our use of the term lookup tables: here values are precomputed rather than computed on demand, and interpolation is used to approximate values where an exact match is not available.

The use of a lookup table is a worthwhile efficiency saving when the expression contains exponential (as in the case for equation (6.1.1), and generally in Hodgkin–Huxley style formulations of ion channel behaviour) or trigonometric functions, since these are expensive to compute.[1] The technique has been in use for some time (e.g. Dexter et al., 1989; Fox et al.,

[1] An estimate of the relative costs is given in Chapter 7, as are further examples.

2002), with the lookup table code hand-written for each equation; we generate the code automatically, as well as treating subexpressions rather than just whole equations.

The (explicit) mathematics contained within a CellML file is described using MathML, and hence is tree structured. It is thus trivial to construct a recursive algorithm to check any expression (either the whole right hand side of an assignment, or a subexpression thereof) for suitability for conversion to using a lookup table. The two key criteria we check are:

1. the only variable used is the transmembrane potential V;

2. the expression contains exponential, logarithmic, or trigonometric functions.

These criteria are checked for each node of the expression tree, starting at the top. The recursion terminates (for a given branch) as soon as both are satisfied, so that maximal subexpressions are matched.

The lookup tables transformation annotates suitable expressions with attributes in an extension namespace specifying how to generate the table,[2] since CellML does not yet contain suitable constructs for describing these. It contains no tabular or array data types (every variable is considered to be a real number), and there is also no facility for specifying that certain expressions should be evaluated prior to simulation, rather than at every timestep. The annotations used specify the variable used to index the table, minimum and maximum values for this variable, and the step size (which we denote by τ; choosing a suitable τ will be discussed later). This provides more flexibility than is required when considering only tables indexed by the transmembrane potential, but allows for possible future extensions to other variables constrained by physiological bounds.

A code generation tool may make use of these annotations to incorporate lookup tables within the generated cell model source code. The precise form this code takes can vary depending on the target programming language and simulation environment. Indeed, one advantage of decoupling the lookup table analysis from the code generation is that one may easily exper-

[2]Unsuitable expressions are annotated with the reasons why a table may not be used.

iment with different representations for lookup tables. This question is considered further in Section 6.2.

The first criterion for when a lookup table may be used is very simplistic. Constant variables, and variables whose values depend only on constants, could be included within an expression that is converted to use a lookup table. The key question is:

is the value known when the lookup table is generated?

In other words, can the value of the expression be computed (given a value for V) when the lookup tables are generated prior to simulation, or does such a computation require values that are not known until the simulation is in progress? Such an analysis however is basically a binding time analysis. Hence if partial evaluation is applied before the lookup table analysis then a more complex version of the second criterion is not required. This is because expressions which can be computed prior to simulation are computed by partial evaluation and replaced by their value as a constant. The correctness proof for the lookup table analysis then comes from the proof of the partial evaluator.

6.2 Lookup tables in generated code

The representation of lookup tables within generated code is important for the effectiveness of the technique. In particular, it is crucial to lay out the values in memory such that access to the tables makes good use of the memory cache in typical computer architectures. When performing a simulation, at a given time step every table will (typically) be accessed, but the same index will be used for each table. Rather than storing each table in its own block of memory, it thus makes sense to arrange the tables together, such that values for a given index are stored contiguously. They are then likely to share a cache line, leading to fewer cache misses and better data throughput.

When generating C++ code for use with Chaste (Pitt-Francis et al., 2008), the current version of PyCml generates two classes: one for the cell model itself, and one following the Singleton design pattern (Gamma et al., 1995) containing the lookup tables for the model. The use of a singleton means that even in a multi-cellular simulation only one instance of the lookup tables will be generated, thus keeping memory usage to a minimum. The generation of the lookup tables is performed within the constructor of this class, and (inline) methods are provided to perform linear interpolation on each table. Within the cell model class, calls to these methods replace the appropriate expressions.

Since the index i and the factor $(V-V_i)/(V_{i+1}-V_i)$ in the linear interpolation formula (6.1.2) are common to all tables, they are computed once per timestep within the cell model class, and passed to the interpolation methods. The index i is given by the integer part of $(V - V_0)/\tau$.

6.3 Lookup table error analysis

The analysis above of when an expression may be replaced by a lookup table is not the only facet we must consider in regard to proving this optimisation correct. Recall that the use of lookup tables involves an approximation—the expression replaced is approximated by a piecewise-linear function. We therefore cannot say that the transformed model has precisely the same meaning as the original, since an error is introduced. Instead, we must show that this error, even accumulated over the course of a simulation, will be sufficiently small that it may be ignored. In other words, using the same informal notation as in Section 5.3, we must show that for any ODE solver used,

$$\begin{array}{l} \forall\; i \in \mathit{inputs},\; c \in \mathit{CellMLmodels} \\ \quad \|\mathit{solver}(c, i) - \mathit{solver}(\mathcal{LT}\; c, i)\| \leq \varepsilon \end{array}$$

for some suitable error bound ε.

To the best of our knowledge, there is no existing work analysing this error in any rigorous way. Instead, the accuracy of a simulation using lookup tables is verified by running the same simulation without lookup tables, perhaps also using a smaller time step (see e.g. Fox et al., 2002). Usually the error in physiological quantities of interest for the study, such as action potential duration, is considered, rather than comparing the solutions directly. We have sought to use mathematical error analysis to determine a computable error bound which will allow the lookup table step size to be chosen so as to guarantee that the error is within acceptable limits.

One approach is to use truncation error analysis, noting that the problem bears some similarities to analysing the error caused by floating point inaccuracies. However, the ‘stiff’ nature of the ODE systems representing cardiac cells (the fact that the upstroke of an action potential occurs on a very fast timescale, whereas the rest of the activity occurs on a much longer timescale) makes this approach infeasible. The error bound obtained is far too large to be useful, as well as being very difficult to calculate, as is shown in Section 6.4.

A better approach is to use *a posteriori* error analysis, which has its roots in adaptive finite element techniques. Finite element methods discretise the problem domain into a ‘mesh’ of elements, and approximate the true solution by a function (e.g. a low-order polynomial) on each element. *A posteriori* error analysis is used to refine the mesh until a prescribed global accuracy is achieved. It provides an algorithm for computing the error bound, and gives a much tighter bound, within an order of magnitude of the actual error. This approach is the subject of Sections 6.5 and 6.6.

6.4 Truncation error analysis

Truncation error analysis considers the propagation of errors due to ‘truncation’ of values at a given precision, such as is caused by the limited precision of floating point arithmetic. We describe the main features of the technique here, and refer the reader to books for further details (Lambert, 1973; Henrici, 1962, 1963).

Firstly, some notation. Recall the generic ODE system model of Equation (3.0.1), which we can write as

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{y}, t), \qquad \boldsymbol{y}(a) = \boldsymbol{\eta}, \tag{6.4.1}$$

where $\boldsymbol{\eta}$ gives the initial conditions of the state variables at time a. We denote the components of the vector $\boldsymbol{y}$ by $y_1, y_2, \ldots, y_M$.

A Lipschitz condition ensures that (6.4.1) has a unique solution on some region D, defined by $a \le t \le b$ (a and b finite), $-\infty < y_i < \infty$, $i = 1, 2, \dots, M$. We require that there exist a constant L such that, for every $t, \boldsymbol{y}, \boldsymbol{y}^*$ such that $(\boldsymbol{y}, t)$ and $(\boldsymbol{y}^*, t)$ are both in D,

$$\|\boldsymbol{f}(\boldsymbol{y}, t) - \boldsymbol{f}(\boldsymbol{y}^*, t)\| \le L\|\boldsymbol{y} - \boldsymbol{y}^*\|, \tag{6.4.2}$$

where $\|\cdot\|$ denotes some vector norm.

Computational methods to solve ODEs generally seek an approximation to $\boldsymbol{y}(t)$ at a set of discrete points $\{t_n | n = 0, 1, \dots, (b-a)/h\}$, where h is the *steplength* of the method, and is constant for all methods considered here. Let $\boldsymbol{y}_n$ be an approximation to $\boldsymbol{y}(t_n)$, the theoretical solution at t_n, and let $\boldsymbol{f}_n \equiv \boldsymbol{f}(\boldsymbol{y}_n, t_n)$. A *linear k-step method* for determining the $\boldsymbol{y}_n$ takes the form of a linear relationship between $\boldsymbol{y}_{n+j}, \boldsymbol{f}_{n+j}, j = 0, 1, \dots, k$, and can be written as

$$\sum_{j=0}^{k} \alpha_j \boldsymbol{y}_{n+j} = h \sum_{j=0}^{k} \beta_j \boldsymbol{f}_{n+j}, \tag{6.4.3}$$

where the α_j and β_j are constants. We assume that α_0 and β_0 are not both 0, and without loss of generality we can assume that $\alpha_k = 1$.

To solve this difference equation we first need to obtain a set of *starting values* $\boldsymbol{y}_0, \dots, \boldsymbol{y}_{k-1}$. In the case where $k = 1$ we only need $\boldsymbol{y}_0$, and normally choose $\boldsymbol{y}_0 = \eta$. Methods for obtaining these values will not be discussed here (see e.g. Süli and Mayers, 2003).

We say that the method (6.4.3) is *explicit* if $\beta_k = 0$, and *implicit* otherwise. For an explicit method we can obtain the current value $\boldsymbol{y}_{n+k}$ directly from (6.4.3), and so such methods are easier to compute, but implicit methods enjoy better numerical stability properties.

Henrici (1963, Theorem 4.1) gives the following bound for the global truncation error $\boldsymbol{e}_n = \boldsymbol{y}(t_n) - \boldsymbol{y}_n$ when $h|\alpha_k|^{-1}L|\beta_k| < 1$:

$$\|\boldsymbol{e}_n\| \le \Gamma^*[Ak\delta + (t_n - a)GYh^p]e^{(t_n-a)\Gamma^* LB}, \tag{6.4.4}$$

where

$$\begin{aligned} Y &= \max_{t\in[a,b]} \|\boldsymbol{y}^{(p+1)}(t)\|, \\ A &= \sum_{j=0}^{k} |\alpha_j|, \\ B &= \sum_{j=0}^{k} |\beta_j|, \\ \Gamma^* &= \frac{\Gamma}{1 - h|\alpha_k|^{-1}L|\beta_k|}, \\ \Gamma &= \sup_{l=0,1,\dots} |\gamma_l|, \\ 1/(\alpha_k + \alpha_{k-1}\zeta + \ldots + \alpha_0\zeta^k) &= \gamma_0 + \gamma_1\zeta + \gamma_2\zeta^2 + \ldots, \end{aligned}$$

and δ is the maximum error in the starting values under some vector norm. Note that in the case of an explicit method $\Gamma^* = \Gamma$.

Errors due to lookup tables can be incorporated in the analysis in the same way as Henrici addresses round-off error, leading to the bound

$$\|\boldsymbol{e}_n\| \le \Gamma^*[Ak\delta + (t_n - a)(GYh^p + |\varepsilon|/h\Gamma^*)]e^{(t_n-a)\Gamma^* LB}, \tag{6.4.5}$$

where $|\varepsilon|$ is a small constant bounding the error due to lookup tables introduced at a single step of the ODE solver. As shown by Henrici (1963, Theorem 5.1) the additional accumulated error from this source is bounded by

$$\frac{\varepsilon(t_n - a)}{h} e^{(t_n-a)\Gamma^* LB},$$

which when added to (6.4.4) yields (6.4.5).

Computing those parts of the error bound which depend solely on the ODE solution method may not always be straightforward, but they need only be done once per method and so do not present a significant computational challenge. There are two greater challenges to the practical use of the error bound above. The first is to compute a value for Y, a bound on the $p + 1^{\text{th}}$ derivative of the problem. In general, since $\boldsymbol{f}$ may depend on $\boldsymbol{y}$, we may need to know values of $\boldsymbol{y}$ in order to compute Y, which may not be possible. Fortunately, in the case of cardiac electrophysiological models the physiology of the problem gives us bounds on $\boldsymbol{y}$ which we may in principle use to compute Y, although performing the differentiation will be difficult, since the ODE systems are complex.

Obtaining a value for L is also hard. If we can calculate the 2-norm (i.e. largest eigenvalue) of the Jacobian matrix of the problem, and find its maximum over D, then this gives us L. Given the complexity of cardiac models, however, this approach does not appear fruitful, and there is still a further problem. We can estimate L by choosing random values for the dependent variables, within D, and calculating the corresponding value of L which gives equality in (6.4.2). For the Hodgkin–Huxley system of equations (Hodgkin and Huxley, 1952), this gives a value for L of at least 9000. Unfortunately, such a large value makes the error bound above useless for the purpose of choosing suitable step sizes. The magnitude of the Lipschitz constant is due to the stiff nature of the ODE system—the upstroke of an action potential occurs on a much faster timescale than the remainder of the activity.

Thus we conclude that, while such an error analysis may possibly be useful for proving theoretical safety properties of the lookup table optimisation, it is not suitable for use in determining what table step size to use.

6.5 An *a posteriori* error analysis

The aim of *a posteriori* error analysis is to obtain a computable bound on the error of an approximation U to the true solution u of a problem, where the approximate solution U is obtained using the finite element method. A key feature of this error analysis is that it does not require any knowledge about the true solution u beyond the fact that it *is* a solution. The error bound is given solely in terms of the computed solution U, not the unknown analytic solution u. The name '*a posteriori*' is due to this fact that the bound is only computable *after* the numerical solution to the problem has been obtained. This section derives some error bounds, and the results of applying them are shown in Section 6.6.

In this section, the system of M ODEs representing a cardiac cell model is written as

$$\begin{aligned} \frac{\mathrm{d}u_i}{\mathrm{d}t} + f_i(\mathbf{u}, t) &= 0, \qquad T_0 < t \leq T_1, \quad i = 1, 2, \ldots, M, && (6.5.1) \\ u_i(0) &= u_{i,0}, \qquad i = 1, 2, \ldots, M, && (6.5.2) \end{aligned}$$

with the initial conditions $u_{i,0}$. The system is simulated over the time interval $[T_0, T_1]$.

The key assumption made about the true solution $\mathbf{u}$ is that it is continuous for the duration of the simulation, and that $\mathrm{d}\boldsymbol{u}/\mathrm{d}t$ exists everywhere. This is a reasonable assumption since we expect biological systems to exhibit fairly smooth continuous behaviour, and this should therefore be reflected in models of such systems.

In the next subsection, we introduce the main concepts of the finite element method, and then in Section 6.5.2 show how it applies to solving the system of ODEs above, and how the use of lookup tables can be incorporated. Subsequent sections deal with the error analysis, first introducing the technique in general, then applying it to cardiac cell models.

An important benefit of *a posteriori* error analysis is that, rather then just determining the error of the solution in some norm, it can be applied to calculate a bound on the error in any *functional* of the solution.[3] Section 6.5.6 uses this fact to derive two alternative error bounds, which provide more insight into the behaviour of the models.

Finally, Section 6.6 evaluates the effectiveness of these techniques on a variety of cell models, as well as considering questions such as the robustness of the analysis, and convergence of the error as the mesh size or lookup table step size are reduced.

6.5.1 The finite element method

Galerkin's method for solving a general differential equation is based on finding an approximate solution as a function in a (finite-dimensional) space of functions. This space should be spanned by a set of basis functions which are easy to differentiate and integrate. In the finite-dimensional

[3]A functional is a real-valued function on a vector space V, usually of functions. Recall that the finite element method gives a solution of the problem as a function, so a function of this solution is a functional.

case, the method leads to a system of linear or nonlinear simultaneous equations which may be solved using a computer.

The basic finite element method is thus just Galerkin's method using piecewise polynomials for the basis functions. Süli and Mayers (2003) give a very brief introduction to the finite element method, from a fairly mathematical viewpoint, while Eriksson et al. (1996) provide a good introduction to the topic, including examples of its application to various classes of problems.

Given a set of basis functions $\phi_i : i = 1, \dots, n$, any function in the space, including the approximate solution, may be represented as a linear combination of basis functions:

$$U(t) = \sum_{i=1}^{n} \zeta_i \phi_i(t),$$

for some coefficients $\zeta_i \in \mathbb{R}$. The coefficients for the approximate solution are obtained by requiring U to satisfy the differential equation in an 'average' sense, such that U is the solution to the *weak* or *variational* form of the problem.

The method is best understood by way of an example, and so in the next section we apply it to our cardiac model, equations (6.5.1) and (6.5.2).

6.5.2 The finite element method applied to a cardiac model

For the purposes of our presentation of *a posteriori* error analysis applied to cardiac cell models, we will seek a piecewise constant finite element solution to equation (6.5.1). This is equivalent to using the backward Euler method to solve the ODE system. We will thus require piecewise constant basis functions. While piecewise linear basis functions would be likely to give a more accurate solution for a given mesh size, there are advantages to using a low order solution which will be seen later.

Suppose that the time interval of the simulation, $[T_0, T_1]$ is partitioned into N sub-intervals

I_n where

$$I_n = (t_{n-1}, t_n], \qquad \text{and} \qquad T_0 = t_0 < t_1 < \ldots < t_N = T_1.$$

These intervals I_n are the finite elements. The mesh size on element I_n is given by

$$h_j = t_j - t_{j-1}, \qquad j = 1, 2, \ldots, N.$$

We then define the piecewise constant basis functions $V_1, V_2, \ldots, V_N$ such that V_j is 1 on element I_j and zero elsewhere, i.e.

$$V_j(t) = \begin{cases} 1 & t \in I_j \\ 0 & \text{otherwise.} \end{cases} \tag{6.5.3}$$

The vector space spanned by these basis functions is denoted by W.

The finite element solution $\mathbf{U}$ to (6.5.1) is then given by

$$U_i(t) = \sum_{j=1}^{N} U_{i,j} V_j(t), \qquad i = 1, 2, \ldots, M,$$

where the $U_{i,j}$'s are constants that are to be determined. This is known as the "dG(0)" solution, where "d" denotes that the solution is discontinuous, "G" stands for Galerkin, and the "(0)" denotes that the piecewise polynomials are of degree zero (Eriksson et al., 1996, p. 212).

In order to reason effectively about discontinuous functions, we make use of the following notation:

$$\begin{aligned} v(t_{k-1}^{+}) &= \lim_{\varepsilon \to 0^+} v(t_{k-1} + \varepsilon), \\ v(t_{k-1}^{-}) &= \lim_{\varepsilon \to 0^+} v(t_{k-1} - \varepsilon), \end{aligned}$$

where $\varepsilon > 0$. These expressions give the values of the function v as we approach a discontinuity from either side. The "jump" in a quantity across the boundary between I_k and I_{k+1} is denoted by

$$[v^{(k)}] = v(t_k^{+}) - v(t_k^{-})$$

The weak form

The *weak form* of equation (6.5.1) is given by an integral equation. Let $L_2([T_0, T_1])$ denote the vector space of functions that are square–integrable on the interval $[T_0, T_1]$, and let $\mathbf{v} = (v_1, v_2, \ldots, v_M)$ such that $v_i \in L_2([T_0, T_1])$, for $i = 1, 2, \ldots, M$. Then, using equation (6.5.1), we have that

$$\int_{T_0}^{T_1} \left(\frac{\mathrm{d}u_i}{\mathrm{d}t} + f_i(\mathbf{u}, t) \right) v_i \, \mathrm{d}t = 0, \qquad i = 1, 2, \ldots, M.$$

This says that the true solution $\mathbf{u}$ (or rather, its *residual error* $\frac{\mathrm{d}u_i}{\mathrm{d}t} + f_i(\mathbf{u}, t)$) is orthogonal to all such functions $\mathbf{v}$. This orthogonality property is the way in which we specify that the finite element solution satisfies the differential equation in an 'average' sense.

Noting that $u_i(t)$ is continuous, and thus has no jumps, we may write

$$\int_{T_0}^{T_1} \left(\frac{\mathrm{d}u_i}{\mathrm{d}t} + f_i(\mathbf{u}, t) \right) v_i \, \mathrm{d}t + \sum_{k=1}^{N} [u_i^{(k-1)}] v_i(t_{k-1}^+) = 0, \qquad i = 1, 2, \ldots, M, \tag{6.5.4}$$

since all the $[u_i^{(k-1)}]$ are zero. This is the weak form of the problem.

The reason for adding a zero term in (6.5.4) becomes clear when we consider that the finite element solution *may* have jumps, since it is permitted to be discontinuous. Since the true solution is continuous, however, we wish to penalise jumps in the approximate solution, which is accomplished by this term. The coefficients $U_{i,j}$ of the finite element solution are thus calculated by demanding that

$$\begin{gathered} \int_{T_0}^{T_1} \left(\tfrac{\mathrm{d}U_i}{\mathrm{d}t} + f_i(\mathbf{U}, t) \right) V_j \, \mathrm{d}t + \textstyle\sum_{n=1}^{N} [U_i^{(n-1)}] V_j(t_{n-1}^+) = 0, \\ i = 1, 2, \ldots, M, \quad j = 1, 2, \ldots, N, \end{gathered} \tag{6.5.5}$$

thus asserting that $\mathbf{U}$ is orthogonal to all functions in the space W defined earlier. The V_j are often known as *test functions*. In order to satisfy the initial conditions (6.5.2) we fix $U_{i,0} = u_{i,0}$.

As V_j is non-zero only on the interval I_j, and $\mathbf{U}$ is constant on this interval, we may write equation (6.5.5) as

$$U_{i,j} - U_{i,j-1} + h_j f_i(\mathbf{U}_j, t) = 0, \qquad i = 1, 2, \ldots, M, \quad j = 1, 2, \ldots, N, \tag{6.5.6}$$

where $\mathbf{U}_j$ denotes $\mathbf{U}$ on the interval I_j, and we thus obtain a numerical scheme identical to the backward Euler finite difference formula. Note that we have assumed that since $\mathbf{U}$ is constant on I_j, $f_i(\mathbf{U}, t)$ is also. This is usually the case for cardiac models, since they contain no direct reference to t except via a stimulus protocol, and the stimulus is usually represented by a step function, which can be made to change on element boundaries.

6.5.3 The finite element approximation using lookup tables

How may we account for the presence of lookup tables? The use of lookup tables alters the function $\mathbf{f}$ slightly; we denote this altered version by $\hat{\mathbf{f}}$. With this change to the ODE system, we can proceed just as before. The finite element approximation in this case is denoted by

$$\hat{U}_i(t) = \sum_{j=1}^{N} \hat{U}_{i,j} V_j(t), \qquad i = 1, 2, \ldots, M,$$

satisfies

$$\int_{T_0}^{T_1} \left(\frac{\mathrm{d}\hat{U}_i}{\mathrm{d}t} + \hat{f}_i(\hat{\mathbf{U}}, t) \right) V_j \, \mathrm{d}t + \sum_{n=1}^{N} [\hat{U}_i^{(n-1)}] V_j(t_{n-1}^+) = 0, \qquad i = 1, 2, \ldots, M, \quad j = 1, 2, \ldots, N, \tag{6.5.7}$$

and is calculated from

$$\hat{U}_{i,j} - \hat{U}_{i,j-1} + h_j \hat{f}_i(\hat{\mathbf{U}}_j, t) = 0, \qquad i = 1, 2, \ldots, M, \quad j = 1, 2, \ldots, N, \tag{6.5.8}$$

$$\hat{U}_{i,0} = u_{i,0}, \qquad i = 1, 2, \ldots, M. \tag{6.5.9}$$

6.5.4 *A posteriori* error analysis

A posteriori error analysis is based on representing the error in terms of the solution of a continuous *dual problem* to the original ODE system (6.5.1), which is used to determine the effects of the accumulation of errors. To motivate this, consider the simple ODE $\frac{\mathrm{d}u}{\mathrm{d}t} + au = f$ for $0 < t < T$, which has the weak form

$$\int_0^T \left(\frac{\mathrm{d}u}{\mathrm{d}t} + au \right) v \, \mathrm{d}t = \int_0^T f v \, \mathrm{d}t.$$

Integration by parts gives us

$$u(T)v(T) - u(0)v(0) + \int_0^T u(t) \left(-\frac{\mathrm{d}v}{\mathrm{d}t} + av \right) \mathrm{d}t = \int_0^T fv \, \mathrm{d}t,$$

for all test functions v. If we choose v such that it solves the dual problem $-\frac{\mathrm{d}v}{\mathrm{d}t} + av = 0$ for $0 < t < T$, then this simplifies to

$$u(T)v(T) = u(0)v(0) + \int_0^T fv \, \mathrm{d}t.$$

We are thus able to use the fact that u solves the differential equation, and the solution v of the dual problem, to get information about the final value $u(T)$ without knowing the true solution. Nonlinear cardiac cell models are more complex, but the same principle applies. For a detailed survey of the subject of *a posteriori* error estimation, we refer the reader to Ainsworth and Oden (2000).

We now proceed to apply the analysis to equation (6.5.1). Define the error between the (unknown) true solution and the finite element solution calculated using lookup tables as

$$\mathbf{e} = \mathbf{u} - \hat{\mathbf{U}}.$$

We consider the following dual problem, defined to be the system of "backwards" ODEs

$$\begin{aligned} -\frac{\mathrm{d}\boldsymbol{\phi}}{\mathrm{d}t} + A\boldsymbol{\phi} &= \mathbf{e}, \qquad T_0 \le t < T_1, && (6.5.10) \\ \boldsymbol{\phi}(T_1) &= \mathbf{0}, && (6.5.11) \end{aligned}$$

where the matrix A is given by

$$A_{i,j}(t) = \int_0^1 \frac{\partial f_j}{\partial u_i}(s\mathbf{u} + (1-s)\hat{\mathbf{U}}, t) \, \mathrm{d}s, \qquad i = 1, 2, \dots, M, \quad j = 1, 2, \dots, M, \quad (6.5.12)$$

and the term in brackets in the integral above indicates the point at which $\partial f_j / \partial u_i$ is evaluated. This choice of A is required because the ODE is nonlinear, and it allows us to examine the effect of approximating $\mathbf{u}$ by $\hat{\mathbf{U}}$, as we see in result (i) below.

We will seek a bound on the L_2 norm of the error, defined by

$$\|\mathbf{e}\|^2_{L_2([T_0,T_1])} = \int_{T_0}^{T_1} \sum_{i=1}^{M} e_i e_i \,\mathrm{d}t.$$

In the analysis which follows, we will require two subsidiary results.

(i) For a fixed value of t, $\mathbf{u} - \hat{\mathbf{U}}$ is constant and so we may write

$$\begin{aligned}
\sum_{i=1}^{M} A_{i,j} e_i &= \sum_{i=1}^{M} A_{i,j}(u_i - \hat{U}_i) \\
&= \int_0^1 \sum_{i=1}^{M} (u_i - \hat{U}_i) \frac{\partial f_j}{\partial u_i}(s\mathbf{u} + (1-s)\hat{\mathbf{U}}, t)\,\mathrm{d}s \quad \begin{bmatrix}\textit{since } \mathbf{u}-\hat{\mathbf{U}} \textit{ is} \\ \textit{constant}\end{bmatrix} \\
&= \int_0^1 \frac{\mathrm{d}}{\mathrm{d}s}\Big(f_j(s\mathbf{u} + (1-s)\hat{\mathbf{U}}, t)\Big)\,\mathrm{d}s \quad \begin{bmatrix}\textit{by the general} \\ \textit{chain rule}\end{bmatrix} \\
&= f_j(\mathbf{u}, t) - f_j(\hat{\mathbf{U}}, t).
\end{aligned}$$

(ii) Integrating by parts will give us the following term, which we wish to replace with one containing a jump only in $\mathbf{e}$.

$$\begin{aligned}
\sum_{k=1}^{N}\sum_{i=1}^{M} [\phi_i e_i]_{t_{k-1}}^{t_k} &= \sum_{k=1}^{N}\sum_{i=1}^{M} \left(\phi_i(t_k)e_i(t_k^-) - \phi_i(t_{k-1})e_i(t_{k-1}^+)\right) \\
&\qquad [\textit{assuming continuity of } \boldsymbol{\phi}] \\
&= \sum_{k=1}^{N}\sum_{i=1}^{M} \phi_i(t_k)e_i(t_k^-) - \sum_{k=0}^{N-1}\sum_{i=1}^{M} \phi_i(t_k)e_i(t_k^+) \quad [\textit{renumbering}] \\
&= \sum_{k=0}^{N-1}\sum_{i=1}^{M} \phi_i(t_k)e_i(t_k^-) - \sum_{k=0}^{N-1}\sum_{i=1}^{M} \phi_i(t_k)e_i(t_k^+) \\
&\qquad [\textit{defining } e_i(t_0^-) = 0 \textit{ and using } (6.5.11)] \\
&= \sum_{k=0}^{N-1}\sum_{i=1}^{M} \phi_i(t_k)\left(e_i(t_k^-) - e_i(t_k^+)\right) \quad [\textit{combining like terms}] \\
&= -\sum_{k=0}^{N-1}\sum_{i=1}^{M} \phi_i(t_k^+)[e_i^{(k)}] \quad [\textit{assuming continuity of } \boldsymbol{\phi}] \\
&= -\sum_{k=1}^{N}\sum_{i=1}^{M} \phi_i(t_{k-1}^+)[e_i^{(k-1)}] \quad [\textit{renumbering}]
\end{aligned}$$

Using these results, we may now proceed to manipulate the L_2 norm:

$$
\begin{aligned}
\|\mathbf{e}\|^2_{L_2([T_0,T_1])} &= \int_{T_0}^{T_1} \sum_{i=1}^{M} e_i e_i \,\mathrm{d}t \\
&= \int_{T_0}^{T_1} \sum_{i=1}^{M} e_i \left(-\frac{\mathrm{d}\phi_i}{\mathrm{d}t} + \sum_{j=1}^{M} A_{i,j}\phi_j \right) \mathrm{d}t \quad [\textit{using } (6.5.10)] \\
&= \sum_{k=1}^{N} \int_{t_{k-1}}^{t_k} \sum_{i=1}^{M} e_i \left(-\frac{\mathrm{d}\phi_i}{\mathrm{d}t} + \sum_{j=1}^{M} A_{i,j}\phi_j \right) \mathrm{d}t \\
&= \sum_{k=1}^{N} \left(\sum_{i=1}^{M} \left(\int_{t_{k-1}}^{t_k} \phi_i \frac{\mathrm{d}e_i}{\mathrm{d}t} + \sum_{j=1}^{M} A_{i,j} e_i \phi_j \,\mathrm{d}t - [\phi_i e_i]_{t_{k-1}}^{t_k} \right)\right) \\
&\qquad [\textit{distributing } e_i \textit{ and integrating by parts}] \\
&= \sum_{k=1}^{N} \left(\sum_{i=1}^{M} \left(\int_{t_{k-1}}^{t_k} \phi_i \frac{\mathrm{d}u_i}{\mathrm{d}t} - \phi_i \frac{\mathrm{d}\hat{U}_i}{\mathrm{d}t} \mathrm{d}t \right) + \int_{t_{k-1}}^{t_k} \sum_{j=1}^{M} (f_j(\mathbf{u},t) - f_j(\hat{\mathbf{U}},t))\phi_j \,\mathrm{d}t \right. \\
&\qquad \left. + \sum_{i=1}^{M} \phi_i(t_{k-1}^{+})[e_i^{(k-1)}] \right) \quad \left[\begin{array}{l}\textit{using results (i) and (ii)} \\ \textit{and the definition of } e_i\end{array}\right] \\
&= \sum_{k=1}^{N} \left(\int_{t_{k-1}}^{t_k} \sum_{i=1}^{M} \phi_i \left(\frac{\mathrm{d}u_i}{\mathrm{d}t} + f_i(\mathbf{u},t) - \frac{\mathrm{d}\hat{U}_i}{\mathrm{d}t} - f_i(\hat{\mathbf{U}},t) \right) \mathrm{d}t \right. \\
&\qquad \left. - \sum_{i=1}^{M} \phi_i(t_{k-1}^{+})[\hat{U}_i^{(k-1)}] \right) \quad [\textit{since } u_i \textit{ is continuous}] \\
&= \sum_{k=1}^{N} \left(\int_{t_{k-1}}^{t_k} \sum_{i=1}^{M} -\phi_i f_i(\hat{\mathbf{U}},t) \,\mathrm{d}t - \sum_{i=1}^{M} \phi_i(t_{k-1}^{+})[\hat{U}_i^{(k-1)}] \right) \\
&\qquad [\textit{by } (6.5.1) \textit{ and since } \hat{U}_i \textit{ is piecewise constant}] \\
&= \sum_{k=1}^{N} \left(\int_{t_{k-1}}^{t_k} \sum_{i=1}^{M} -\phi_i \hat{f}_i(\hat{\mathbf{U}},t) \,\mathrm{d}t - \sum_{i=1}^{M} \phi_i(t_{k-1}^{+})[\hat{U}_i^{(k-1)}] \right) \\
&\qquad + \int_{T_0}^{T_1} \sum_{i=1}^{M} \phi_i \left(\hat{f}_i(\hat{\mathbf{U}},t) - f_i(\hat{\mathbf{U}},t) \right) \mathrm{d}t.
\end{aligned}
$$

Using equation (6.5.7) and the fact that $\hat{\mathbf{U}}$ is piecewise constant, we may 'add zero' to obtain

$$
\begin{aligned}
\|\mathbf{e}\|^2_{L_2([T_0,T_1])} &= \sum_{k=1}^{N} \left(\int_{t_{k-1}}^{t_k} \sum_{i=1}^{M} (\bar{\phi}_i - \phi_i) \hat{f}_i(\hat{\mathbf{U}},t) \,\mathrm{d}t + \sum_{i=1}^{M} (\bar{\phi}_i(t_{k-1}^{+}) - \phi_i(t_{k-1}^{+}))[\hat{U}_i^{(k-1)}] \right) \\
&\qquad + \int_{T_0}^{T_1} \sum_{i=1}^{M} \phi_i \left(\hat{f}_i(\hat{\mathbf{U}},t) - f_i(\hat{\mathbf{U}},t) \right) \mathrm{d}t
\end{aligned}
$$

where $\bar{\phi}$ is any function in the vector space W spanned by the basis functions. This gives us an exact formula for the L_2 norm of the error, but it is not feasible to compute it. However, we can use the triangle inequality to obtain an upper bound:

$$\|\mathbf{e}\|^2_{L_2([T_0,T_1])} \leq \sum_{k=1}^{N}\sum_{i=1}^{M} \mathcal{E}_{i,k} + \int_{T_0}^{T_1} \left| \sum_{i=1}^{M} \phi_i \mathcal{F}_i \right| \, \mathrm{d}t \tag{6.5.13}$$

where

$$\begin{aligned} \mathcal{E}_{i,k} &= \int_{t_{k-1}}^{t_k} \left|(\phi_i - \bar{\phi}_i)\hat{f}_i(\hat{\mathbf{U}},t)\right| \, \mathrm{d}t + \left|(\phi_i(t^+_{k-1}) - \bar{\phi}_i(t^+_{k-1}))[\hat{U}_i^{(k-1)}]\right|, & (6.5.14)\\ \mathcal{F}_i &= \max\left|\hat{f}_i(\hat{\mathbf{U}},t) - f_i(\hat{\mathbf{U}},t)\right|. & (6.5.15) \end{aligned}$$

This bound can then be computed as described in the next section, but first we consider what the bound actually tells us. The solution to the dual problem gives us a measure of how errors are propagated through the simulation, while the terms involving $\hat{f}_i$ and f_i measure the production of error on each element. Where ϕ is large or changes rapidly, any error produced will have a greater impact on the overall solution. The term involving $[\hat{U}_i^{(k-1)}]$ penalises large jumps in the solution, since u is continuous. Section 6.6.2 considers further the intuitive meaning of the terms in this error bound.

6.5.5 Practical application of the error bound

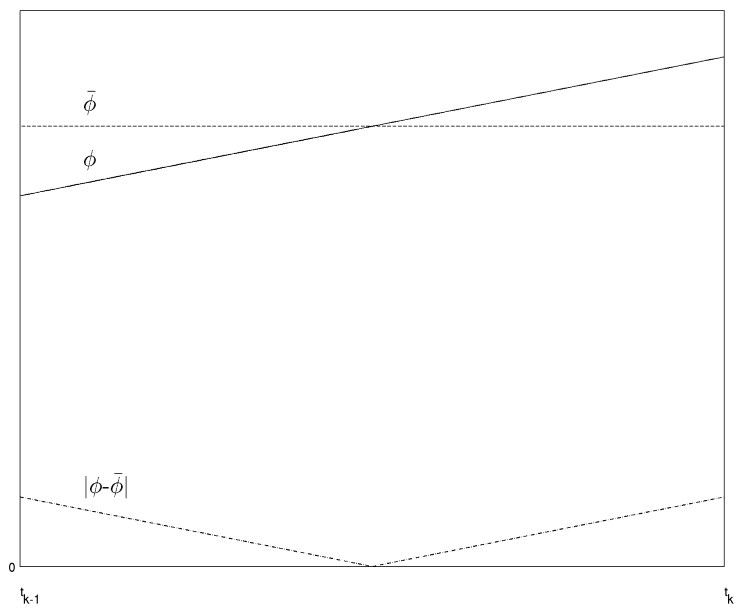
In order to actually compute the error bound above, we first need to solve the ODE system and the dual problem. A suitable choice of $\bar{\phi}$ will then enable us to compute all the terms in the error bound.

The finite element solution using lookup tables is calculated using equation (6.5.8), setting the initial conditions according to equation (6.5.9), and using a suitable nonlinear solver[4] to compute the $U_{i,j}$ for each $j = 1, 2, \ldots, N$ in turn.

In order to solve the dual problem, we need to know A and $\mathbf{e}$. In practice we assume that the finite element solution $\hat{\mathbf{U}}$ is accurate enough for us to approximate the matrix A using this

[4]e.g. `fsolve` in Matlab.

Figure 6.1 An illustrative example of $\int_{I_k} |\phi_i - \bar{\phi}_i|$.



solution. This allows us to write

$$A_{i,j}(t) = \frac{\partial f_j}{\partial u_i}$$

which is evaluated at $\hat{\mathbf{U}}$. Since $\mathbf{e}$ is unknown, we replace the right hand side of equation (6.5.10) by $\mathbf{1}$, assuming that the error is approximately constant everywhere. We can then calculate the dual solution from equations (6.5.10) & (6.5.11), using a higher order finite element method (such as linear finite elements; see below) so that we can assume that our solution is the true solution for $\boldsymbol{\phi}$.

We then define $\bar{\boldsymbol{\phi}}$ piecewise to be the average of $\boldsymbol{\phi}$ on each element I_k:

$$\begin{aligned} \bar{\boldsymbol{\phi}} &= \frac{1}{h_k}\int_{t_{k-1}}^{t_k} \boldsymbol{\phi}\,\mathrm{d}t \\ &= \frac{\boldsymbol{\phi}(t_{k-1}) + \boldsymbol{\phi}(t_k)}{2}; \end{aligned}$$

the latter expression holds if $\boldsymbol{\phi}$ is computed using linear finite elements.

Using linear finite elements to solve the dual problem has the advantage that it is easy to compute integrals involving $\boldsymbol{\phi}$ exactly. Since we have assumed that $\hat{f}_i$ does not vary on I_k we may take it outside the integral when computing $\mathcal{E}$, and determine $\int_{I_k} \left|\phi_i - \bar{\phi}_i\right|$ by summing two triangles, as shown in Figure 6.1. The values of $\mathcal{F}_i$ are calculated using equation (6.5.15). Again, since we have already assumed that $\hat{f}_i$ and f_i do not vary within an element, this is straightforward: we compute

$$\mathcal{F}_i = \max_{0 \le k \le N} \left| \hat{f}_i(\hat{\mathbf{U}}_k, t_k) - f_i(\hat{\mathbf{U}}_k, t_k) \right|.$$

The integral involving $\mathcal{F}$ can then be computed exactly using the trapezium rule.

We now have everything we need to compute the error bound given by equation (6.5.13).

Linear finite element solution of the dual problem

We solve the dual problem using continuous linear finite elements, and thus have linear basis functions $V_0, V_1, \ldots, V_N$ defined such that

$$V_j(t_i) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{6.5.16}$$

Let $C_0^1(T_0, T_1)$ denote the space of functions continuous on (T_0, T_1) which are zero where Dirichlet boundary conditions—initial conditions in this case—are imposed. For any function $v(t) \in C_0^1(T_0, T_1)$, the weak form of the dual problem is given by

$$\int_{T_0}^{T_1} \left(-\frac{\mathrm{d}\boldsymbol{\phi}}{\mathrm{d}t} + A\boldsymbol{\phi} \right) v(t)\ \mathrm{d}t = \int_{T_0}^{T_1} v(t)\ \mathrm{d}t$$

or element-wise as

$$\sum_{k=1}^{N} \int_{t_{k-1}}^{t_k} \left(-\frac{\mathrm{d}\boldsymbol{\phi}_i}{\mathrm{d}t} + \sum_{j=1}^{M} A_{i,j}\phi_j - 1 \right) v(t)\ \mathrm{d}t = 0, \qquad i = 1, 2, \ldots, M. \tag{6.5.17}$$

Let $\boldsymbol{P}(t)$ denote the finite element solution, so that

$$P_i(t) = \sum_{k=0}^{N} P_{i,k} V_k(t), \qquad i = 1, 2, \ldots, M.$$

Note that on I_k,

$$\frac{\mathrm{d}P_i}{\mathrm{d}t} = \frac{P_{i,k} - P_{i,k-1}}{h_k}, \qquad i = 1, 2, \ldots, M.$$

We use $V_0, V_1, \ldots, V_{N-1}$ as test functions, since the dual problem has a Dirichlet boundary condition at T_1. Substituting into (6.5.17) we have, for $i = 1, 2, \ldots, M$ and $l = 0, 1, \ldots, N-1$,

$$0 = \sum_{k=1}^{N} \left(\frac{P_{i,k-1} - P_{i,k}}{h_k} \int_{t_{k-1}}^{t_k} V_l(t)\ \mathrm{d}t + \sum_{j=1}^{M} A_{i,j}(t_k) \int_{t_{k-1}}^{t_k} P_j(t) V_l(t)\ \mathrm{d}t - \int_{t_{k-1}}^{t_k} V_l(t)\ \mathrm{d}t \right),$$

since $A_{i,j}$ is constant on I_k, since $\hat{\mathbf{U}}$ is piecewise-constant. Also, on element I_k only the basis functions V_k and V_{k-1} are non-zero, and so for each l at most 2 terms in each summation over k are non-zero. The values of the integrals are given by

$$\int_{t_{k-1}}^{t_k} V_l(t)\ \mathrm{d}t = \begin{cases} h_k/2 & l = k, k-1, \\ 0 & \text{otherwise;} \end{cases}$$

$$\int_{t_{k-1}}^{t_k} P_j(t) V_l(t)\ \mathrm{d}t = \begin{cases} \frac{h_k}{6}(2P_{j,k} + P_{j,k-1}) & l = k, \\ \frac{h_k}{6}(P_{j,k} + 2P_{j,k-1}) & l = k-1, \\ 0 & \text{otherwise.} \end{cases}$$

We thus obtain the equations

$$\frac{P_{i,0} - P_{i,1}}{2} + \frac{h_1}{6} \sum_{j=1}^{M} A_{i,j}(t_1)(P_{j,1} + 2P_{j,0}) = \frac{h_1}{2}, \qquad i = 1, 2, \ldots, M, \quad l = 0, \tag{6.5.18}$$

and for $i = 1, 2, \ldots, M$ and $l = 1, 2, \ldots, N-1$,

$$\frac{P_{i,l-1} - P_{i,l+1}}{2} + \frac{h_l}{6} \sum_{j=1}^{M} A_{i,j}(t_l)(2P_{j,l} + P_{j,l-1}) + \frac{h_{l+1}}{6} \sum_{j=1}^{M} A_{i,j}(t_{l+1})(P_{j,l+1} + 2P_{j,l}) = \frac{h_l + h_{l+1}}{2}. \tag{6.5.19}$$

For $k = 1, 2, \ldots, M$ we define the column vectors $\tilde{\boldsymbol{x}}_k$ of length M such that $(\tilde{\boldsymbol{x}}_k)_i = P_{i,k}$ for $i = 1, 2, \ldots, M$. Equations (6.5.18) and (6.5.19) can then be written in matrix form as

$$\frac{\tilde{\boldsymbol{x}}_0 - \tilde{\boldsymbol{x}}_1}{2} + \frac{h_1}{6} A(t_1)(\tilde{\boldsymbol{x}}_1 + 2\tilde{\boldsymbol{x}}_0) = \frac{\boldsymbol{h}_1}{2}, \qquad l = 0, \tag{6.5.20}$$

and for $l = 1, 2, \ldots, N-1$,

$$\frac{\tilde{\boldsymbol{x}}_{l-1} - \tilde{\boldsymbol{x}}_{l+1}}{2} + \frac{h_l}{6} A(t_l)(2\tilde{\boldsymbol{x}}_l + \tilde{\boldsymbol{x}}_{l-1}) + \frac{h_{l+1}}{6} A(t_{l+1})(\tilde{\boldsymbol{x}}_{l+1} + 2\tilde{\boldsymbol{x}}_l) = \frac{\boldsymbol{h}_l + \boldsymbol{h}_{l+1}}{2}. \tag{6.5.21}$$

To solve this system of equations we construct the linear system

$$\begin{pmatrix} \tilde{D}_{0,0} & \tilde{D}_{0,1} & \cdots & \tilde{D}_{0,N} \\ \tilde{D}_{1,0} & \tilde{D}_{1,1} & \cdots & \tilde{D}_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{D}_{N,0} & \tilde{D}_{N,1} & \cdots & \tilde{D}_{N,N} \end{pmatrix} \begin{pmatrix} \tilde{\boldsymbol{x}}_0 \\ \tilde{\boldsymbol{x}}_1 \\ \vdots \\ \tilde{\boldsymbol{x}}_N \end{pmatrix} = \begin{pmatrix} \tilde{\boldsymbol{b}}_0 \\ \tilde{\boldsymbol{b}}_1 \\ \vdots \\ \tilde{\boldsymbol{b}}_N \end{pmatrix}$$

where the $\tilde{D}_{i,j}$ are M-by-M matrices and the $\tilde{\boldsymbol{b}}_i$ are column vectors of length M, given by

$$\begin{aligned}
\tilde{D}_{0,0} &= I/2 + h_1 A(t_1)/3, && \\
\tilde{D}_{0,1} &= -I/2 + h_1 A(t_1)/6, && \\
\tilde{D}_{l,l-1} &= I/2 + h_l A(t_l)/6, && l = 1, 2, \dots, N-1, \\
\tilde{D}_{l,l} &= h_l A(t_l)/3 + h_{l+1} A(t_{l+1})/3, && l = 1, 2, \dots, N-1, \\
\tilde{D}_{l,l+1} &= -I/2 + h_{l+1} A(t_{l+1}/6, && l = 1, 2, \dots, N-1, \\
\tilde{D}_{N,N} &= I, && \\
\tilde{D}_{i,j} &= \mathbf{0}, && \text{otherwise}, \\
\tilde{\boldsymbol{b}}_0 &= \boldsymbol{h}_1/2, && \\
\tilde{\boldsymbol{b}}_l &= (\boldsymbol{h}_l + \boldsymbol{h}_{l+1})/2, && l = 1, 2, \dots, N-1, \\
\tilde{\boldsymbol{b}}_N &= \mathbf{0}. &&
\end{aligned}$$

This system can also be written as $D\boldsymbol{x} = \boldsymbol{b}$ and may be solved by, for example, Gaussian elimination.

6.5.6 Alternative error measures

A posteriori error analysis can be used to determine the error in any *functional* of the solution, rather than just the L_2 norm of the error as done above (see e.g. Harriman et al., 2004). Two measures in particular give us a better grasp of the behaviour of the system.

Error at the end time

A heart beat is a cyclical process, and this is reflected in the ODE system used to model the action potential. After a beat, it returns to a resting state, ready for the next stimulus. It is thus instructive to consider what difference the use of lookup tables makes to this resting state. If after a single beat the system is returned to essentially the same state as it would be in without lookup tables, then simulations of many beats will be robust.

To derive a bound on the error of a component m of the system at time T_1, we consider the following dual problem, which is quite similar to our simple example in Section 6.5.4:

$$\begin{aligned} -\frac{\mathrm{d}\boldsymbol{\phi}}{\mathrm{d}t} + A\boldsymbol{\phi} &= \mathbf{0}, \qquad T_0 \le t < T_1, && (6.5.22) \\ \boldsymbol{\phi}(T_1) &= \boldsymbol{v}, && (6.5.23) \end{aligned}$$

where A is defined by (6.5.12) as before, and $\boldsymbol{v}$ is given by

$$v_i = \begin{cases} 1 & i = m, \\ 0 & i \neq m, \end{cases} \qquad (6.5.24)$$

that is, $\boldsymbol{v}$ selects the desired component. Again we will solve this dual problem using linear finite elements. The derivation is very similar to Section 6.5.5, and we obtain the same matrix D. The right hand side $\boldsymbol{b}$ is given by

$$\begin{aligned} \tilde{b}_l &= \mathbf{0}, \qquad l = 0, 1, \ldots, N-1, \\ \tilde{b}_N &= \boldsymbol{v}. \end{aligned}$$

We will need the following result to enable us to get at the component of interest in order to bound it.

$$\begin{aligned} \sum_{k=1}^{N}\sum_{i=1}^{M}[\phi_i e_i]_{t_{k-1}}^{t_k} &= e_m(T_1) - \sum_{i=1}^{M}\phi_i(t_{N-1})e_i(t_{N-1}^{+}) \\ &\quad + \sum_{k=1}^{N-1}\sum_{i=1}^{M}\left(\phi_i(t_k)e_i(t_k^{-}) - \phi_i(t_{k-1})e_i(t_{k-1}^{+})\right) \\ &\quad \left[\textit{assuming continuity of } \boldsymbol{\phi}\right] \end{aligned}$$

$$
\begin{aligned}
&= e_m(T_1) - \sum_{i=1}^{M} \phi_i(t_{N-1}) e_i(t_{N-1}^+) + \sum_{k=1}^{N-1}\sum_{i=1}^{M} \phi_i(t_k) e_i(t_k^-) \\
&\quad - \sum_{k=0}^{N-2}\sum_{i=1}^{M} \phi_i(t_k) e_i(t_k^+) \quad [\textit{renumbering}] \\
&= e_m(T_1) + \sum_{i=1}^{M} \phi_i(t_{N-1})(e_i(t_{N-1}^-) - e_i(t_{N-1}^+)) \\
&\quad + \sum_{k=1}^{N-2}\sum_{i=1}^{M} \phi_i(t_k)\left(e_i(t_k^-) - e_i(t_k^+)\right) + \sum_{i=1}^{M} \phi_i(t_0)(0 - e_i(t_0^+)) \\
&\quad [\textit{considering the first and last elements separately}] \\
&= e_m(T_1) + \sum_{k=0}^{N-1}\sum_{i=1}^{M} \phi_i(t_k)\left(e_i(t_k^-) - e_i(t_k^+)\right) \quad [\textit{defining } e_i(t_0^-) = 0] \\
&= e_m(T_1) - \sum_{k=0}^{N-1}\sum_{i=1}^{M} \phi_i(t_{k-1})[e_i^{(k-1)}]
\end{aligned}
$$

Hence by (6.5.22)

$$
\begin{aligned}
0 &= \int_{T_0}^{T_1} \sum_{i=1}^{M} e_i \left(-\frac{\mathrm{d}\phi_i}{\mathrm{d}t} + \sum_{j=1}^{M} A_{i,j}\phi_j \right) \mathrm{d}t \\
&= \sum_{k=1}^{N} \int_{t_{k-1}}^{t_k} \sum_{i=1}^{M} e_i \left(-\frac{\mathrm{d}\phi_i}{\mathrm{d}t} + \sum_{j=1}^{M} A_{i,j}\phi_j \right) \mathrm{d}t \\
&= \sum_{k=1}^{N}\sum_{i=1}^{M} \left(\int_{t_{k-1}}^{t_k} \phi_i \frac{\mathrm{d}e_i}{\mathrm{d}t} + \sum_{j=1}^{M} A_{i,j} e_i \phi_j \,\mathrm{d}t - [\phi_i e_i]_{t_{k-1}}^{t_k} \right) \quad [\textit{integrating by parts}] \\
&= \sum_{k=1}^{N}\sum_{i=1}^{M} \int_{t_{k-1}}^{t_k} \phi_i \frac{\mathrm{d}e_i}{\mathrm{d}t} + \sum_{j=1}^{M} A_{i,j} e_i \phi_j \,\mathrm{d}t - e_m(T_1) + \sum_{k=0}^{N-1}\sum_{i=1}^{M} \phi_i(t_{k-1})[e_i^{(k-1)}],
\end{aligned}
$$

using the result above at the last step. Following the earlier analysis, we thus have that

$$
\begin{aligned}
e_m(T_1) &= \sum_{k=1}^{N} \left(\int_{t_{k-1}}^{t_k} \sum_{i=1}^{M} -\phi_i \hat{f}_i(\hat{\mathbf{U}}, t)\,\mathrm{d}t - \sum_{i=1}^{M} \phi_i(t_{k-1})[\hat{\mathbf{U}}_i^{(k-1)}] \right) \\
&\quad + \int_{T_0}^{T_1} \sum_{i=1}^{M} \phi_i \left(\hat{f}_i(\hat{\mathbf{U}}, t) - f_i(\hat{\mathbf{U}}, t) \right) \mathrm{d}t,
\end{aligned}
$$

and so

$$
|e_m(T_1)| \leq \sum_{k=1}^{N}\sum_{i=1}^{M} \mathcal{E}_{i,k} + \int_{T_0}^{T_1} \left| \sum_{i=1}^{M} \phi_i \mathcal{F}_i \right| \mathrm{d}t \tag{6.5.25}
$$

with $\mathcal{E}$ and $\mathcal{F}$ defined as before (equations (6.5.14) and (6.5.15)).

Error in the area under the curve

The greatest error is introduced during the upstroke of the action potential, when the solution varies dramatically on a short timescale. However, since the upstroke is of short duration, it is plausible that the area bounded by the graph of the transmembrane potential is only slightly affected by the use of lookup tables.

Let us consider the error in the area under the curve of component m of the ODE system. The relevant dual problem is given by

$$\begin{aligned} -\frac{\mathrm{d}\boldsymbol{\phi}}{\mathrm{d}t} + A\boldsymbol{\phi} &= \boldsymbol{v}, \qquad T_0 \le t < T_1, && (6.5.26)\\ \boldsymbol{\phi}(T_1) &= \mathbf{0}, && (6.5.27) \end{aligned}$$

where A and $\boldsymbol{v}$ are as defined by (6.5.12) and (6.5.24) respectively. Again the matrix D is given by our earlier analysis, whereas $\boldsymbol{b}$ takes values

$$\begin{aligned} \tilde{b}_0 &= \boldsymbol{v}\boldsymbol{h}_1/2,\\ \tilde{b}_l &= \boldsymbol{v}(\boldsymbol{h}_l + \boldsymbol{h}_{l+1})/2, \qquad l = 1, 2, \ldots, N-1,\\ \tilde{b}_N &= \mathbf{0}. \end{aligned}$$

The quantity we wish to bound is given by

$$\begin{aligned} \int_{T_0}^{T_1} e_m \,\mathrm{d}t &= \int_{T_0}^{T_1} \sum_{i=1}^{M} e_i v_i \,\mathrm{d}t\\ &= \int_{T_0}^{T_1} \sum_{i=1}^{M} e_i \left(-\frac{\mathrm{d}\phi_i}{\mathrm{d}t} + \sum_{j=1}^{M} A_{i,j}\phi_j \right) \mathrm{d}t. \quad [\textit{by } (6.5.26)] \end{aligned}$$

We may thus proceed as before, and so

$$\left| \int_{T_0}^{T_1} e_m \,\mathrm{d}t \right| \le \sum_{k=1}^{N} \sum_{i=1}^{M} \mathcal{E}_{i,k} + \int_{T_0}^{T_1} \left| \sum_{i=1}^{M} \phi_i \mathcal{F}_i \right| \mathrm{d}t \qquad (6.5.28)$$

with $\mathcal{E}$ and $\mathcal{F}$ defined as in equations (6.5.14) & (6.5.15).

6.5.7 Summary of assumptions made

We have made various assumptions during the derivation of the above error bounds, and for ease of reference we summarise them here.

1. The true solution u is continuous on $[T_0, T_1]$, the simulation interval, and $\mathrm{d}\boldsymbol{u}/\mathrm{d}t$ exists here. This assumption is made implicitly by the model.

2. If $\mathbf{U}$ is constant on I_j, then $f_i(\mathbf{U}, t)$ and $\hat{f}_i(\mathbf{U}, t)$ are also. This requires that any stimulus is represented by a step function, which changes only on element boundaries.

3. The finite element solution $\hat{\mathbf{U}}$ is accurate enough for us to approximate the matrix A using this solution.

4. The linear finite element solution for ϕ is of sufficiently better accuracy than $\hat{\mathbf{U}}$ that we can assume it is the true solution.

5. For the L_2 norm error bound only, we also assume that the error is approximately constant for the duration of the simulation.

6.6 Results

The *a posteriori* error analyses described above have been implemented in Matlab. PyCml is capable of generating Matlab code usable with this analysis framework, and so we have been able to apply the analysis to a variety of cell models. The results of this are shown primarily in Section 6.6.3. We also consider the questions of verifying the accuracy of our implementation of the analyses (Section 6.6.1), evaluating the effect of models pathologically unsuited to lookup tables (Section 6.6.4), and the convergence of the various error bounds as the lookup table step size is reduced (Section 6.6.5).

6.6.1 Verification of analysis implementation

Two approaches have been taken in verifying that our Matlab implementation of the error analyses above accurately reflects the theory.

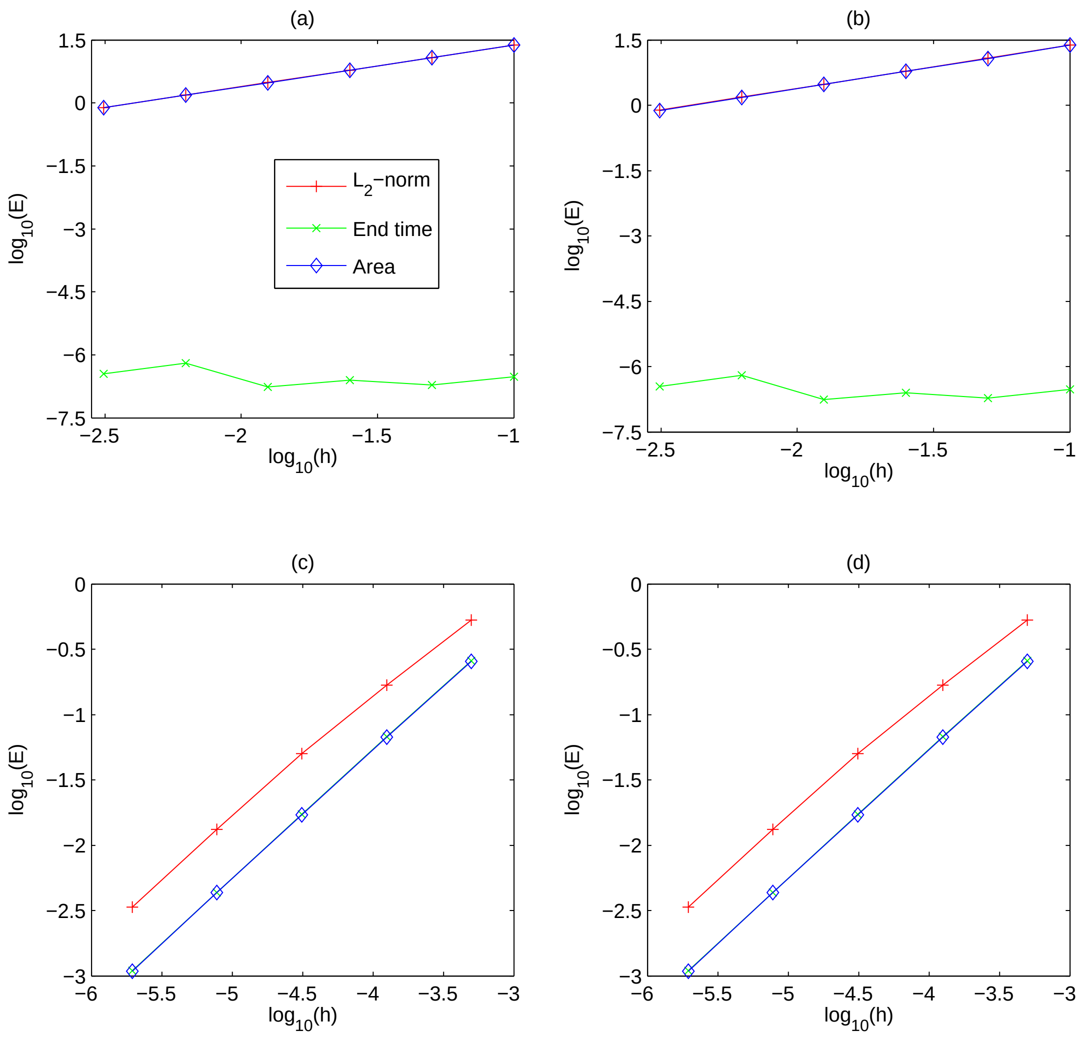
Firstly, we have solved both the model ODE systems and the dual problem using Matlab's own ODE solvers, in order to verify the correctness of our finite element codes.

Secondly, we have looked at the convergence of the error bounds as the mesh size h is varied. Theoretically, when not using lookup tables the convergence of the error in the L_2 norm is $O(h)$ (Eriksson et al., 1996, p. 224), as is the convergence in the error in any linear functional, since backward Euler is an $O(h)$ method (Lambert, 1973). We have checked this for a sample of models, and the results are shown in Figure 6.2 (plots (b) and (d)). The models chosen are relatively simple, in order that computing the error bounds could be undertaken in a reasonable amount of time even on the smallest mesh, but are still illustrative of typical behaviour.

Note that in most cases the gradient of the plots is 1, indicating $O(h)$ convergence. The 'end time' error bound for the Hodgkin–Huxley model, however, does not converge. This is because the nerve action potential is much shorter than that in cardiac cells, and hence for much of the duration of the simulation being analysed the model is in a steady state. The magnitude of the end time error is thus very small, on the order of the tolerance used by the nonlinear solver in our FEM implementation, and hence the FEM solution is incapable of resolving to a higher accuracy. If a tighter tolerance is used, convergence of this bound matches the rest.

Another point to note is that the L_2 norm and area error bounds always have practically the same value. This is due to the nature of cardiac ionic models. Both error bounds are defined as integrals over the solution, with the area bound only considering the transmembrane potential, whereas the L_2 norm involves all components of the state variable vector. However, with the units typically used in these models, the transmembrane potential is significantly larger than the other state variables, and thus dominates the L_2 norm. This would naturally suggest using a weighted norm instead of the L_2 norm, and the analysis could be adjusted to do so. We have

Figure 6.2 Log-log plot of convergence in the contribution of $\mathcal{E}$ term to the error bound as the mesh size h decreases. The gradients indicate $O(h)$ convergence. The upper plots use the Hodgkin–Huxley squid axon model (Hodgkin and Huxley, 1952); the lower plots use an early Noble model (Noble, 1962). For the plots on the right lookup tables were not used; the $\mathcal{E}$ term is thus the only contribution to the error bound in this case. Note that the lines for the L_2 norm and area error bounds are superimposed.



not done so, however, since this would not address further issues with the L_2 norm error bound described in Section 6.6.3.

6.6.2 The meaning of the error bound contributions

Recall that each error bound is made up of contributions from two terms, $\mathcal{E}$ and $\mathcal{F}$, using the formula

$$E + F := \sum_{k=1}^{N} \sum_{i=1}^{M} \mathcal{E}_{i,k} + \int_{T_0}^{T_1} \left| \sum_{i=1}^{M} \phi_i \mathcal{F}_i \right| \, \mathrm{d}t$$

where $\mathcal{E}$ and $\mathcal{F}$ are defined by

$$\begin{aligned} \mathcal{E}_{i,k} &= \int_{t_{k-1}}^{t_k} \left| (\phi_i - \bar{\phi}_i) \hat{f}_i(\hat{\mathbf{U}}, t) \right| \, \mathrm{d}t + \left| (\phi_i(t_{k-1}^+) - \bar{\phi}_i(t_{k-1}^+))[\hat{U}_i^{(k-1)}] \right| \\ \mathcal{F}_i &= \max \left| \hat{f}_i(\hat{\mathbf{U}}, t) - f_i(\hat{\mathbf{U}}, t) \right| . \end{aligned}$$

From this it can be seen that the most direct contribution of lookup tables to the error bound is through $\mathcal{F}$. If lookup tables are not used (i.e. we set $\hat{\mathbf{f}} = \mathbf{f}$) then this term is zero. The $\mathcal{E}$ term is indirectly influenced by the use of lookup tables, since $\hat{\mathbf{f}}$ occurs both directly and is used to determine ϕ. However, as can be seen by comparing the left and right plots in Figure 6.2, the effect on the error bound via this route is minimal (at least assuming that lookup tables are not a poor approximation): for these models the results are practically identical.

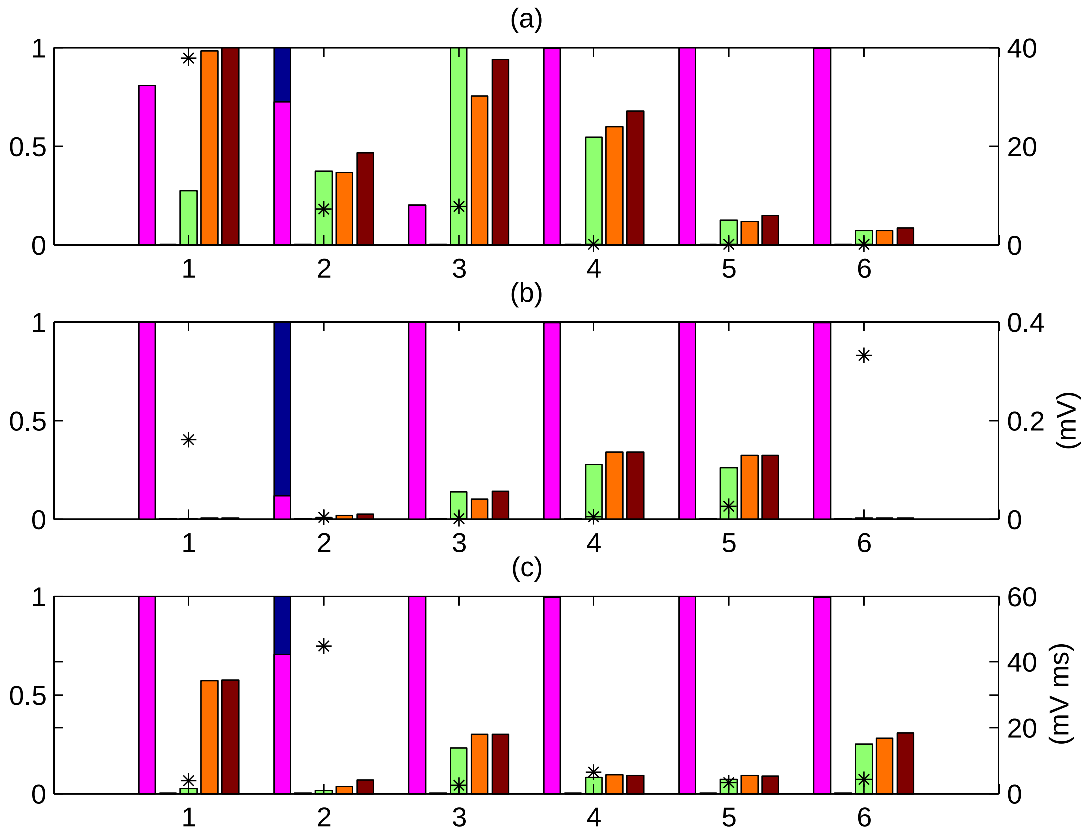
Roughly speaking, therefore, we can consider the $\mathcal{E}$ term to represent the error due to the particular ODE solver used (Backward Euler in this case), and the $\mathcal{F}$ term gives the additional error due to the use of lookup tables.

6.6.3 Application to various cell models

The error analyses above have been applied to a selection of cell models, and the results are presented graphically in Figure 6.3.

In considering the suitability of each error bound, we need to know how tightly it bounds the actual error. However, we typically do not have an analytic solution for the ODE systems

Figure 6.3 Plots of error bounds along with actual error measures for various models and bound types. Plot (a) measures error in the L_2 norm, plot (b) gives the error at the end time, and (c) uses the area measure. For each model, the left bar gives the error bound, with the contribution of the $\mathcal{E}$ term at the bottom and coloured magenta, and that of the $\mathcal{F}$ term at the top shown in dark blue. The other 4 bars represent the 'actual error' obtained using different ways of approximating the true solution: FEM, FEM on a finer mesh, Matlab's ode45 solver, and ode45 on a finer mesh. The first of these (FEM) often is not apparent due to its low comparative magnitude. The bars are normalised in height by model, such that the maximum error for each model (whether the error bound or an actual error) is 1. The value used for normalisation is indicated by an asterisk, with the relevant axes on the right. The models are taken from (1) Bondarenko et al. 2004; (2) Fox et al. 2002; (3) Hodgkin and Huxley 1952; (4) Noble 1962; (5) Noble and Noble 1984; (6) Zhang et al. 2000. The time mesh step size was $0.01\,\mathrm{ms}$, except for model 1 which used $0.005\,\mathrm{ms}$. A mesh 5 times finer was used in approximating the true solution. The end time for each model was chosen to be shortly after the cell had returned to its resting potential, where possible. A lookup table step of $0.01\,\mathrm{mV}$ was used in all cases.



modelling cardiac cells, and thus the actual error is unknown. We must instead obtain a better approximation than $\hat{\mathbf{U}}$ to the true solution u and use that to compute the 'actual error.' There are many approaches one could take, and four of them are compared in Figure 6.3.

To aid in reading this figure, we describe the first two groups of bars verbally here. These both address the error in the L_2 norm. The first group uses the model from Bondarenko et al. (2004). In this case the error bound obtained is 31, and the contribution of the $\mathcal{F}$ term is negligible. The last two estimates of the actual error exceed the error bound, being 37 and 38 respectively, and this latter value is used for normalisation, hence the position of the asterisk; all bars in the group are scaled accordingly. The first measure of the actual error is on the order of 10^{-5} and so a bar is not visible; the second measure gives the error as 10.

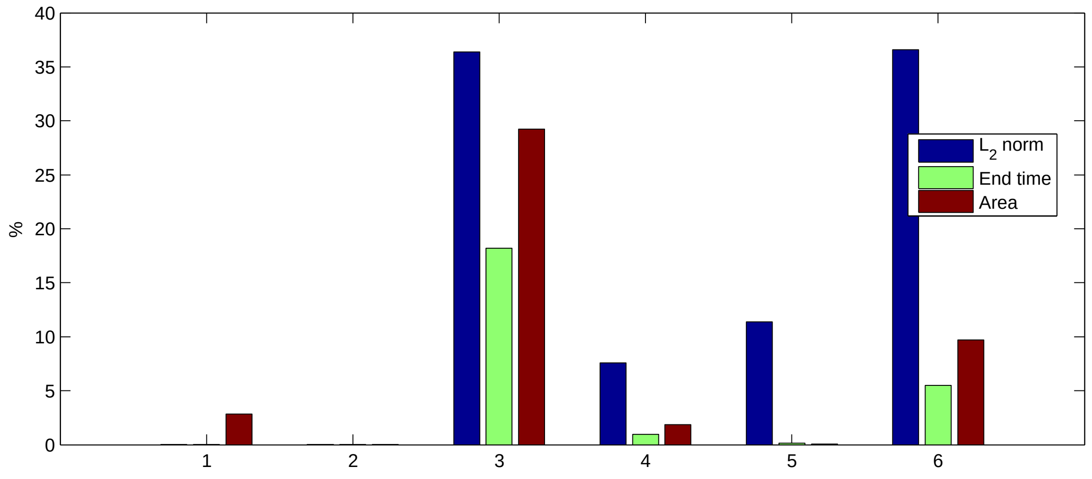
The second group of bars uses the Fox et al. (2002) model. In this case the error bound is 7, with the $\mathcal{F}$ term contributing roughly one third of this value. This is also larger than any of the actual error estimates, so this is used for normalisation. Again the first measure of the actual error is on the order of 10^{-5} and so does not show up. The other measures all give an actual error of approximately 3, and so are normalised to around 0.36–0.46.

As we saw in these two examples, the first approximation to the true solution typically underestimates the actual error by a considerable margin. It simply solves using the same method and mesh, but without using lookup tables, i.e. we use $\mathbf{U}$ to approximate u. It thus captures the additional error due to the use of lookup tables well, and in this regard is compared with the $\mathcal{F}$ contribution in Figure 6.4—we see that the contribution of the $\mathcal{F}$ term always exceeds this error measure. However, it ignores any error due to the ODE solver used, and is thus not a good choice for the 'true solution'.

A better method of approximating the true solution is to assume that the ODE solver will converge to the true solution as the mesh is refined, and hence to solve on a finer mesh to determine u (again, without using lookup tables). For comparison purposes, we can also use a different ODE solver, either on the original mesh or a refined mesh.

Note that both the L_2 norm and the area error measures involve computing an integral of

Figure 6.4 A comparison of the difference between $\mathbf{U}$ and $\hat{\mathbf{U}}$ with the contribution of $\mathcal{F}$ to the error bound. The bars show $|\mathbf{U} - \hat{\mathbf{U}}|$ as a percentage of the $\mathcal{F}$ contribution for each error bound type. Models are as in Figure 6.3.



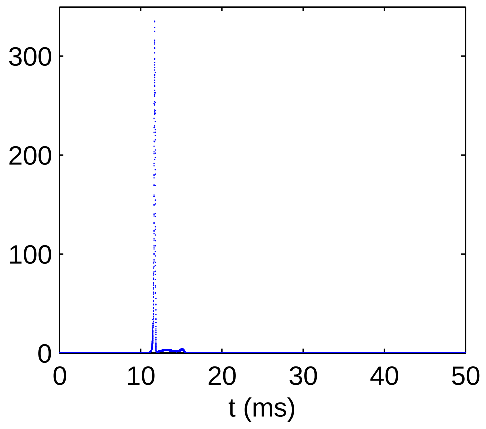
the solution. Where the dG(0) solver is used, the solution is a piecewise constant function, and computing an exact integral is straightforward. When using Matlab's ODE solvers, however, we do not have a functional form for the solution, and thus use Simpson's rule to evaluate the integral.

There is no clear pattern allowing us to say definitively which 'true solution' is the best approximation for any given problem. We would consider using FEM on a finer mesh to be a good default choice, since it provides a better like-for-like comparison with $\hat{\mathbf{U}}$. Using a different ODE solver is more suited to examining the effect of the solution method than of lookup tables.

In most cases the error bound is within an order of magnitude of the actual error, and the contribution of the $\mathcal{F}$ term is negligible compared with the $\mathcal{E}$ term. We thus infer that the use of lookup tables (with a step size of $0.01\,\mathrm{mV}$) has a negligible impact on accuracy in these cases. There are several exceptions, however, which require further consideration.

For two of the models evaluated, the bound on the L_2 norm of the error is less than the actual L_2 norm, whichever 'true solution' we use. This indicates that at least one of the assumptions in the error analysis does not hold. It may be that $\hat{\mathbf{U}}$ is not 'sufficiently accurate' due to small time offsets during the upstroke of the AP. Looking at time-course plots for the Hodgkin–Huxley

Figure 6.5 Plot of the integrand used to compute the L_2 norm of the actual error for the Hodgkin–Huxley model, using FEM on a finer mesh to approximate the true solution.



model confirms that these offsets occur, and this causes significant fluctuation in the actual error when evaluating convergence.

On the other hand, since this assumption underlies all the error bounds, we would expect the other bounds to also be affected if it did not hold, which is not the case. An assumption only used for the L_2 norm is that the error is approximately constant over the whole time mesh. Since the upstroke is much harder to resolve accurately than the rest of the simulation, it is quite likely that this assumption does not hold. Plotting the integrand against time (Figure 6.5) shows that this is indeed the case. Furthermore, similar spikes (albeit of smaller amplitude) are seen with the other models, indicating that it may be unsafe to rely on the L_2 error bound for any of them. Such spikes also appear in the $\mathcal{E}$ term of the error bound, which would indicate to an adaptive FEM algorithm that those elements should be refined (i.e. a smaller timestep used) and the simulation repeated.

Also, it is hard to relate the L_2 norm to anything of physiological significance. It is defined as the square root of the integral over time of the sum of the squared absolute error in each component of the ODE system. This is not a quantity which lends itself to an intuitive understanding, and it cannot be given in physical units. The other error bounds, by contrast, have clear physiological meanings, as well as having well-defined units.

Note the relatively poor performance of the 'end time' bound on the Zhang *et al.* model.

This is because it models the sino-atrial node, and is hence self-cycling, and never remains at a resting potential. The end point used in Figure 6.3 is actually on the upstroke of the second AP, for dramatic effect, but other points show a similar effect. The 1984 Noble model is also of the SAN. For this model we chose as the end time a brief period after an AP in which V varies little. While other state variables show considerable variation at this point in time, the bound is still relatively good, illustrating the importance of choosing the end time carefully.

The Bondarenko *et al.* model is very detailed, and hence requires a finer time step in order to correctly capture its behaviour in a simulation. Despite the fact that it is not a SAN model, the end time error bound is still unusually large. Looking in more detail at the results from this simulation, we see that while the end time chosen is some time after V returns to rest, the calcium concentrations are still varying considerably. This could well have affected the error bound.

The Fox *et al.* model shows a large contribution of the $\mathcal{F}$ term to the error bound. The reason for this is explored further in Figure 6.6. From this we see that, as expected, the large contribution of the $\mathcal{F}$ term is indicative of the lookup table step size chosen being too large for this particular model, and reducing τ reduces the error by reducing the $\mathcal{F}$ term. Reducing h, on the other hand, reduces the overall error by reducing the absolute contribution of both $\mathcal{E}$ and $\mathcal{F}$, but the relative contribution of $\mathcal{F}$ is still high.

Computation of the Jacobian matrix

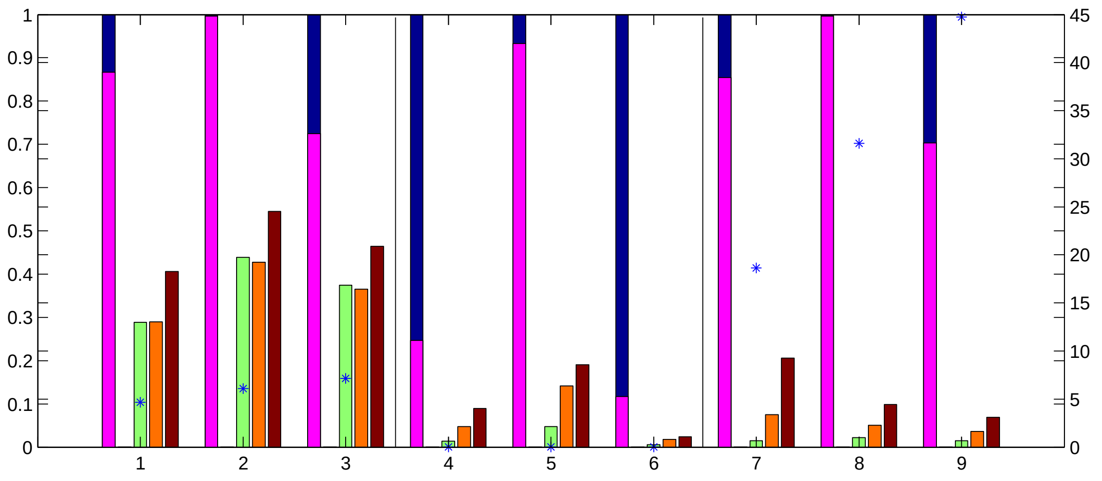
In computing the error bounds we approximate the matrix A in the dual problem, given by (6.5.12), by the transpose of the Jacobian matrix of the ODE system:

$$A_{i,j} = \frac{\partial f_j}{\partial u_i}.$$

By using Maple[5] to perform symbolic differentiation, we are able to automatically compute an analytic form for this matrix, and this was used in calculating many of the results shown above. This approach does not work for all models at present—Maple is unable to differentiate certain

[5]`http://www.maplesoft.com/Products/Maple/`

Figure 6.6 Error bound plots for the Fox *et al.* model (Fox et al., 2002). The first group of 3 sets of bars shows the error in the L_2 norm, the next end time, and the last area. In each group the first set uses $h = 0.005\,\mathrm{ms}$, the middle set uses $\tau = 0.001\,\mathrm{mV}$, and the last set uses the settings from Figure 6.3.



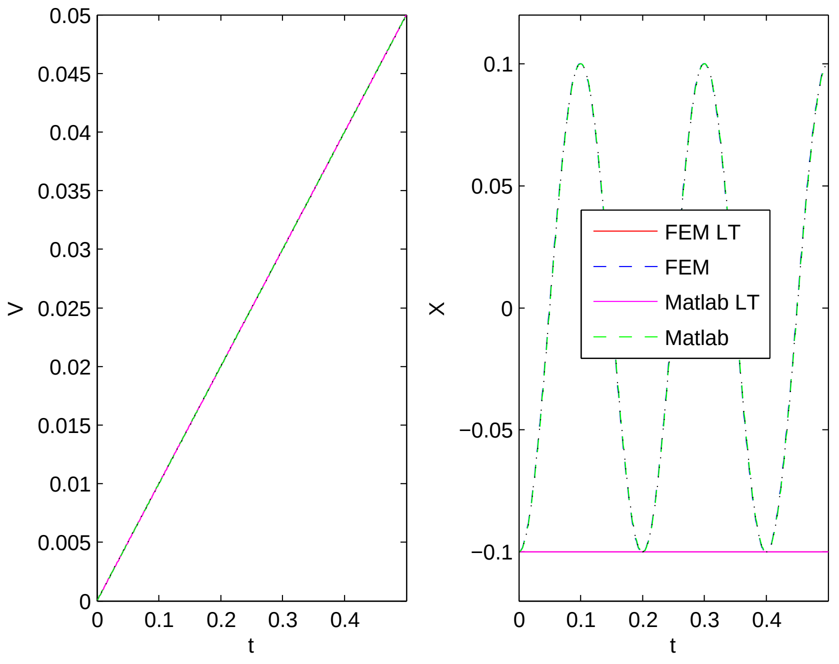
complex piecewise expressions, notably when the variable with respect to which we perform the differentiation is included within the test for one or more cases, and we have not yet determined a workaround for this. However, using a numerical approximation to the Jacobian matrix still gives good results, as can be seen with the 1984 Noble model.

6.6.4 Pathological cases

One of the assumptions made in computing the error bounds is that the finite element solution to the model ODE system is sufficiently accurate for us to approximate the matrix A using this solution when solving the dual problem. If lookup tables introduce a significant error then this assumption is no longer necessarily valid. An important question is therefore what the effect of this incorrect assumption will be: in particular, will the error bound produced be smaller than the actual error, or larger?

As a further verification of this method of error analysis, therefore, we have invented 'models' which will exhibit a significant error when lookup tables are used. These models contain expressions which appear to be suitable for replacement by lookup tables, but vary significantly on the scale of a typical lookup table step size, hence the use of linear interpolation is a poor

Figure 6.7 Simulation of our pathological model. The left plot shows V, whilst the right plot shows X. The use of lookup tables introduces a significant error in the latter. The mesh size h was 0.001, with $\tau = 0.01$.



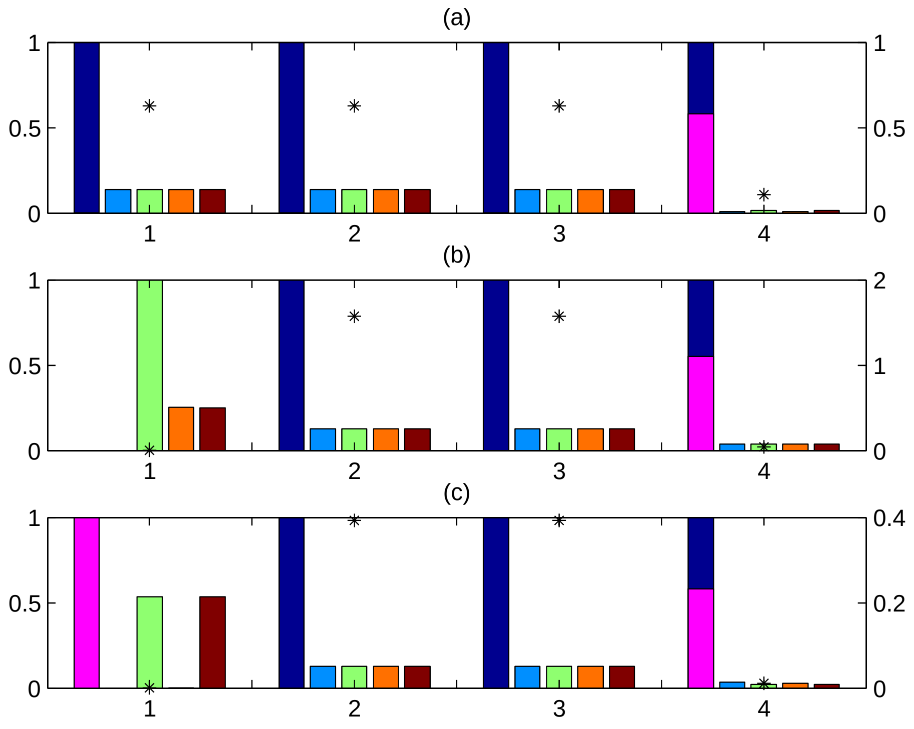
approximation.

One such model is given by

$$
\begin{aligned}
\frac{\mathrm{d}V}{\mathrm{d}t} &= 0.1, \\
\frac{\mathrm{d}X}{\mathrm{d}t} &= 10\pi \sin(100\pi V)\frac{\mathrm{d}V}{\mathrm{d}t}, \\
V(0) &= 0, \\
X(0) &= -0.1.
\end{aligned}
$$

Our naive analysis of where lookup tables may be used considers the term $10\pi \sin(100\pi V)$ in the second ODE a suitable candidate, since it only depends on the value of the transmembrane potential, and contains an expensive to compute trigonometric function. However, using a table step size of 0.01 to interpolate V results in this term being approximated by a constant, thus introducing an error in the term with a maximum magnitude of 10π. Furthermore, due to the placing of lookup table entries, this constant value is 0, hence no variation in X occurs. The

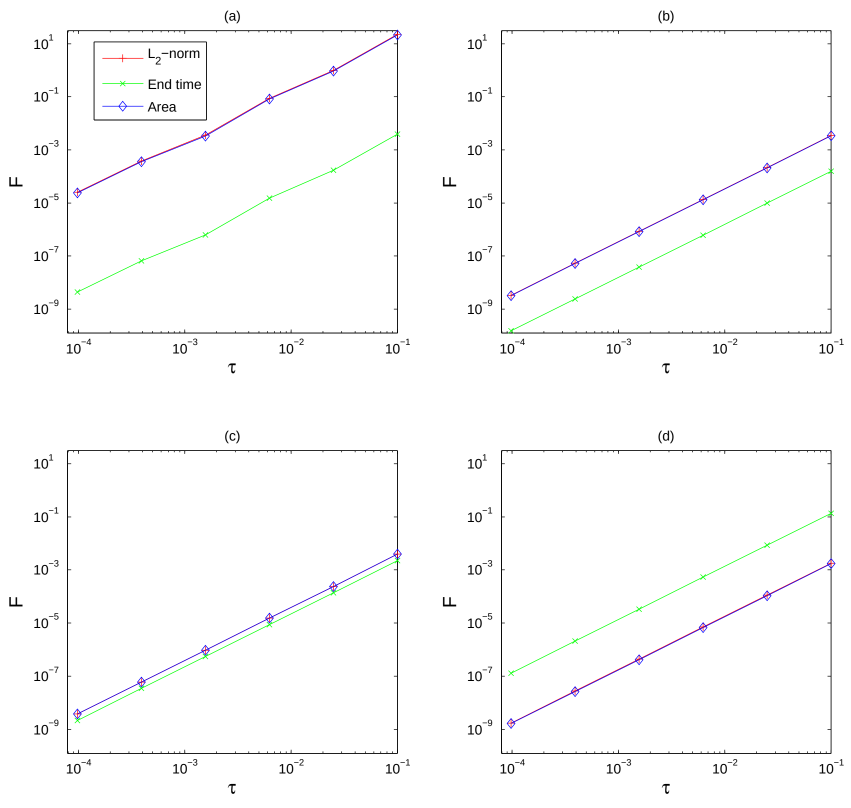
Figure 6.8 Error bound plots for the pathological model. Plots (a)–(c) show error in the L_2 norm, end time, and area, respectively. As in Figure 6.3, the bars are grouped in fives, showing the error bound and four measures of the actual error. For each model, the left bar gives the error bound, with the contribution of the $\mathcal{E}$ term at the bottom and coloured magenta, and that of the $\mathcal{F}$ term at the top shown in dark blue. In each plot, groups 1 and 2 both use $h = 0.001$ and $\tau = 0.01$, and show the error in V and X respectively. Groups 3 and 4 both show the error in X, with 3 reducing the mesh spacing ($h = 0.0001$) and 4 using finer lookup tables ($\tau = 0.001$). Each group is normalised such that the highest bar in the group has height 1; the value used for normalisation is indicated by an asterisk, with the relevant axes on the right.



effect of this is shown in Figure 6.7.

Figure 6.8 presents the error bounds for this pathological model. The use of lookup tables introduces a significant error in X, but no error in V, and this is reflected in the difference between the groups of bars labelled 1 and 2. The L_2 norm is identical in both cases, since this takes all variables into account, but in the case of V the other error measures are miniscule, on the order of discretisation error or less; this can be seen by comparing the position of the asterisks, which indicate what value was used for normalisation. Also, the contribution of $\mathcal{F}$ to the error bound is zero. For X, on the other hand, the error bound is dominated by $\mathcal{F}$ (as is the L_2 norm bound). Happily, the bound is still larger than any measure of the actual error.

Figure 6.9 Log-log plots of convergence in the $\mathcal{F}$ term in the error bound as the lookup table step size τ decreases. The gradients are each 2, showing $O(\tau^2)$ convergence. Models are taken from: (a) Fox et al. 2002; (b) Hodgkin and Huxley 1952; (c) Noble 1962; (d) Garny et al. 2003a. A mesh size h of 0.1 milliseconds was used, except for (b) which used $0.01\,\mathrm{ms}$.



Reducing the time mesh spacing, as in the third group of bars, has negligible effect. In order to reduce both the error and the bound, τ must be reduced so that the model is no longer pathological. The results are seen in the fourth group of bars: more than half the error bound is now due to $\mathcal{E}$.

In Section 6.7 we consider the question of whether we can perform some analysis of the model to automatically choose τ such that each function replaced by a lookup table looks linear everywhere on the scale of τ, in order to ensure that we do not have pathological behaviour.

6.6.5 Convergence

We can use the error bounds presented here to investigate the effect the lookup table step size has on the overall simulation error. Changing this step size, τ, influences the error bound through the $\mathcal{F}$ term. As can be seen from Figure 6.9 this term converges as $O(\tau^2)$ in all the error bounds we have considered, for these four models at least.

We can intuitively explain this behaviour by considering the Taylor series expansion of a function $g(V_m)$ that represents a subexpression of $\mathbf{f}$ which is replaced by a lookup table. Suppose the solution of the ODE system requires evaluating $g(a)$, where a lies between the lookup table entries v_0 and $v_0 + \tau$. Using linear interpolation between these lookup table entries, $\hat{g}(a)$ is given by

$$\hat{g}(a) = g(v_0) + \frac{a - v_0}{\tau}(g(v_0 + \tau) - g(v_0)).$$

By Taylor's theorem,

$$g(a) = g(v_0) + \frac{g'(v_0)}{1!}(a - v_0) + \frac{g''(v_0)}{2!}(a - v_0)^2 + R_2,$$

and so, approximating g' using forward differences,

$$g(a) - \hat{g}(a) \approx \frac{g''(v_0)}{2!}(a - v_0)^2 + R_2.$$

Now $0 \leq a - v_0 < \tau$, so assuming that g is sufficiently differentiable and τ is small, the remainder term R_2 is $O(\tau^3)$ and the error introduced by the linear interpolation is $g(a) - \hat{g}(a) = O((a - v_0)^2) = O(\tau^2)$.

In typical cardiac models, terms replaced by lookup tables occur as simple factors within $\mathbf{f}$, i.e. they are only ever raised to positive powers. The overall error introduced is thus at worst $O(\tau^2)$, as we have seen.

6.7 Discussion

We have seen that using *a posteriori* error analysis allows us to compute a bound on the error involved in simulating a cardiac cell using lookup tables. While the bound produced is not especially tight, typically an order of magnitude above more reliable measures of the actual error, examination of the terms in the bound does still have diagnostic power, and the nature of the bounds themselves can give insight into the behaviour of cardiac models.

The relative contributions of the $\mathcal{E}$ and $\mathcal{F}$ terms to the error bound can be used to determine whether the lookup tables have been made sufficiently fine. This was illustrated with both the Fox *et al.* model and our pathological model, and explained briefly in Section 6.6.2. If the contribution of $\mathcal{F}$ is large, this indicates that the lookup tables are contributing significantly to the overall error, and thus τ should be reduced. That this is a correct diagnosis can be verified by reducing τ and h independently; the former will reduce the relative contribution of $\mathcal{F}$, while the latter will not. If, on the other hand, the error bound is large but $\mathcal{F}$ does not contribute significantly, then it is not worth reducing τ: a smaller time step should be used instead.

This provides us with a 'trial and error' procedure for choosing a suitable step size. Can we instead automate a different analysis of the model to determine τ *a priori*? For each expression eligible for replacement by a lookup table, we would need to be able to compute a measure of its curvature, in order to determine how closely the curve is approximated by a straight line over intervals of length τ. Given a suitability threshold, we could then in principle use this to choose τ. Recall that $\mathcal{F}$ is defined by

$$\mathcal{F}_i = \max \left| \hat{f}_i(\hat{\mathbf{U}}, t) - f_i(\hat{\mathbf{U}}, t) \right|$$

and so may be computed without a simulation being run, if the maximum can be determined analytically. This also requires both computing second derivatives of the function with respect to the transmembrane potential, and analysing the continuity of the function. We believe this to be possible in theory for many common functions, but not trivial, and it is likely to be a computationally intensive process. In our view the approach presented here is easier to implement and use, especially since few real-world models are pathological.

The different error bounds provide different views on the behaviour of the model being analysed. The L_2 norm is commonly considered to provide a good overall measure. However, as we have seen, it is not ideal for these problems. The assumption that the error is approximately constant does not hold, and thus the bound is not reliable. It also does not relate to the underlying physiology. Another point is that the different state variables in the models have very different typical magnitudes. Without suitable weighting, therefore, a few state variables (especially the transmembrane potential) will dominate the L_2 norm.

A significant advantage of *a posteriori* error analysis in this context is that it allows us to consider the error in functionals of the solution. We have presented two examples of this. Analysing the error at the end time is important if we wish to consider long running simulations. It would be computationally infeasible to run an error analysis on the whole simulation, but this is unnecessary due to the cyclical nature of the system, reflected in the very small error at the end time (measured during the diastolic interval when the cell is 'at rest').

The area under the curve of the transmembrane potential, while not usually of interest to physiologists, can still provide interesting information. When coupled with other measures, we could use it as an indicator that the shape of the action potential is not greatly altered through the use of lookup tables.

When lookup tables have been used in the literature, the step size chosen has typically been $0.01\,\mathrm{mV}$. From our results presented here, including the convergence results of Figure 6.9, this appears to be a sensible choice to make in the absence of any indication to the contrary.

Finally, it must be noted that the analysis we have presented only applies directly if the cell model is simulated using the finite element method (in particular, dG(0)). What, then, can we say about the behaviour of other solvers, since backward Euler is rarely used? Firstly, backward Euler is a low order solver—only $O(h)$. Assuming that a solver is operating within the bounds of numerical stability, it is thus unlikely to perform worse for a given mesh size. There is thus a good chance that the same error bound will hold.

A more important point, however, is that our recipe presented above for choosing τ is still rather qualitative, since the error bounds we have obtained are not always very tight. Since we can (roughly) consider the $\mathcal{F}$ term as representing the contribution of lookup tables to the error, whilst the $\mathcal{E}$ term represents the contribution of the ODE solver, if the $\mathcal{F}$ term is insignificant then lookup tables will not dominate the error whatever ODE solver is used.

We have now seen how the lookup table optimisation may be applied to CellML models, and shown that the error introduced by the approximations involved may be controlled. In the next chapter, we will apply this optimisation technique to a sample of models, and thus investigate its effectiveness in practice.

7 Experimental Results

In Chapters 5 and 6 we have presented two optimisation techniques for CellML models, and proved that they are correct in that they do not significantly alter the results of simulations. We now consider the question of effectiveness: how much improvement in simulation speed do these techniques provide?

This chapter will contain two approaches to answering the question. On the one hand it will present experimental results of applying the optimisations to the range of models introduced in Chapter 1, showing the speed increases produced by each technique individually as well as in combination (Section 7.2). It will also present a method for approximating the speedup without actually simulating the model (Section 7.1). The results from both approaches will be used to discuss why we see the results that we do, notably in Section 7.2.1.

The experimental setup and framework for applying the optimisations will be explained. A key point to note is the synergy between the two techniques used: when PE is performed before LT the combined effect is greater than the product of the effects of each optimisation in isolation. The use of PE makes the lookup table analysis more effective.

7.1 Estimating the potential speedup

One question that users of transformation tools such as ours may have is ‘how effective will this be for my model?’ This is a difficult question to answer precisely, as there are many factors which influence the actual speed increase observed and are difficult to account for theoretically. For instance, the machine used to run experiments can have a large impact, with concerns such as pipelining, or cache utilisation of table lookups affecting the results. Furthermore, the speedup experienced in a given simulation environment will depend on features of the environment used: the choice of ODE solver and its implementation, for example. In a tissue-level (as opposed to cell-level) simulation, e.g. of a whole heart, the choice of tissue model also has an effect, since simulating this portion of the combined model will take a certain proportion of the total time. We thus need to consider approximate answers, which at least allow us to say ‘this model will probably be somewhere between these models, and experience roughly this improvement in simulation time within this context.’

We assign a rough execution cost, or complexity, to each type of expression node, based on *a priori* expectations of relative costs and tuned with some experimental timing runs. These costs are shown in Table 7.1. This then allows us to estimate the execution cost of an entire model, both before and after we have performed PE. This is straightforward for all nodes except for piecewise expressions, where the real evaluation time will generally depend on run time data. Our estimate takes the sum of the complexities of the conditions and the maximum of the complexity of the case results; this gives us an upper bound on the complexity of the piecewise expression.

We can also account for the presence of lookup tables in our complexity estimate: where an expression can be replaced by a table, we may count its cost as that of linear interpolation on a table, by summing the costs of the elementary operations used to perform the interpolation.

One might consider using some form of parameter optimisation to adjust the costs listed in Table 7.1. An automatic approach to doing so is not fruitful, however. Since the complexity of

Table 7.1 Approximate execution costs for each type of expression node.

Node type	**Complexity estimate**
constants	0.5
variables	0.7
`divide`	15
`abs`	5
`floor, ceiling`	20
`power`	5 (x^2, x^3) or 30
`root`	30
trigonometric, `exp, log`	70
other operators	*number of operands − 1*
all containers	*sum cost of children*

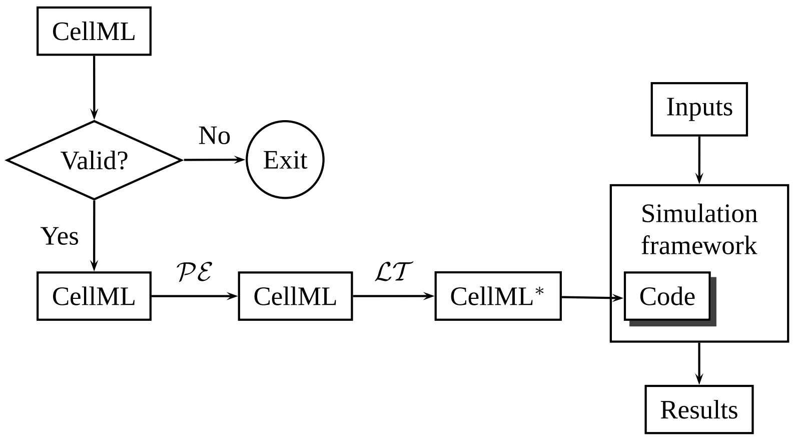
a model is a linear combination of node complexities, it is possible to use actual measured run times to set up a linear system, which may be solved to obtain costs for each class of node. The results thus obtained are not sensible—several of the parameters are negative, whichever set of models are used to provide run time data. This is due to the difficulties in producing an accurate complexity estimate described above—these mean that in reality the performance is not a linear combination of node costs. One could try to improve the situation by further subdivision of node classes, but this increases the likelihood that we would merely over-fit to the particular experimental setup used. We have, however, been able to use results from the experiments detailed in the next section to perform some manual adjustments of the costs listed.

Given this complexity estimate, we can then use it to estimate the speedups obtained by our optimisation techniques, by comparing the complexity measures obtained before and after. In the next section, results graphs will be ordered according to this estimate of the speedup. Section 7.2.1 will also use the complexity measure to analyse the factors underlying any simulation performance improvements.

7.2 Actual speedup results

There are several settings in which we wish to evaluate the effectiveness of our optimisations, in order to obtain some indication of how the speedup obtained might depend on context. Our first

Figure 7.1 The optimisation framework for CellML models (reproduced from Figure 1.1). $\mathcal{PE}$ and $\mathcal{LT}$ are the optimisation techniques described in Chapters 5 and 6 respectively. When both optimisations are combined, $\mathcal{PE}$ is performed before $\mathcal{LT}$ to exploit synergy, as explained in the text.



test setting will be followed in Section 7.2.1 by an analysis of the primary factors influencing performance, using the insights gained from Section 7.1. We will then present other settings and discuss how they affect the results.

In performing simulations of our sample of models, we use the Chaste simulation framework (Pitt-Francis et al., 2008), which we have helped to develop over the last three years. It consists primarily of C++ libraries (along with supporting infrastructure and tests), and provides a small selection of ODE solvers, as well as classes for solving the monodomain and bidomain equations using the finite element method. For each model, we use PyCml[1] to apply our optimisation techniques and generate C++ classes which integrate into Chaste. This process typically takes only a few seconds once a model has been loaded and validated; for no model in our sample does it take more than thirty seconds in total. Four versions of each model are generated: a control with no optimisation, one using just lookup tables,[2] one using just partial evaluation, and one with both optimisations applied as shown in Figure 7.1. In all simulation runs, full

[1]PyCml is our Python implementation of the optimisations; see Chapter 5 for discussion on this point.
[2]See also Section 6.2.

Table 7.2 Time step sizes used for simulations with Euler's method.

Model	**Euler time-step (ms)**
Bondarenko *et al.* 2004	0.0002
Faber and Rudy 2000	0.001
Fox *et al.* 2002	0.001
ten Tusscher and Panfilov 2006	0.001
others	0.01

compiler optimisation was used, so as to give a fairer comparison with real-life usage.

In each experiment, each type of simulation was performed three times, and the smallest run time of these was used to compare with other optimisation settings. Since any other activity on the machine used to perform simulations will *increase* the run time, taking the smallest run time gives us the best approximation to the situation where no interference occurs.[3]

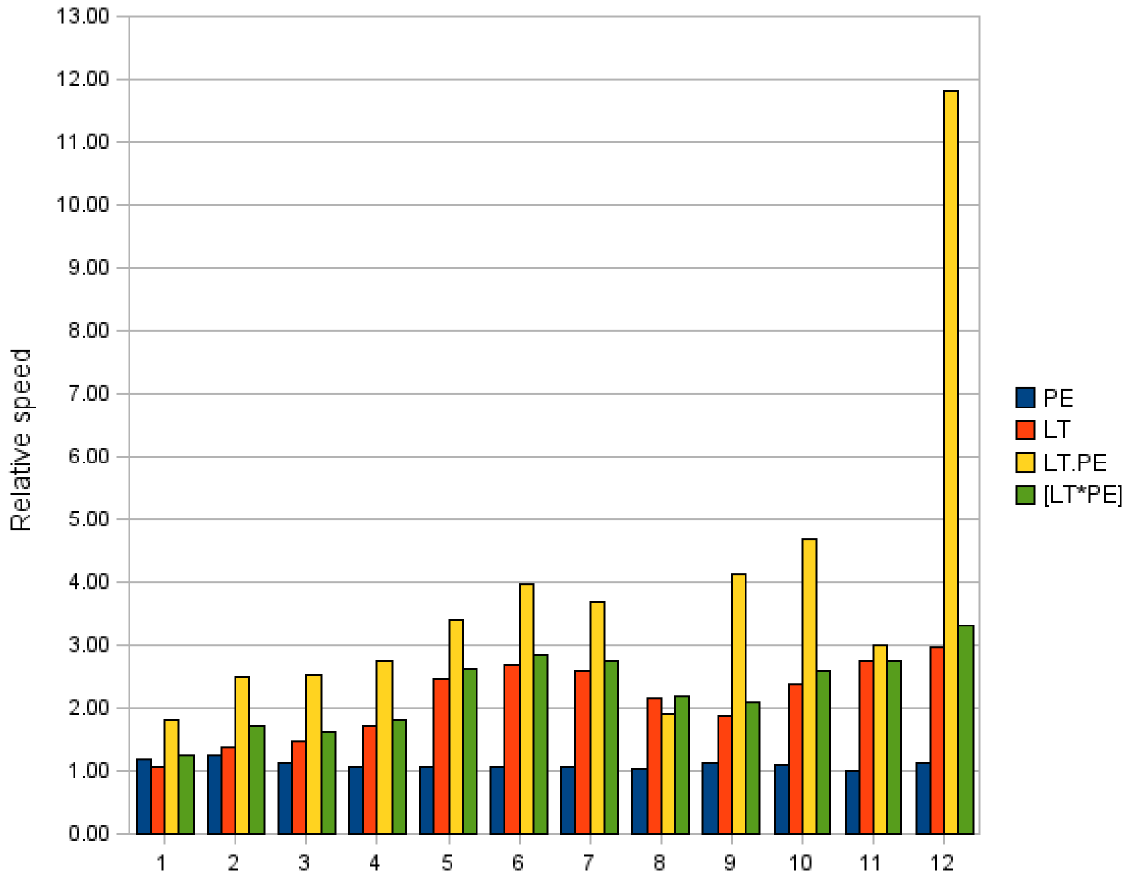
Figure 7.2 gives an indication of the results to be expected when studying a single cell in isolation. The run times recorded are for repeated short simulations of a single myocyte. Further details of the experimental setup are given in the figure caption. Full discussion of the results is contained in Section 7.2.1, but we note a few points here. Firstly, with a few exceptions that will be covered later, there is a correlation between our estimate of the speedup and the observed speedup, demonstrated by the fact that the actual speedup increases from left to right. Secondly, the speedup when $\mathcal{LT}$ and $\mathcal{PE}$ are combined is usually significantly greater than the product of the individual speedups. We will now explore the factors leading to this in more detail.

7.2.1 Analysis of results

In the results shown above we see that the use of partial evaluation alone leads to only modest improvements, typically no more than 10%. The translation to C++ does 'flatten' the CellML model, by placing all the mathematics within a single method. Hence we expect this to perform somewhat better than C++ code that retains the hierarchical structure. This sort of transformation (unfolding program structure) is also performed by partial evaluation, and so the effect of partial evaluation may be under-represented here. On the other hand, this flattening is a

[3]This idea is not original to me, but I cannot remember where I first read of it.

Figure 7.2 The effects of partial evaluation and lookup tables on run times. Each model was simulated using Euler's method with a step size of 0.01ms (except where a smaller step was required for stability; see Table 7.2), for 1 second of simulated time. These simulations were each run 25 times and the total time recorded. Each bank of simulations was repeated 3 times, and the fastest running time for each was used in calculating speeds relative to the normal model. All simulations used the GNU C++ compiler with full (processor-neutral) optimisation, and for all models the lookup table step size was set to $0.01\,\mathrm{mV}$. The LT.PE columns show the combined performance with both optimisations applied. For comparison, the [PE*LT] columns show the product of the individual LT and PE speedups; this illustrates the synergy obtained by combining the techniques. Models from left to right, ordered according to estimated LT.PE speedup, are: (1) Noble and Noble 1984; (2) Noble et al. 1998; (3) DiFrancesco and Noble 1985; (4) Faber and Rudy 2000; (5) Bondarenko et al. 2004; (6) ten Tusscher and Panfilov 2006; (7) Courtemanche et al. 1998; (8) Hodgkin and Huxley 1952; (9) Fox et al. 2002; (10) Luo and Rudy 1991; (11) Noble 1962; (12) Garny et al. 2003a.



very natural translation, and is used by other simulation software, so it is reasonable to compare against such flattened code. The main reason for a poor performance by partial evaluation is that the code implied by a CellML model is fairly simple—a CellML model consists only of mathematical expressions, without complex user-defined functions—thus PE is mainly performing constant folding, which is also done by the compiler. Hence we do not see the impressive improvements often obtained by PE in other contexts.

The main benefit of PE here is seen when it is combined with lookup tables. While the $\mathcal{LT}$ transformation by itself does provide a better speedup than $\mathcal{PE}$, it is the combination of the techniques that yields the most impressive results. The increase in efficiency when both optimisations are used is, in most cases, significantly larger than the product of the speedups obtained by either technique in isolation, indicating that the interaction of the techniques is favourable.

The analysis for when a lookup table may be used is simple: if the only variable within the expression is V, and 'expensive' functions are used, then we should use a lookup table. We could be more sophisticated, allowing other variables to appear in such expressions: constant variables, for example, or more generally those whose value can be determined statically. In other words, any expression which would be annotated as static by the BTA, were it not for the presence of V, is admissible. If partial evaluation is performed prior to the lookup table analysis, then static portions of such expressions will have been evaluated and replaced by constants, and the simple analysis we use annotates them for replacement by a table. This is the root cause of the synergy between these two optimisations.

The synergy is manifested in two factors which influence the speedup. In many cases the number of lookup tables used increases significantly after PE—more expressions are replaced by a lookup table. In others the mean complexity of expressions covered by a lookup table increases, without the number of tables necessarily increasing—bigger expressions are replaced. Details of these factors for each model in our sample can be found in Table 7.3.

The complexity figures are all calculated based on the unspecialised model, i.e. they give

Table 7.3 Indicators and influencers of model speedup, showing the number of lookup tables both before and after partial evaluation, as well as the mean complexity of expressions replaced by lookup tables. The final pair of columns show the proportion of the whole model that can be replaced by lookup tables. Expression complexities are all calculated from the original version of the expression, not the optimised version.

Model	Num. Tables		Mean Complexity		Complexity Proportion	
	Before	After	Before	After	Before PE	After PE
Hodgkin–Huxley 52	6	6	98	98	76%	76%
Noble 62	7	7	114	114	86%	86%
DiFrancesco–Noble 85	12	23	103	102	26%	49%
Noble–Noble 84	4	13	87	98	8%	30%
Luo–Rudy 91	14	16	150	153	66%	76%
Courtemanche *et al.* 98	33	31	133	167	60%	70%
Noble *et al.* 98	16	32	92	93	21%	42%
Zhang *et al.* 00	29	31	130	146	65%	77%
Faber–Rudy 00	24	39	141	125	41%	59%
Fox *et al.* 02	20	29	120	121	46%	68%
Bondarenko *et al.* 04	32	36	113	116	55%	63%
ten Tusscher–Panfilov 06	31	29	133	166	60%	70%

the complexity of expressions prior to PE being performed. Those figures headed 'After PE', however, count the complexity of those expressions which will be replaced by lookup tables after specialisation. This is done in order to obtain a better comparison—since PE simplifies expressions, evaluating complexity after PE gives a lower cost for many expressions.

Often both of these influencing factors are present within the same model, to varying extents. As can be seen in Table 7.3, many models show increases in both the number of lookup tables and the mean complexity of replaced expressions. In other models this is not so clear. For example, consider the Fox *et al.* model (Fox et al., 2002), and two expressions within it in particular:

$$i_{K_p} = g_{K_p} K_{p_V} (V - E_K), \tag{7.2.1}$$

$$K_{p_V} = \frac{1}{1 + e^{(7.488 - V)/5.98}}, \tag{7.2.2}$$

$$E_K = \frac{RT}{F} \log(K_o/K_i),$$

$K_o, K_i, g_{K_p}, R, T, F$ constant.

Prior to partial evaluation, only (7.2.2) may be replaced by a lookup table. However, since E_K and g_{K_p} are static, after partial evaluation they are replaced by their (constant) values. Also, the definition of K_{p_V} is instantiated within (7.2.1), and so the whole of (7.2.1) is deemed replaceable by our analysis, since the only variable it now contains is V and it contains an exponential function. There is still only one lookup table, but it replaces a larger expression, and thus is more effective.

Also, the number of lookup tables used increases by 9 after PE. One such new lookup table is within the calculation of i_{CaK}:

$$e^{\frac{VF}{RT}} - 1.$$

Prior to PE the presence of the variables F, R, and T remove this expression from consideration. Since they are constant, however, the expression after PE is simply $e^{V(96.5)(3.88\cdot 10^{-4})} - 1$ and hence is eligible for replacement.

Regardless of whether it is due to an increased number of lookup tables, larger expressions replaced by tables, or both, we see that the proportion (in terms of our complexity measure) of the model which may be replaced by lookup tables increases after partial evaluation, hence the effectiveness of using lookup tables is increased.

The best results are seen in the case of the Zhang et al. (2000) model. As mentioned in Section 2.2.8, the CellML encoding of this model by Garny *et al.* has a (static) parameter which selects different versions of the model. The model can also describe different cell types within the SAN (e.g. a central or peripheral cell) which have different properties, and hence different values for constants. There are also many computed variables whose values depend solely on these properties, which are thus able to be computed at partial evaluation time, and their constant values inserted in the optimised model.

Also, due primarily to the model version selection, the model contains many piecewise expressions selecting different versions of key expressions, where the choice can be made statically, i.e. by PE. Many of these are also within expressions that would otherwise by suitable for replacement by a lookup table. One such example is the equation for τ_r:

$$\tau_r = \begin{cases} 0.001\left(2.98 + \frac{15.59}{1.037e^{0.09(V+30.61)}+0.369e^{-0.12(V+23.84)}}\right), & Version = 0, \\ 0.0025\left(1.191 + \frac{7.838}{1.037e^{0.09012(V+30.61)}+0.369e^{-0.119(V+23.84)}}\right), & Version = 1, \\ 0.001\left(2.98 + \frac{15.59}{1.037e^{0.09012(V+30.61)}+0.369e^{-0.119(V+23.84)}}\right), & \text{otherwise.} \end{cases}$$

Before PE, the lookup table analysis considers that the result of each individual case may be replaced by a table, but the expression as a whole may not. After PE, the expression is replaced by just one of the cases, and the whole expression, possibly including nodes higher up the tree, may be converted to a lookup table. There is thus a high amount of synergy.

This model also demonstrates a side-effect of our complexity measure for piecewise expressions. Prior to PE there are several piecewise expressions for which the result of each case may become a lookup table (the equation above is one such). If each of these is counted separately, the model has 49 lookup tables in total. The figure 29, quoted in Table 7.3, is obtained by counting only 1 for each such piecewise expression, since at run-time only 1 table will actually be used, and hence contribute to any speed increase. After PE, there *is* only 1 table per expression, since the piecewise choice is made statically.

7.2.2 Unexpected results

If our estimation of the speedup described in Section 7.1 were suitable for all models, we would expect Figure 7.2 to show a steady increase in actual speedup from left to right. A few models, however, do not fit this pattern. In this section we attempt to explain why.

The speedup due to $\mathcal{PE}$ alone is consistently overestimated. Any good optimising compiler is capable of performing constant folding, and thus will do essentially the work done by the partial evaluator (albeit after any lookup tables analysis). The estimated speedup calculation, however, does not take compiler optimisation into consideration, and thus predicts an improvement due to PE which is not seen in practice.

The simplest cases to explain are the simplest models: the Hodgkin–Huxley equations and the 1962 Noble model. With both of these, exactly the same expressions are replaced by lookup tables both before and after partial evaluation (compare the optimised Hodgkin–Huxley model shown in Listing 7.1 with the original in Listing 2.3 for example). There is thus no synergy when optimising these models, and so inaccuracies in the complexity estimate are far more evident. This is also the main reason why, for the Hodgkin–Huxley equations, the product of the $\mathcal{LT}$ and $\mathcal{PE}$ speedups is greater than the combined speedup—in this case the compiler is less effective at optimising the model with lookup tables once $\mathcal{PE}$ has been performed.

The ten Tusscher and Courtemanche models are shown in the opposite order to that predicted by our complexity estimate. This is simply due to inaccuracies in the node execution costs assigned by the estimate. Both in estimated and actual speedup the two models show very similar behaviour, despite their differences physiologically, and so any error in the estimate is quite likely to lead to an incorrect ordering.

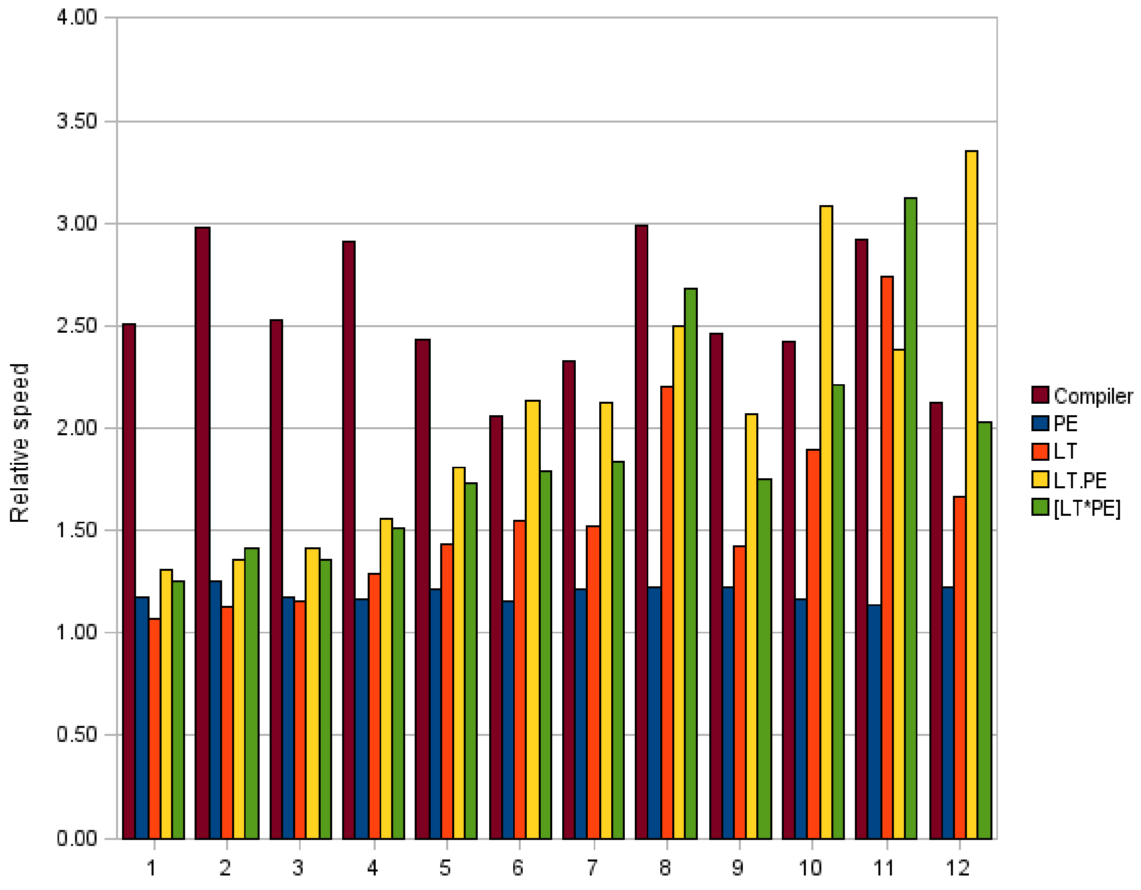
7.2.3 Other experimental settings

We now proceed to look at the effect of our optimisations in some different settings, to consider the influence of factors such as the C++ compiler or ODE solver used on the observed speedup. We also briefly consider multi-cellular simulations.

Figure 7.3 uses the same experimental setup as before, but a different compiler: the Intel C++ compiler. This optimises better for the advanced features of Pentium™ processors, and also uses a high performance mathematics library, which has faster routines for computing functions such as exponentials. These factors lead to normal run times 2 to 3 times faster than with the GNU compiler. Better compiler optimisation would also tend to reduce the accuracy of our complexity estimate, although the predicted ordering of models contains only the same errors as in Figure 7.2.

Curiously, the speedup obtained by PE on its own is better than was seen with the GNU compiler, whereas one would expect the opposite to be true. This is especially odd given that the optimisations performed by the Intel compiler also appear to interact with our techniques in such a way as to exacerbate the anomalies described in Section 7.2.2 for our two simplest cell

Figure 7.3 The effect of compiler choice on optimisation efficiency. All simulations used the Intel C++ compiler, optimised for Pentium™ processors; all other settings were as for Figure 7.2. The results are displayed as in Figure 7.2, with the addition of an extra bar for each model showing the speedup achieved on the normal model by using the different compiler.



Listing 7.1: The modified Hodgkin–Huxley equations of Listing 2.3 after $\mathcal{PE}$ and $\mathcal{LT}$ optimisations. Expressions annotated for replacement by a lookup table are shown bold and underlined.

```
def model hodgkin_huxley_1952_version07 as
    def unit millisecond from
        unit second {pref: milli};
    def unit per_millisecond from
        unit second {pref: milli, expo: −1};
    def unit millivolt from
        unit volt {pref: milli};
    def unit milliS_per_cm2 from
        unit siemens {pref: milli};
        unit metre   {pref: centi, expo: −2};
    def unit microF_per_cm2 from
        unit farad {pref: micro};
        unit metre {pref: centi, expo: −2};
    def unit microA_per_cm2 from
        unit ampere {pref: micro};
        unit metre  {pref: centi, expo: −2};
    def unit per_millivolt from
        unit volt {pref: milli, expo: −1, mult: 1};
    def unit centimetre2_per_microfarad from
        unit farad {pref: micro, expo: −1, mult: 1};
        unit metre {pref: centi, expo: 2, mult: 1};
    def comp c as
        var environment__time: millisecond;
        var membrane__V:       millivolt {init: −75};
        var membrane__i_Na:    microA_per_cm2;
        var membrane__i_K:     microA_per_cm2;
        var membrane__i_L:     microA_per_cm2;
        var membrane__i_Stim:  microA_per_cm2;
        var sodium_channel_m_gate__m:     dimensionless {init: 0.05};
        var sodium_channel_h_gate__h:     dimensionless {init: 0.6};
        var potassium_channel_n_gate__n:  dimensionless {init: 0.325};

        ode(membrane__V, environment__time) =
            −(−membrane__i_Stim+membrane__i_Na+membrane__i_K
              +membrane__i_L)*1{centimetre2_per_microfarad};
        membrane__i_Na = 120{milliS_per_cm2}
            *pow(sodium_channel_m_gate__m, 3{dimensionless})
            *sodium_channel_h_gate__h*(membrane__V−40{millivolt});
        ode(sodium_channel_m_gate__m, environment__time) =
            −0.1{per_millisecond}*(membrane__V+50{millivolt}) /
                (exp(−(membrane__V+50{millivolt})*0.1{dimensionless})
                    −1{dimensionless})
                *(1{dimensionless}−sodium_channel_m_gate__m)
            −4{per_millisecond}*exp(−(membrane__V+75{millivolt})
                                     *0.0555555555556{per_millivolt})
                *sodium_channel_m_gate__m;
        ode(sodium_channel_h_gate__h, environment__time) =
            0.07{per_millisecond}*exp(−(membrane__V+75{millivolt})
                                       *0.05{per_millivolt})
                *(1{dimensionless}−sodium_channel_h_gate__h)
        −1{per_millisecond}/(exp(−(membrane__V+45{millivolt})
                                          *0.1{dimensionless})
                                      +1{dimensionless})
            *sodium_channel_h_gate__h;
    membrane__i_K = 36{milliS_per_cm2}
        *pow(potassium_channel_n_gate__n, 4{dimensionless})
        *(membrane__V−−87{millivolt});
    ode(potassium_channel_n_gate__n, environment__time) =
        −0.01{per_millisecond}*(membrane__V+65{millivolt}) /
            (exp(−(membrane__V+65{millivolt})*0.1{dimensionless})
                −1{dimensionless})
            *(1{dimensionless}−potassium_channel_n_gate__n)
        −0.125{per_millisecond}*exp((membrane__V+75{millivolt})
                                          *0.0125{per_millivolt})
            *potassium_channel_n_gate__n;
    membrane__i_L = 0.3{milliS_per_cm2}*(membrane__V−
                                              −64.387{millivolt});

    membrane__i_Stim = sel
        case (environment__time ≥ 50{millisecond}) and
             (environment__time ≤ 50000{millisecond}) and
             (environment__time −50{millisecond}
               −floor((environment__time −50{millisecond})
                         *0.005{per_millisecond})
                 *200{millisecond} ≤ 0.5{millisecond}):
            20{microA_per_cm2};
        otherwise:
            0{microA_per_cm2};
```

models, suggesting that the addition of lookup tables reduces the effectiveness of this compiler on partially evaluated code. Without a detailed look at the machine code generated by the compiler, it is hard to determine why this is the case. Various changes to the structure of generated code have not affected these features in the results, so it would appear that the issue is fundamental to the interaction of the compiler with our partial evaluation. One possibility is that the Intel compiler is more effective than the GNU compiler at optimising the large statements that result from PE (i.e. the GNU compiler prefers the source code to contain more explicit temporary variables) but that the inclusion of method calls to perform table lookups adversely impacts this ability.

The Noble et al. (1998) model is also adversely affected by this compiler, with the product of the individual $\mathcal{LT}$ and $\mathcal{PE}$ speedups being greater than the combined speedup. It is possible that this is related to the unusually complex way in which many of the currents are expressed in the CellML file, using many nested divisions and n-ary multiplications. This code structure may impact the effectiveness of compiler optimisation.

We must also consider the impact of the ODE solver used. In all the simulations above, we have used Euler's method, which being the simplest ODE solver also gives the least overhead, meaning that actual evaluation of the cell model provides as high a proportion of the run time as possible. However, due to the multiscale nature of the cell models the ODE system is stiff, and hence explicit ODE solvers may be unstable; Euler's method thus requires the use of a small time step, and so is not very efficient.

The CVODE library[4] includes adaptive methods for solving stiff systems of ODEs, and has been used for simulations of single cardiac cells; it is used by COR, for example. PyCml includes a code generator targeting this library, which we have used to obtain the results shown in Figure 7.4.

An extra bar in this figure demonstrates the speedup obtained from using the better ODE solver, which can be quite pronounced, especially in the case of the Bondarenko et al. (2004) model (model 5). This is because the model is especially stiff, and hence required a much smaller time step when solved using Euler's method (see Table 7.2).

Comparing Figures 7.2 and 7.4, the effect of our optimisations appears to be reduced when using the CVODE solver. This is because the adaptive nature of the solver introduces a much greater overhead than is needed for Euler's method, and thus the proportion of the total simulation time spent actually evaluating the cell models is reduced. We can measure the time spent in such evaluation, and the corresponding speedup is shown in Figure 7.5, showing almost identical behaviour regardless of which solver is used.

The performance of a tissue simulation is highly dependent on the efficiency of the simulation framework, and hence the proportion of the total simulation time spent in solving the ODE systems, as opposed to solving the PDEs, writing output to disk, etc. For illustrative purposes, however, we include some timing results of performing a simulation of a small block of tissue

[4]`https://computation.llnl.gov/casc/sundials/main.html`

Figure 7.4 The effect of ODE solver on optimisation efficiency. Simulations were performed as for Figure 7.2, except that the CVODE library was used to solve the ODEs. This uses adaptive time-stepping, rather than a fixed time step; the maximum time step was set for each model to the duration of the applied stimulus current, to ensure that the stimulus was acted upon. The results are displayed as in Figure 7.2, with the addition of an extra bar for each model showing the speedup achieved on the normal model by using the different ODE solver; this bar is scaled according to the axes on the right of the plot.

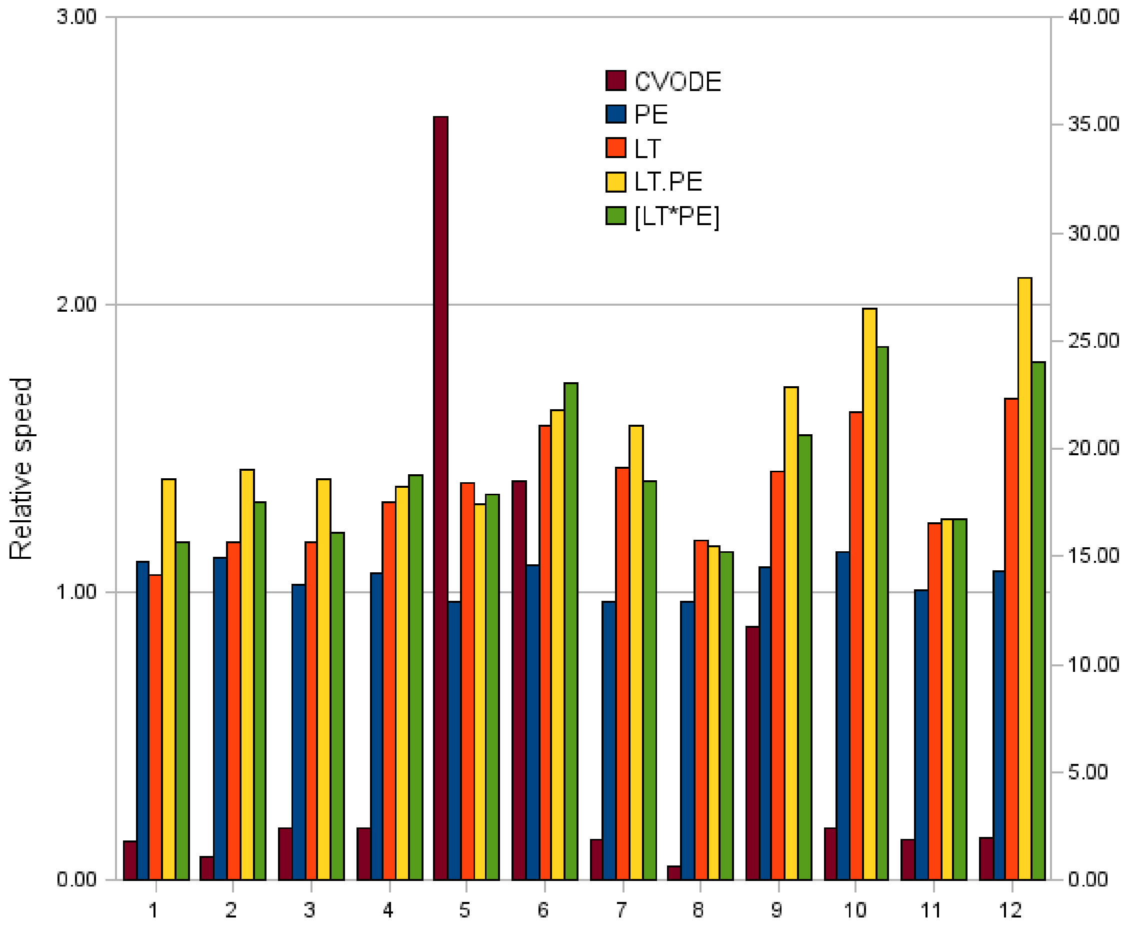


Figure 7.5 Fully optimised cell model evaluation times for different ODE solvers. The bars show the speed increase in evaluating the right hand side of the ODE system representing each cell model, after $\mathcal{PE}$ and $\mathcal{LT}$ have been performed, for both the Euler and CVODE solvers. Settings used for the simulations are as in Figure 7.2 (Euler) and Figure 7.4 (CVODE). Models are as in Figure 7.2.

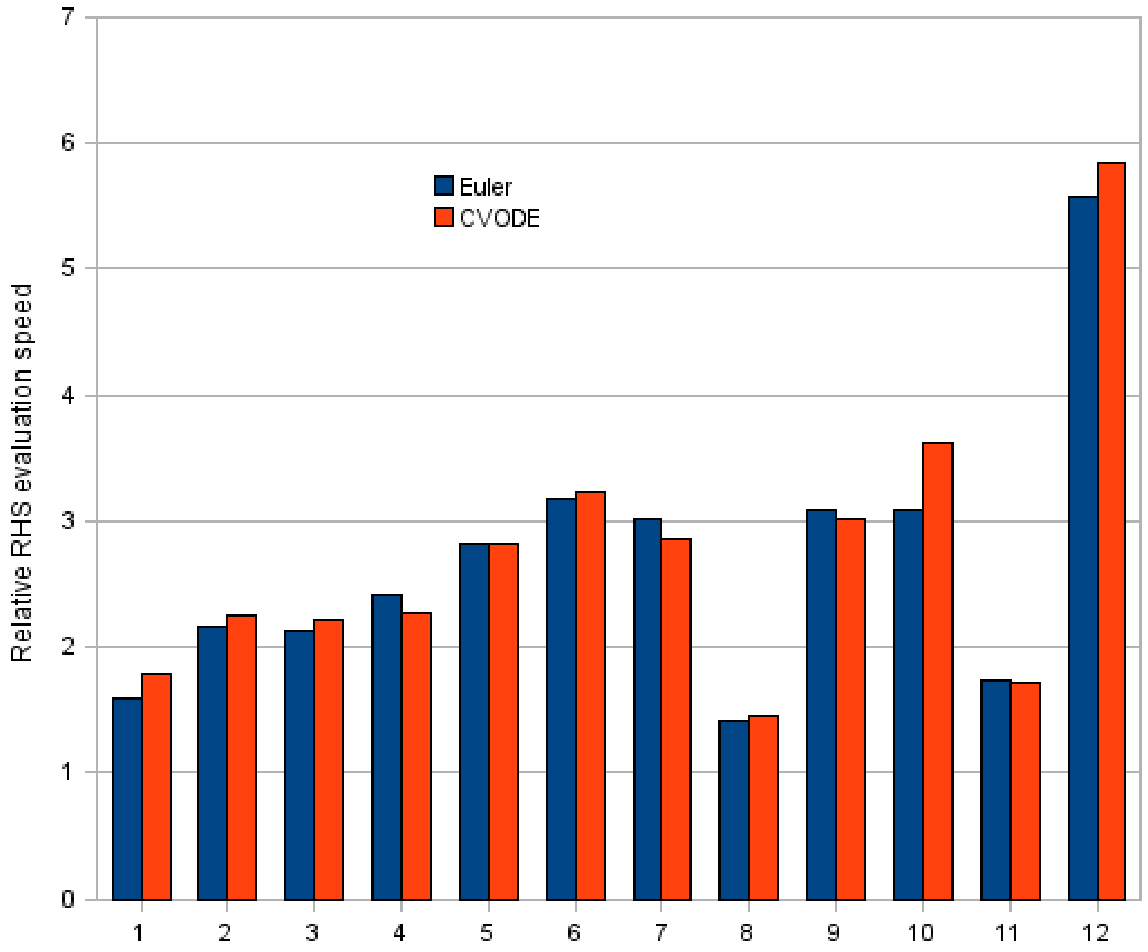
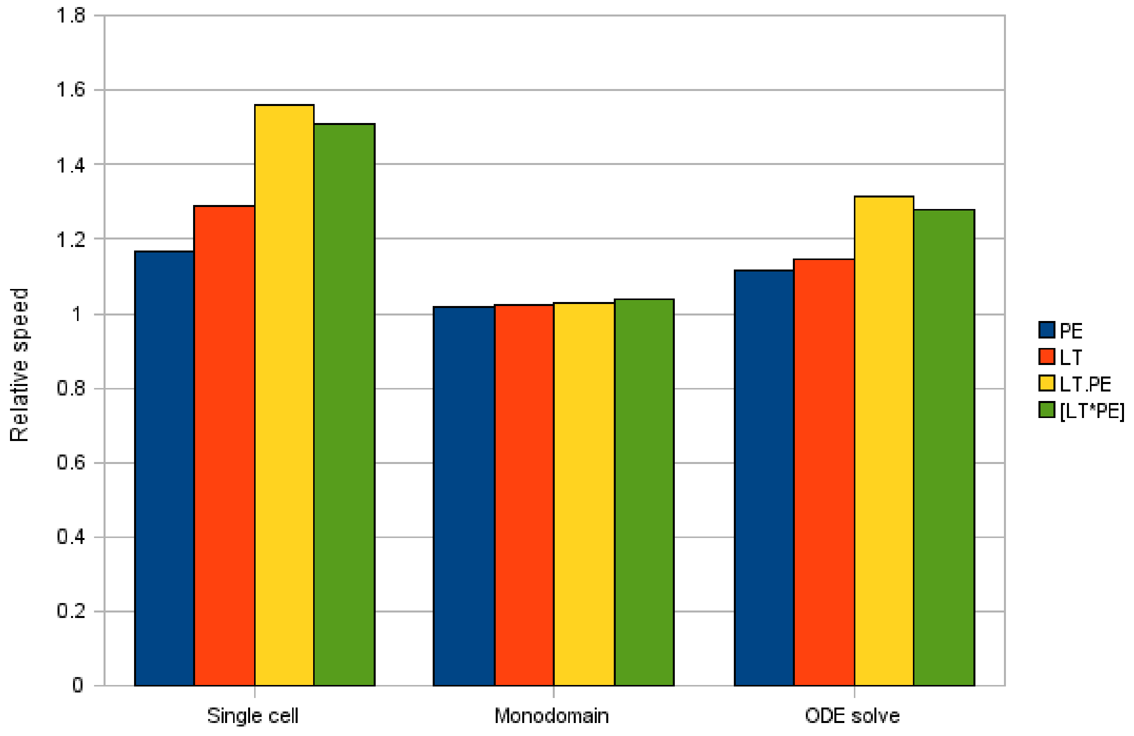


Figure 7.6 Timing results for a monodomain simulation of the Faber–Rudy model. Chaste was used to simulate a $2\,\mathrm{mm}^3$ block of tissue for $1\,\mathrm{s}$, with a single stimulus current applied to the left face at the start. The cell model ODEs were solved using Euler's method with a step size of $0.005\,\mathrm{ms}$. PDE solver settings were left at the defaults. The Intel compiler was used, and the simulation was run in parallel on an 8 core machine. The first set of bars shows results of a single cell simulation, taken from Figure 7.3 for comparison. The middle set shows the effect on the total simulation time, and the right hand set the effect on the time taken to solve the ODEs.



using Chaste. We are more restricted in our choice of models for this, since not all the models have been designed for use in a tissue simulation, and thus some can break down when used in such a context.

Figure 7.6 shows the results of such a simulation of the Faber and Rudy (2000) model. We see that the effect on the total running time for this (very small) simulation is slight, due to the overhead involved. Looking at the performance of the ODE solver itself, however, we see that the results are comparable with a single cell simulation, although the improvement is somewhat reduced. This is most likely to be due to poorer cache performance in the multicellular parallel simulation, and to the overhead involved in looping over cells, since this loop is included within the recorded time.

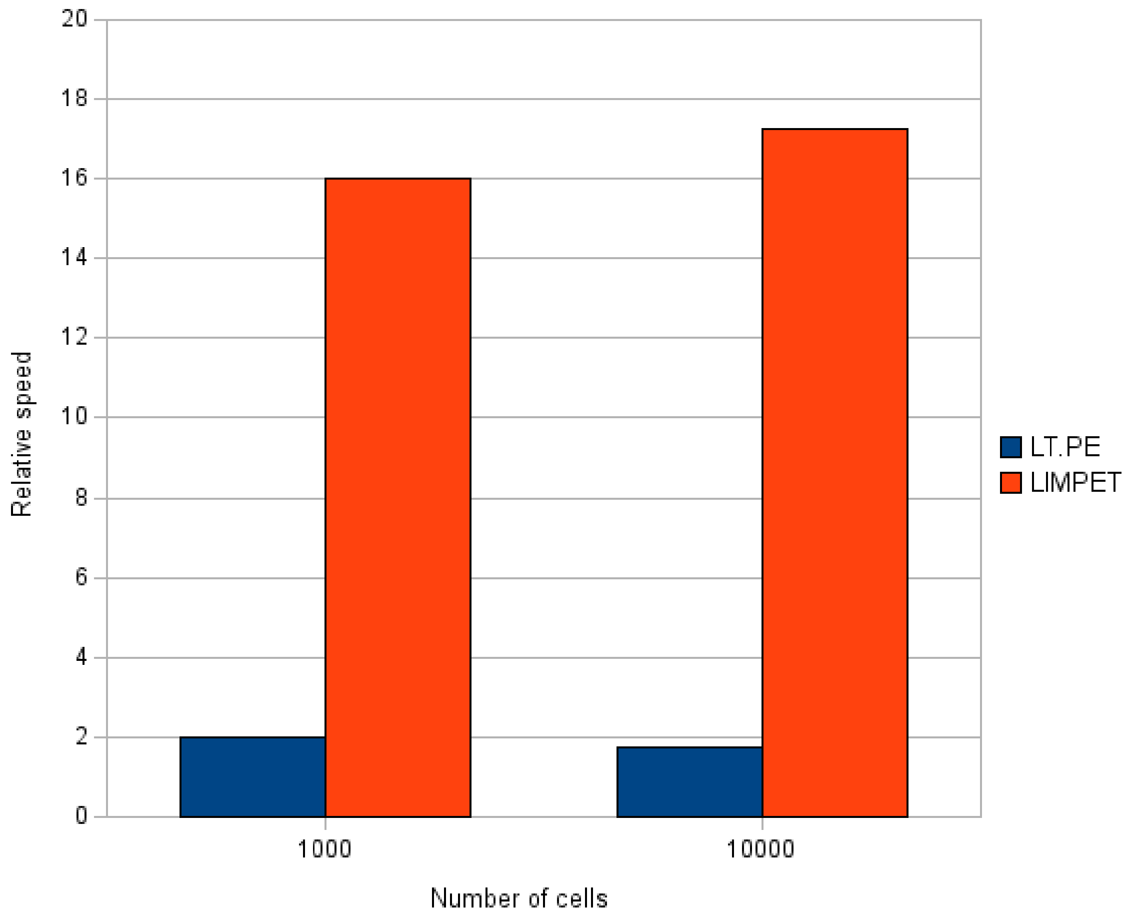
Finally, we also wish to know how the performance of our automatic techniques compares with hand-optimised code. For this, we have decided to test against the LIMPET cell model library, which is part of CARP (Vigmond et al., 2003). The models in this library have received considerable attention to ensure they match the original authors' versions, and have been extensively optimised for use in bidomain simulations. They can thus be considered the state of the art in this respect.

The additional memory requirements of a multicellular simulation can affect performance by significantly altering the usage of the processor's memory cache. LIMPET is typically benchmarked with one million cells, since the authors have observed that performance characteristics of single cell tests are so different as to make them meaningless. The simplest method of evaluating this effect is to simulate multiple 'disconnected' cells, imitating a tissue simulation but without considering interactions between cells.

Since a direct comparison requires using a CellML file that produces the same results when simulated as the LIMPET implementation of the model, we have thus far compared only one model—that of Courtemanche et al. (1998)—in a simulation of disconnected cells for 1 second of simulated time. The results are shown in Figure 7.7. We see that LIMPET outperforms our automatic optimisations, but we are within an order of magnitude. The results also suggest that LIMPET is slightly better at scaling to large numbers of cells than is Chaste.

The comparison is not entirely fair, since the automatically optimised code uses Euler's method to solve the ODE system, whereas LIMPET uses a more advanced solver hardwired into the model. It makes use of Rush–Larson techniques (Rush and Larsen, 1978) as well as updating different currents on different time scales to reduce the amount of computation required. Various other coding techniques are employed to make best use of cache and pipelining. It is thus not surprising that LIMPET gives the better performance at present. Future work (see also Chapter 8) will look at utilising this solution approach in automatically generated code.

Figure 7.7 Comparison of our optimisations with the hand-optimised LIMPET library. A collection of disconnected cells using the Courtemanche et al. (1998) model was simulated for 1 s, using both LIMPET, and Euler's method in Chaste with our optimisations. The running times were compared against that of simulating using Euler's method in Chaste with only compiler optimisation. In all cases the Intel compiler was used with full compiler optimisation.



7.2.4 Verification of code generation

To verify correctness of the translation to C++, the simulation data (i.e. a trace of the value of the transmembrane potential against time) produced when no optimisations were applied was compared with data produced by COR (Garny et al., 2003b). The same ODE solver and parameters were used in both cases. The results were compared numerically, and plotted to compare by eye. No significant differences were observed.

7.3 Summary

The results presented above show that the optimisations of CellML described in this thesis do decrease simulation times by a substantial amount. With 3-fold or better combined speedups of cell model evaluation for more complex models, we envisage techniques such as these becoming used as standard within the computational biology community. While at present hand-optimised model implementations are still faster, there is scope to automate the other optimisation techniques used, and so close the gap.

We have also further explored the interaction of partial evaluation with the use of lookup tables, and explained the reasons why applying PE first improves the effectiveness of the latter technique, allowing a greater proportion of the model to be approximated by lookup tables.

The optimisation techniques we have studied can be utilised no matter what compiler and simulation framework are used, or what machine simulations are run on. However, achieving optimal performance from simulations still requires knowledge of these factors. The use of modelling languages allows a better division of labour—those developing supporting tools need to be aware of these issues, but modellers need not be.

It would be desirable if the complexity estimate of Section 7.1 could be used to prove that performance will not be reduced by our optimisations. By induction, if each ‘rewrite rule’ of the partial evaluator either preserves or decreases our complexity measure, then the estimated speedup will be at least 1. A similar argument could be used for lookup tables—a single evaluation of the exponential function will always be more expensive than an interpolation (at least for non-trivial cases). However, these arguments only apply for actual simulation speeds if the complexity estimate is sufficiently close to reality, which we saw is not the case, primarily because the influences of machine architecture and compiler optimisation are not taken into account.

Having seen that automatic optimisation does work in practice, we now proceed to summarise the work performed, and look to future research directions.

8

Conclusions

In this chapter we summarise the main contributions of this work, and then proceed to discuss ways in which it might be extended. We have seen that insights from programming languages can be fruitfully applied to biological modelling languages, and this is expanded upon in Section 8.1.

Section 8.2 considers various extensions to our work, looking both at improving the techniques we have used, and at the possibilities of further optimisation techniques and additional application domains. One extension which we discuss in some detail in Section 8.3 is to apply our techniques to the whole CellML language, rather than the subset described in Chapter 3. We also consider the implications of some changes that have been proposed for future versions of CellML.

Finally, Section 8.4 takes a wider look at the physiological modelling research area, and considers what role computer science might play.

8.1 Summary

The use of quantitative mathematical models to describe the behaviour of biological systems is becoming increasingly important within both the biological and clinical sciences. This has been driven by major breakthroughs in biotechnology providing a deluge of data. Full quantitative understanding of this data can only be achieved through the iterative interplay between mathematical modelling, computational simulation, and laboratory experiments.

Rapid progress has led to a proliferation both of the models and the software for simulating them. The abundance of approaches, differing between research groups, causes difficulties for collaboration, reviewing models, reproducing results, and reusing simulation codes. At a recent European Heart Modelling Workshop, the issue of sharing models across research groups through the use of standard approaches was identified as one of the key obstacles to progress in international collaboration in the modelling of biological and physiological systems. Such collaboration is essential due to the complexity of the systems being modelled.

For certain classes of models, including cardiac cell models, the CellML language, described in Section 2.3, is gaining widespread acceptance. Such model description languages play a vital role in facilitating collaboration. Our contribution has shown that insights from programming languages can be fruitfully applied to biological modelling languages like CellML.

Firstly, as we saw in Chapter 4, static checks aid modellers by catching more errors earlier, and thus will improve the quality of published models. One benefit of validation is that if a CellML model is valid according to the rules defined in the specification, then it should be treated appropriately by any software that conforms to the specification, and thus its usefulness is increased—this promotes interoperability. Further checks, such as the units checking we have implemented, provide an important ‘reality check’ of the model. When coupled with automatic units conversion, this greatly promotes model reuse.

Secondly, optimisation of simulation software is essential for making better use of compute resources. If we are to obtain clinical usefulness, at least real-time simulation is needed. Due to the multiplicity of models, and the fact that model development is ongoing, automating optimisations is necessary. Furthermore, provably correct optimisation is essential if we are to rely on the results of our simulations, especially in the light of clinical applications. The contribution of this thesis, in automating and proving correct two optimisation techniques, is thus an important first step. We have shown that lookup tables and partial evaluation are effective optimisations of cardiac cellular models, particularly when combined to exploit synergy, as seen in the results of Chapter 7.

In summary, strong validation and provably correct optimisation provide us with both reduced simulation times and greater confidence that simulation results accurately reflect the model being simulated, speeding the progress of scientific investigation. The work we have performed, however, is only a start, and so we now consider what remains to be done.

8.2 Extensions

This research can be extended in various directions. We first consider improvements to the optimisation techniques already described, and then the possibilities of other optimisations. We also discuss whether techniques such as ours could be applied to models of other biological systems, perhaps defined using different modelling languages. The related issue of processing more of the CellML language is dealt with in Section 8.3. Section 8.2.1 addresses an important side issue raised by some cardiac models, which has implications for optimisation techniques, particularly the use of lookup tables.

The lookup tables technique admits a range of modifications. The framework developed here can easily be applied to tables indexed on other physiological quantities, for example the intracellular calcium concentration. Another possibility would be to look at using other types of lookup tables, for example using cubic spline interpolation to give a better approximation, or using nearest neighbour lookup to reduce further the computational cost. In this case the analysis for suitability would be unchanged—only the code generation would need to be updated. The code generation approach also makes it easy to tweak the output to enable maximum benefit from particular compilers or machine architectures. For example, currently inline methods are used to perform table lookups. If detailed profiling revealed this to introduce an undesirable overhead, an alternative strategy would be to interpolate all tables with one call at the start of each timestep, storing the results in a local array which could then be indexed as needed.

Another alteration to the code generation, which could improve performance in multicellular simulations (but requires appropriate support from the simulation environment, which Chaste does not yet provide), is to move the loop over cells inside the ODE solver loop, including it as part of the cell model class, for instance. This would facilitate the use of certain compiler optimisations such as automatic vectorization of floating point operations.

Other optimisation techniques for cardiac cell models have been investigated (see e.g. Whiteley, 2006, 2007) and implemented by hand for particular models. There is scope for these techniques to be automated. Doing so requires more extensive transformation of the cell models than the techniques we have presented, including symbolic rearrangement of mathematical expressions. Some work has already been done, and PyCml includes prototype code for performing the manipulations described by Whiteley (2006). A similar transformation would be to set Hodgkin–Huxley formulations of gating variables in the style proposed by Rush and Larsen (1978).

Another technique which is being trialled is to decompose the ODE system representing a model according to the timescale on which each differential equation operates. The principle is that equations operating on a fast timescale require small timesteps to be taken by the solver, whereas larger timesteps can be used for slow equations, and thus computational effort is reduced for these. Automating this technique will require careful analysis of the model to determine the timescales involved and choose appropriate time steps. A possible approach is to build on non-dimensionalisation techniques (see e.g. Khanin, 2001), since these can also involve an analysis of timescales. Proving correctness in this case will require numerical analysis to show that the timesteps chosen are appropriate.

To implement such techniques, it may be beneficial to do so in the context of a general functional transformation framework, as suggested in Chapter 5. Rather than use an in-memory representation of a CellML model that is closely tied to the XML structure, we may define a data type that is well suited to the processing required. With suitable ‘meta’ functions for handling this data type, model transformations can be described very concisely and elegantly.

Further benefits can be obtained from pattern matching in the implementation language, as this provides a simple means to transform only expressions having a certain form. There may also be scope for using automatic theorem provers to analyse the correctness of optimisations, if the required correctness properties can be stated in a suitable manner.

The design philosophy of CellML does not lend itself to ease of processing. The language is designed to be very flexible, allowing the expression of complex mathematics and hence a wide range of models, but with relatively little additional structure to aid tools. Processing software thus has to be capable of analysing the mathematics in order to determine how to proceed. Most tools restrict themselves to certain forms of expression, and the introduction of secondary specifications in CellML 1.2 will formalise this concept. Providing a more structured data type for representing CellML models could ease the burden on transformation tool developers.[1]

This might also provide a convenient bridge to aid in applying transformations to other modelling languages, such as SBML. There is currently some support for converting models between SBML and CellML, but it is limited. Since SBML provides more structure, being designed for a smaller class of models, it should be easier to convert into other forms.

There is certainly scope for the optimisation techniques we have looked at to be applied to other biological systems. By their very nature, biological quantities tend to be constrained within fixed limits, and hence there is the potential for the use of lookup tables to approximate portions of models. The effectiveness of this will depend on the complexity of the model. Where it is computationally complex, but with few varying quantities that may be used to index tables, then it is likely that lookup tables will be useful. If there are complex interactions between quantities, however, then there may be few expressions containing only one key variable, and so little scope for using tables.

In applying these techniques, the use of (biological) metadata will be crucial. Such information could be used to determine the natural ranges over which quantities vary, and hence where lookup tables could be used.

[1]See also `http://monasticxml.org/mirage.html`

Partial evaluation may also play a supporting role, as in this work, by improving the lookup table analysis. Where model structures are similar to those we have studied, implying simple code structure, partial evaluation by itself will have only a limited impact, as explained in Chapter 7. We would expect partial evaluation to have a much greater impact at the level of a whole-heart simulation, or other multi-level simulations. The concept of specialising a heart simulation to the particular cell models involved is certainly intriguing. However, this is a much more challenging prospect, since we would need a partial evaluator for the language the simulation environment was written in. Chaste is written in C++, for which we are not aware of any partial evaluators. There are tools for Java, in which some other environments are written, so that may be an avenue worth exploring. General simulation environments designed using software engineering techniques of abstraction, modularisation and object-orientation for the sake of clarity would benefit from the use of partial evaluation to give extra efficiency. It is expected that we could obtain significant speedups, due to the elimination of method lookup overheads, simplification of complex datatypes, removal of conversions between internal datatypes, and the like.

8.2.1 Singularities

Many cardiac cell models have equations similar to this one in the sodium channel in the Hodgkin-Huxley model:

$$\alpha_m = \frac{0.1(V+25)}{e^{0.1(V+25)}-1} \quad . \tag{8.2.1}$$

When the transmembrane potential V is $-25\,\mathrm{mV}$ this equation has a singularity. By setting $\tilde{v} = 0.1(V+25)$ and expanding $e^{\tilde{v}}$ as a Taylor series, we see that as $\tilde{v}$ tends to 0, α_m tends to 1, so we can make the equation well-defined for all physiological values of V by setting $\alpha_m = 1$ if $V = -25\,\mathrm{mV}$ and using (8.2.1) elsewhere.

Modellers usually treat such singularities as special cases and manually write code to take each occurrence into account. This is not always done in the CellML descriptions however. While these singularities rarely cause problems in a normal simulation (the transmembrane potential does not take a value close enough to $-25\,\mathrm{mV}$ to trigger the singularity), if one of the points of a lookup table lies at a singularity then this entry will be incorrect (it will be the floating point error value ‘NaN’), and so simulation with lookup tables is likely to give incorrect, non-numerical, results. We have adopted an ad-hoc solution of shifting the lookup table points slightly in this situation, so they no longer fall exactly at the singularity, and this has been sufficient to avoid any problems. A more rigorous treatment would be desirable, however. This requires begin able to recognise such singularities by analysing the model. The simple analysis used by our early prototype (Cooper et al., 2006) would be sufficient for those models we have seen, but a more general technique may be needed in the future. The simple analysis involves pattern matching on the expression tree to identify expressions with a form similar to (8.2.1); such expressions are then replaced by a piecewise expression. If the value of V is $-25\,\mathrm{mV}$ (or whatever the corresponding problem value is) then an average is taken between values for the expression $10^{-10}\,\mathrm{mV}$ either side of V; otherwise the original expression is used. Since the average uses known values of V, it can be computed at partial evaluation time, so here we see another benefit from partial evaluation. Since this approach is based on pattern matching, it would be extremely simple to implement in our Haskell framework; Python however does not have pattern matching of this sort built in, so this would require more work in PyCml.

In the next section we consider an important class of extensions in greater detail—applying our techniques to the full CellML language, both now and in the future.

8.3 Processing full CellML

As explained in Chapter 3, the semantics we have defined deals with only a subset of the CellML language, albeit a large subset capable of representing most existing models. In this section we consider what changes would be required to our techniques in order to handle the full language.

8.3.1 Implicit equations

Support for implicit equations is the most significant change which could be made. Much of the model analysis is simplified by the assumption that all model equations can be represented as straightforward assignments. If implicit equations are allowed, then software also has to be able to process systems of algebraic equations. A related change is the possibility that connections will lose their directionality in CellML 1.2: this is exactly equivalent to processing an expression such as $a = b$ in which it is not prescribed that b is the known quantity.

This extension has negligible impact on the units checking described in Chapter 4, since all variables and constants are explicitly annotated with their units, and thus each expression may be considered in isolation, regardless of whether it is explicit or implicit. The biggest change is to the variable classification algorithm (see also Section 2.3.2), which needs to consider an extra class of variable, 'system,' for those involved in a system of equations, as well as keeping track of which system(s) a variable is included in.

Identification of constant, state, and free variables is still straightforward, since these must all have an initial value set (in a valid model). Distinguishing between constants and the others is done by analysing the mathematics to see which occur in derivatives.

For the rest of the analysis, we consider a generalisation of the variable concept: an assignable quantity. This includes both variables and derivatives (e.g. $\frac{\mathrm{d}V}{\mathrm{d}t}$).

One approach is to first process each expression in the model (treating connections as expressions of the form $a = b$) to construct for each expression a set of the assignable quantities involved. Those quantities which have already been classified (as constant, state, or free variables) are then removed from the sets. For each expression, there are then three cases.

1. The associated set has size 0. The model is then overconstrained.

2. The associated set has size 1. The expression is then effectively an explicit assignment to the member of the set, although some manipulation and possibly a numerical solve may be required to write it in such a form.

3. The associated set has size greater than 1. The expression then forms part of a system of equations.

Those expressions matching the third case then need to be analysed to determine which must be solved together when simulating the model. Automatically determining the optimal division of such expressions into self-contained systems is still an open problem, being worked on at the University of Auckland. It is not too difficult to determine a reasonable approximation that will be suitable in many cases, however. Systems which are isolated from any others can be identified by associating sets of assignable quantities with the set of expressions involving just those quantities, and noting where the sets are the same size. Removing the relevant assignable quantities from the sets associated with other expressions could then reduce the number of unknowns involved in other systems. There are pathological cases not addressed by this approach, however, for example the set of equations

$$\begin{aligned} a+b &= c, \\ b+c &= d, \\ d+a &= b, \\ c & \quad \mathrm{constant}. \end{aligned}$$

This system of 3 equations can be used to determine a, b, and d, but would not be identified as such by the algorithm described above.

An analysis such as that described here can thus be used to convert a CellML model into an environment similar to that defined in Chapter 3. There are only two extensions needed. One is a data type for representing a system of equations, and an associated library of solver routines for evaluating such systems. The second is a new member of the *EnvValue* type, associating each assignable quantity determine by a system of equations with the appropriate system. Lazy evaluation may then still be used to evaluate the model, with the addition that asking for the value of a member of a system triggers a solve of the system.

This does have important implications for the semantics as a definition of model meaning, however, since with this setup the meaning is dependent on the particular system solver used. For the purposes of correctness proofs it is desirable to remove this dependence, but determining a suitable approach requires further investigation.

Implicit equations have surprisingly little further impact on partial evaluation. The binding time analysis becomes slightly less straightforward, with all the equations comprising a system needing to be treated together. The system is static if and only if no assignable quantity involved (in any of the expressions of which it is comprised) is determined to be dynamic. The partial evaluator can then do a PE-time solve of the system if it is static, and otherwise must reduce each of the equations involved.

8.3.2 CellML 1.1

Relatively little work would be required to support CellML 1.1. Essentially all that is needed is a pre-processing phase to convert a CellML 1.1 model into a CellML 1.0 model, by instantiating all the imports. The optimisation tools can then be used unchanged.

8.3.3 Metadata

As the metadata standards related to CellML develop, it will become possible to make use of them to refine the optimisations described in this thesis. For instance, biological metadata could be used to determine which variable represents the transmembrane potential, and hence should be used to key lookup tables. Further domain knowledge could also be encoded, perhaps indicating other quantities that vary over a known range and thus may be treated in a similar way. Such metadata will also be of use in code generation, allowing a more robust identification of the variables representing ionic currents or a current stimulus—this information is needed when linking cell models into a tissue simulations.

A proper handling of metadata is also good from an extensibility viewpoint. If our tools parsed the RDF into an in-memory graph, it would become feasible to ensure coherent metadata is written out to the specialised model. Metadata could perhaps also be used to store binding time and lookup table annotations, rather than using an extension namespace as at present.

8.3.4 User-defined functions

In order to describe more complicated models, the use of user-defined functions within CellML models is currently being considered. It is impossible to predict exactly how this will impact our work without knowing the form which these will take, but we can foresee some implications. The impact on units checking and conversion, for instance, was discussed in some detail in an earlier paper (Cooper and McKeever, 2008). Previous research on partial evaluation has also addressed the application to functions at great length; see for example Jones et al. 1993.

8.4 Computer science and Physiome modelling

The goal of physiome research is to understand the function of an organism as a single unit, largely through detailed quantitative mathematical modelling (Bassingthwaighte, 2000; STEP Consortium, 2007). While there are many models of components of the physiome (for example, models of the heart such as those described in this thesis), determining how these components interact to yield the functions of the whole system remains a significant challenge. We believe that contributions from the computer science community will be essential to address this, and that the use of modelling languages, such as CellML or SBML, will play a central role.

Some areas in which computer science can contribute are suggested by the work we have done. We have seen that insights from programming languages can be usefully applied to XML-based modelling languages, and we anticipate further progress in this field as new modelling languages, such as FieldML (Christie et al., accepted), are developed. As one example, research on modular design of software could be applied in considering how to interface models—to know when one model can be substituted for another, or to develop re-usable model components.

Another area is to ensure reliability and repeatability of computer simulations. In experimental science great importance is attached to recording every detail of the method followed in order to allow others to repeat the experiment and so check the results obtained. In theory simulation results should be easier to reproduce, since computer hardware provides an almost deterministic system. In practice this is not yet the case for various reasons. Modelling languages are part of the solution, providing a single source for both the simulation code and published mathematics, with efforts such as MIRIAM (Novere et al., 2005) and MIASE[2] standardising what metadata needs to be included, for example to link the mathematics with the biology being modelled, or the information needed about a simulation experiment in order to repeat it. Other projects are looking at tracking workflows, automatically recording the actions scientists take when they interact with a model, and recording metadata about the computer system used. Our work has looked more at the reliability issue—ensuring that optimisations (required to enable efficient simulation) do not change the meaning of the model and thus alter the results.

This remains a vibrant research area, particularly with the recent launch of the Virtual Physiological Human initiative (Fenner et al., 2008), and we have been privileged to play a part in it.

[2]`http://www.ebi.ac.uk/compneur-srv/miase/`

Bibliography

Mark Ainsworth and J. Tinsley Oden. *A Posteriori Error Estimation in Finite Element Analysis*. Wiley, 2000. ISBN 978-0-471-29411-5.

Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele Jr. Object-Oriented Units of Measurement. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 384–403, New York, NY, USA, October 2004. ACM Press. ISBN 1-58113-831-9. doi: 10.1145/1028976.1029008.

Peter Holst Andersen. Partial evaluation applied to ray tracing. Technical Report 95/2, DIKU, University of Copenhagen, January 1995. URL `http://repository.readscheme.org/ftp/papers/topps/D-289.pdf`.

Geoff Baldwin. Implementation of Physical Units. *SIGPLAN Notices*, 22(8):45–50, August 1987. ISSN 0362-1340. doi: 10.1145/35596.35601.

James B. Bassingthwaighte. Strategies for the physiome project. *Ann. Biomed. Eng.*, 28(8): 1043–1058, 2000. doi: 10.1114/1.1313771.

G. W. Beeler and H. Reuter. Reconstruction of the action potential of ventricular myocardial fibres. *J Physiol*, 268(1):177–210, 1977. URL `http://jp.physoc.org/cgi/content/abstract/268/1/177`.

A. A. Berlin. Partial evaluation applied to numerical computation. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 139–150, 1990.

A. A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2 edition, January 1998.

Vladimir E. Bondarenko, Gyula P. Szigeti, Glenna C. L. Bett, Song-Jung Kim, and Randall L. Rasmusson. A computer model of the action potential of the mouse ventricular myocytes. *Am J Physiol Heart Circ Physiol*, 287:H1378–H1403, May 2004. doi: 10.1152/ajpheart.00185.2003.

E. Carmeliet and J. Vereecke. *Cardiac cellular electrophysiology*. Kluwer Academic Publishers, London, 2002.

G. Richard Christie, Poul M. F. Nielsen, Shane A. Blackett, Chris P. Bradley, and Peter J. Hunter. FieldML: Concepts and implementation. *Phil Trans Roy Soc A*, accepted.

E. Chudin, J. Goldhaber, A. Garfinkel, J. Weiss, and B. Kogan. Intracellular Ca2+ Dynamics and the Stability of Ventricular Tachycardia. *Biophys. J.*, 77(6):2930–2941, 1999. URL `http://www.biophysj.org/cgi/content/abstract/77/6/2930`.

Jonathan Cooper and Steve McKeever. Experience Report: A Haskell interpreter for CellML. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 247–250, New York, NY, USA, Oct 2007. ACM Press. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291190.

Jonathan Cooper and Steve McKeever. A model-driven approach to automatic conversion of physical units. *Softw. Pract. Exper.*, 38(4):337–359, 2008. doi: 10.1002/spe.828.

Jonathan Cooper, Steve McKeever, and Alan Garny. On the application of partial evaluation to the optimisation of cardiac electrophysiological simulations. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 12–20, New York, NY, USA, Jan 2006. ACM Press. ISBN 1-59593-196-1. doi: 10.1145/1111542.1111546.

Marc Courtemanche, Rafael J. Ramirez, and Stanley Nattel. Ionic mechanisms underlying human atrial action potential properties: insights from a mathematical model. *Am J Physiol Heart Circ Physiol*, 275(1):H301–321, 1998. URL `http://ajpheart.physiology.org/cgi/content/abstract/275/1/H301`.

Autumn A. Cuellar, Catherine M. Lloyd, Poul F. Nielsen, David P. Bullivant, David P. Nickerson, and Peter J. Hunter. An Overview of CellML 1.1, a Biological Model Description Language. *SIMULATION*, 79(12):740–747, 2003. doi: 10.1177/0037549703040939. URL `http://sim.sagepub.com/cgi/content/abstract/79/12/740`.

Autumn A. Cuellar, Poul F. Nielsen, Matt D.B. Halstead, David P. Bullivant, David P. Nickerson, Warren J. Hedley, Melanie R. Nelson, and Catherine M. Lloyd. CellML Specification 1.1, February 2006. URL `http://www.cellml.org/specifications/cellml_1.1/`.

K.A. Deck and W. Trautwein. Ionic currents in cardiac excitation. *Pflügers Arch.*, 280(1):63–80, March 1964. doi: 10.1007/BF00412616. URL `http://www.springerlink.com/content/r651p6721073k66j`.

F. Dexter, G. M. Saidel, M. N. Levy, and Y. Rudy. Mathematical model of dependence of heart rate on tissue concentration of acetylcholine. *Am J Physiol Heart Circ Physiol*, 256 (2):H520–526, 1989. URL `http://ajpheart.physiology.org/cgi/content/abstract/256/2/H520`.

D. DiFrancesco and Denis Noble. A model of cardiac electrical activity incorporating ionic pumps and concentration changes. *Phil Trans Roy Soc B*, 307:353–398, 1985. URL `http://www.jstor.org/stable/2990169`.

R.W. dos Santos, F. Dickstein, and D. Marchesin. Transversal versus longitudinal current propagation on a cardiac tissue and its relation to MCG. *Biomed Tech (Berl)*, 47(Suppl 1 Pt 1): 249–252, 2002.

A. Dreiheller, M. Moerschbacher, and B. Mohr. Programming Pascal with Physical Units. *SIGPLAN Notices*, 21(12):114–122, December 1986. ISSN 0362-1340. doi: 10.1145/15042.15048.

K. Eriksson, D Estep, P Hansbo, and C Johnson. *Computational differential equations*. Cambridge University Press, 1996. ISBN 0-521-56738-6.

Gregory M. Faber and Yoram Rudy. Action Potential and Contractility Changes in [Na+]i Overloaded Cardiac Myocytes: A Simulation Study. *Biophys. J.*, 78(5):2392–2404, 2000. URL `http://www.biophysj.org/cgi/content/abstract/78/5/2392`.

J.W. Fenner, B. Brook, G. Clapworthy, V. Feipel, H. Gregersen, D.R. Hose, P. Kohl, P. Lawford, K.M. McCormack, D. Pinney, S.R. Thomas, S. Van Sint Jan, S. Waters, and M. Viceconti. The EuroPhysiome, STEP and a roadmap for the virtual physiological human. *Phil Trans Roy Soc A*, 366(1878):2979–2999, 2008. doi: 10.1098/rsta.2008.0089.

Jeffrey J. Fox, Jennifer L. McHarg, and Robert F. Gilmour Jr. Ionic mechanism of electrical alternans. *Am J Physiol Heart Circ Physiol*, 282:H516–H530, 2002. doi: 10.1152/ajpheart.00612.2001. URL `http://ajpheart.physiology.org/cgi/content/abstract/282/2/H516`.

Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995. ISBN 0201633612.

Alan Garny, Peter Kohl, Peter J. Hunter, Mark R. Boyett, and Denis Noble. One-Dimensional Rabbit Sinoatrial Node Models: Benefits and Limitations. *J Cardiovasc Electrophysiol*, 14 (s10):S121–S132, 2003a. doi: 10.1046/j.1540.8167.90301.x.

Alan Garny, Peter Kohl, and Denis Noble. Cellular Open Resource (COR): a public CellML based environment for modelling biological function. *Int J Bif Chaos*, 13(12):3579–3590, 2003b. doi: 10.1142/S021812740300882X. URL `http://cor.physiol.ox.ac.uk/`.

Alan Garny, David Nickerson, Jonathan Cooper, Rodrigo Weber dos Santos, Steve McKeever, Poul Nielsen, and Peter Hunter. CellML and Associated Tools and Techniques. *Phil Trans Roy Soc A*, 366(1878):3017–3043, 2008. doi: 10.1098/rsta.2008.0094.

K. Harriman, D.J. Gavaghan, and E. Süli. Approximation of linear functionals using an hp-adaptive discontinuous galerkin finite element method. Technical Report NA04/19, Oxford University Computing Laboratory, 2004. URL `http://www.comlab.ox.ac.uk/oucl/publications/natr/na-04-19.html`.

Warren J. Hedley and Melanie R. Nelson. CellML Specification 1.0, August 2001. URL `http://www.cellml.org/specifications/cellml_1.0/`.

Warren J. Hedley, Melanie R. Nelson, David P. Bullivant, and Poul F. Nielsen. A short introduction to CellML. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 359(1783):1073–1089, June 2001. doi: 10.1098/rsta.2001.0817. URL `http://www.journals.royalsoc.ac.uk/openurl.asp?genre=article&id=doi:10.1098/rsta.2001.0817`.

Peter Henrici. *Discrete Variable Methods in Ordinary Differential Equations*. Wiley, 1962.

Peter Henrici. *Error Propagation for Difference Methods*. The SIAM Series in Applied Mathematics. Wiley, 1963.

A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in the nerve. *J Physiol*, 117:500–544, 1952.

R. T. House. A proposal for an extended form of type checking of expressions. *Comput. J.*, 26 (4):366–374, 1983. ISSN 0010-4620. doi: 10.1145/4741.4742.

M. Hucka, A. Finney, B.J. Bornstein, S.M. Keating, B.E. Shapiro, J. Matthews, B.L. Kovitz, M.J. Schilstra, A. Funahashi, J.C. Doyle, and H. Kitano. Evolving a lingua franca and associated software infrastructure for computational systems biology: The systems biology markup language (SBML) project. *Systems Biology*, 1(1):41–53, June 2004. URL `http://sbml.org/documents/papers/iee-paper-printed.pdf`.

John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989. doi: 10.1093/comjnl/32.2.98.

P. J. Hunter, A. J. Pullen, and B. H. Smaill. Modelling total heart function. *Annu. Rev. Biomed. Eng.*, 5:147–177, 2003. doi: 10.1146/annurev.bioeng.5.040202.121537. URL `http://arjournals.annualreviews.org/doi/pdf/10.1146/annurev.bioeng.5.040202.121537`.

D. Isbell and D. Savage. Mars climate orbiter failure board releases report, numerous NASA actions underway in response. NASA Press Release 99-134, November 1999. URL `http://nssdc.gsfc.nasa.gov/planetary/text/mco_pr_19991110.txt`.

D. Isbell, M. Hardin, and J. Underwood. Mars climate orbiter team finds likely cause of loss. NASA Press Release 99-113, September 1999. URL `http://marsprogram.jpl.nasa.gov/msp98/news/mco990930.html`.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. ISBN 0-13-020249-5.

Michael Karr and David B. Loveman III. Incorporation of units into programming languages. *Commun. ACM*, 21(5):385–391, May 1978. ISSN 0001-0782. doi: 10.1145/359488.359501.

James Keener and James Sneyd. *Mathematical physiology*, volume 8 of *Interdisciplinary applied mathematics*. Springer, 1998. ISBN 0387983813.

Andrew Kennedy. Dimension Types. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, volume 788 of *LNCS*, pages 348–362, London, UK, 1994. Springer-Verlag. ISBN 3-540-57880-3.

R.C.P. Kerckhoffs, S.N. Healy, T.P. Usyk, and A.D. McCulloch. Computational methods for cardiac electromechanics. *Proceedings of the IEEE*, 94(4):769–783, April 2006. ISSN 0018-9219. doi: 10.1109/JPROC.2006.871772.

Raya Khanin. Dimensional analysis in computer algebra. In *ISSAC '01: Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, pages 201–208, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-417-7. doi: 10.1145/384101.384129.

J. D. Lambert. *Computational Methods in Ordinary Differential Equations*. Introductory Mathematics for Scientists and Engineers. Wiley, 1973.

Catherine M. Lloyd, Matt D.B. Halstead, and Poul F. Nielsen. CellML: its future, present and past. *Progress in Biophysics and Molecular Biology*, 85:433–450, 2004. doi: 10.1016/j.pbiomolbio.2004.01.004.

L. M. Loew and J. C. Schaff. The Virtual Cell: a software environment for computational cell biology. *Trends Biotechnol*, 19:401–406, 2001. URL `http://www.nrcam.uchc.edu/`.

Ching-hsing Luo and Yoram Rudy. A model of the ventricular cardiac action potential: Depolarization, repolarization, and their interaction. *Circ Res*, 68:1501–1526, 1991. URL `http://circres.ahajournals.org/cgi/content/abstract/68/6/1501`.

Ching-hsing Luo and Yoram Rudy. Dynamic model of the cardiac ventricular action potential—simulations of ionic currents and concentration changes. *Circulation Research*, 74:1071–1097, 1994. URL `http://rudylab.wustl.edu/research/cell/methodology/`.

R. Männer. Strong Typing and Physical Units. *SIGPLAN Notices*, 21(3):11–20, March 1986. ISSN 0362-1340. doi: 10.1145/382280.382281.

R E McAllister, D Noble, and R W Tsien. Reconstruction of the electrical activity of cardiac Purkinje fibres. *J Physiol*, 251(1):1–59, 1975. URL `http://jp.physoc.org/cgi/content/abstract/251/1/1`.

Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, April 1968.

W.T. Miller III and D.B. Geselowitz. Simulation studies of the electrocardiogram. i. the normal heart. *Circ Res*, 43(2):301–315, 1978. URL `http://circres.ahajournals.org/cgi/content/abstract/43/2/301`.

Sergey Missan and Terence F McDonald. Cese: Cell electrophysiology simulation environment. *Applied Bioinformatics*, 4(2):155–156, 2005. URL `http://cese.sourceforge.net/`.

P. Neumann. Risks to the public from the use of computers. *ACM Software Engineering Notes*, 10(3):5–16, July 1985.

D. Noble and S.J. Noble. A model of sino-atrial node electrical activity based on a modification of the DiFrancesco-Noble (1984) equations. *Proc Roy Soc B*, 222(1228):295–304, September 1984. URL `http://www.jstor.org/stable/35919`.

Denis Noble. Modelling the heart: insights, failures and progress. *BioEssays*, 24:1155–1163, 2002. doi: 10.1002/bies.10186. URL `http://www3.interscience.wiley.com/cgi-bin/fulltext/101019956/PDFSTART`.

Denis Noble. Modeling the Heart. *Physiology*, 19(4):191–197, 2004. doi: 10.1152/physiol.00004.2004. URL `http://physiologyonline.physiology.org/cgi/content/abstract/19/4/191`.

Denis Noble. Computational models of the heart and their use in assessing the actions of drugs. *J Pharmacol Sci*, 107(2):107–117, 2008. doi: 10.1254/jphs.CR0070042.

Denis Noble. Cardiac action and pacemaker potentials based on the hodgkin-huxley equations. *Nature*, 188:495–497, 1960. URL `http://www.nature.com/nature/journal/v188/n4749/pdf/188495b0.pdf`.

Denis Noble. A modification of the Hodgkin-Huxley equations applicable to Purkinje fibre action and pacemaker potentials. *J Physiol*, 160:317–352, 1962.

Denis Noble and Yoram Rudy. Models of cardiac ventricular action potentials: iterative interaction between experiment and simulation. *Phil Trans Roy Soc A*, 359(1783):1127–1142, June 2001. doi: 10.1098/rsta.2001.0820. URL `http://journals.royalsociety.org/content/xa9lfd46lrk17dnl/`.

Denis Noble, A. Varghese, Peter Kohl, and Penny Noble. Improved guinea-pig ventricular cell model incorporating a diadic space, i_{K_r} and i_{K_s}, length- and tension-dependent processes. *Canadian Journal of Cardiology*, 14(1):123–134, 1998. URL `http://www.pulsus.com/CARDIOL/14_01/nobl_ed.htm`.

Nicolas Le Novere, Andrew Finney, Michael Hucka, Upinder S Bhalla, Fabien Campagne, Julio Collado-Vides, Edmund J Crampin, Matt Halstead, Edda Klipp, Pedro Mendes, Poul Nielsen, Herbert Sauro, Bruce Shapiro, Jacky L Snoep, Hugh D Spence, and Barry L Wanner. Minimum information requested in the annotation of biochemical models (MIRIAM). *Nat Biotech*, 23:1509–1515, 2005. doi: 10.1038/nbt1156.

A. Nygren, C. Fiset, L. Firek, J.W. Clark, D.S. Lindblad, R.B. Clark, and W.R. Giles. Mathematical model of an adult human atrial cell the role of K+ currents in repolarization. *Circ Res*, 82:63–81, 1998. URL `http://circres.ahajournals.org/cgi/reprint/82/1/63`.

Joe Pitt-Francis, Miguel O. Bernabeu, Jonathan Cooper, Alan Garny, Lee Momtahan, James Osborne, Pras Pathmanathan, Blanca Rodriguez, Jonathan P. Whiteley, and David J. Gavaghan. Chaste: Using Agile Programming Techniques to Develop Computational Biology Software. *Phil Trans Roy Soc A*, 366(1878):3111–3136, 2008. doi: 10.1098/rsta.2008.0096.

M. Potse, B. Dube, J. Richer, A. Vinet, and R. M. Gulrajani. A comparison of monodomain and bidomain reaction-diffusion models for action potential propagation in the human heart. *IEEE Transactions on Biomedical Engineering*, 53(12):2425–2435, December 2006. ISSN 0018-9294. doi: 10.1109/TBME.2006.880875.

J Reddy. *An Introduction to the Finite Element Method*. McGraw–Hill, 1993.

Blanca Rodriguez, Li Li, James C. Eason, Igor R. Efimov, and Natalia A. Trayanova. Differences Between Left and Right Ventricular Chamber Geometry Affect Cardiac Vulnerability to Electric Shocks. *Circ Res*, 97(2):168–175, 2005. doi: 10.1161/01.RES.0000174429.00987.17. URL `http://circres.ahajournals.org/cgi/content/abstract/97/2/168`.

Blanca Rodriguez, Natalia Trayanova, and Denis Noble. Modeling Cardiac Ischemia. *Ann NY Acad Sci*, 1080(1):395–414, 2006. doi: 10.1196/annals.1380.029. URL `http://www.annalsnyas.org/cgi/content/abstract/1080/1/395`.

Yoram Rudy and Jonathan R. Silva. Computational biology in the study of cardiac ion channels and cell electrophysiology. *Quarterly Reviews of Biophysics*, 39(1):57–116, February 2006. doi: 10.1017/S0033583506004227.

Stanley Rush and Hugh Larsen. A practical algorithm for solving dynamic membrane equations. *IEEE Transactions on Biomedical Engineering*, BME-25(4):389–392, July 1978. ISSN 0018-9294. doi: 10.1109/TBME.1978.326270.

N G Sepulveda, B J Roth, and Jr Wikswo, J P. Current injection into a two-dimensional anisotropic bidomain. *Biophys. J.*, 55(5):987–999, 1989. URL `http://www.biophysj.org/cgi/content/abstract/55/5/987`.

Lauralee Sherwood. *Human physiology: from cells to systems*. Brooks/Cole, 4 edition, 2001. ISBN 0534568262.

N. P. Smith, D. P. Nickerson, E. J. Crampin, and P. J. Hunter. Multiscale computational modelling of the heart. *Acta Numerica*, 2004. doi: 10.1017/S0962492904000200.

STEP Consortium. Seeding the europhysiome: A roadmap to the virtual physiological human. Online, July 2007. URL `http://www.europhysiome.org/roadmap`.

Endre Süli and David Mayers. *An Introduction to Numerical Analysis*. Cambridge University Press, 2003.

Joakim Sundnes, Glenn Terje Lines, and Aslak Tveito. Efficient solution of ordinary differential equations modeling electrical activity in cardiac cells. *Mathematical Biosciences*, 2001. doi: 10.1016/S0025-5564(01)00069-4.

K. H. W. J. ten Tusscher and A. V. Panfilov. Alternans and spiral breakup in a human ventricular tissue model. *Am J Physiol Heart Circ Physiol*, 291(3):H1088–1100, 2006. doi: 10.1152/ajpheart.00109.2006. URL `http://ajpheart.physiology.org/cgi/content/abstract/291/3/H1088`.

K. H. W. J. ten Tusscher, D. Noble, P. J. Noble, and A. V. Panfilov. A model for human ventricular tissue. *Am J Physiol Heart Circ Physiol*, 286(4):H1573–1589, 2004. doi: 10.1152/ajpheart.00794.2003. URL `http://ajpheart.physiology.org/cgi/content/abstract/286/4/H1573`.

L. Tung. *A bidomain model for describing ischemic myocardial D.C. potentials*. PhD thesis, Massachusettes Institute of Technology, Cambridge, MA, 1978.

Edward J. Vigmond, Matt Hughes, G. Plank, and L. Joshua Leon. Computational tools for modeling electrical activity in cardiac tissue. *Journal of Electrocardiology*, 36:69–74, Dec 2003. doi: 10.1016/j.jelectrocard.2003.09.017.

E.J. Vigmond, R. Weber dos Santos, A.J. Prassl, M. Deo, and G. Plank. Solvers for the cardiac bidomain equations. *Progress in Biophysics and Molecular Biology*, 96(1–3):3–18, 2008. doi: 10.1016/j.pbiomolbio.2007.07.012.

Daniel Weise and Scott Seligman. Accelerating Object-Oriented Simulation via Automatic Program Specialization. Technical Report CSL-TR-92-519, Stanford University, April 1992. URL `http://repository.readscheme.org/ftp/papers/fuse-memos/FUSE-MEMO-92-10.pdf`.

Jonathan P. Whiteley. An efficient numerical technique for the solution of the monodomain and bidomain equations. *IEEE Transactions on Biomedical Engineering*, 53(11):2139–2147, Nov 2006. doi: 10.1109/TBME.2006.879425.

Jonathan P. Whiteley. Physiology driven adaptivity for the numerical solution of the bidomain equations. *Annals of Biomedical Engineering*, 35(9):1510–1520, 2007. doi: 10.1007/s10439-007-9337-3.

Hernán Wilkinson, Máximo Prieto, and Luciano Romeo. Arithmetic with Measurements on Dynamically-Typed Object-Oriented Languages. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 292–300, New York, NY, USA, October 2005. ACM Press. ISBN 1-59593-193-7. doi: 10.1145/1094855.1094964.

Raimond L. Winslow, Jeremy Rice, Saleet Jafri, Eduardo Marban, and Brian O'Rourke. Mechanisms of Altered Excitation-Contraction Coupling in Canine Tachycardia-Induced Heart Failure, II : Model Studies. *Circ Res*, 84(5):571–586, 1999. URL `http://circres.ahajournals.org/cgi/content/abstract/84/5/571`.

Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, 1977. ISSN 0001-0782. doi: 10.1145/359863.359883.

World Health Organisation. The global burden of disease: 2004 update. Online, October 2008. URL `http://www.who.int/healthinfo/global_burden_disease/2004_report_update/en/index.html`.

H. Zhang, A. V. Holden, I. Kodama, H. Honjo, M. Lei, T. Varghese, and M. R. Boyett. Mathematical models of action potentials in the periphery and center of the rabbit sinoatrial node. *Am J Physiol Heart Circ Physiol*, 279(1):H397–421, 2000. URL `http://ajpheart.physiology.org/cgi/content/abstract/279/1/H397`.

A

An Overview of the Haskell Programming Language

In this appendix we give a brief overview of Haskell, in order to enable readers unfamiliar with the language to follow those chapters in which it is used, notably Chapters 3 and 5. For further details on the language itself we suggest Bird (1998) and the `haskell.org` website.

Haskell is a lazy, statically and implicitly typed, pure functional language. The most fundamental of these is that it is *functional*: programs are functions, and evaluating a program is equivalent to evaluating a mathematical function. This contrasts with *imperative* languages which evaluate a sequence of statements in order. Hughes (1989) gives an excellent explanation of the usefulness of functional programming in general.

Haskell is also *pure* because evaluation of functions is not permitted to cause side-effects, which modify the 'state' of the world (e.g. by changing a global variable).[1] Haskell thus does not have *destructive update*—variables are not references to memory locations the contents of which can be changed. Rather a variable is just a binding of an expression to a name. A consequence of purity is that if a function is called twice with the same arguments, it will give the same result each time.

Evaluation is *lazy* in that expressions whose values are not required to determine the result

[1] Of course, printing something to the screen or a file is also a side-effect; Haskell uses *Monads* to isolate impure computations and perform them safely.

of the program are simply not evaluated. This has powerful implications, including that one can construct infinite data structures provided one never uses the entire structure.

Finally, Haskell has a strong *type system*: all expressions have a type, and these are checked for correctness by the compiler. The programmer need not, generally, specify types explicitly, however, since *type inference* is used to determine the type of expressions automatically.

We next present the main features of Haskell's syntax and semantics in Section A.1. Section A.2 builds on this by giving definitions of those functions defined in the Haskell standard library which we make use of.

A.1 The main features of Haskell

A Haskell program consists primarily of a sequence of function definitions.[2] Functions themselves are defined by a series of equations, and may also have a type signature declaration, which we will discuss in Section A.1.1. Simple functions, for example the *increment* function, may be defined by a single equation:

```
increment n = n + 1
```

Multiple arguments may be provided, separated by spaces, for example:

```
add x y = x + y
```

Functions can also be defined by multiple equations. The following recursive definition of the **length** function provides an example introducing several core features of Haskell:

```
length []       = 0
length (x : xs) = 1 + length xs
```

This function takes a single list as input, and determines its length. It is defined in two parts: the length of an empty list is zero, and the length of a list consisting of a single item *x* followed by the remainder of the list *xs* is one plus the length of the remainder of the list. The use of recursion as a basic control structure is common to functional languages, in contrast to imperative languages which make much use of looping—looping does not make much sense in the absence

[2]The order in which functions are defined does not matter.

of destructive update. This definition of **length** also illustrates the use of *pattern matching*: the input to the **length** function is matched against each of the patterns given in the function definition in turn to determine which equation to use. Patterns may be quite complex—as complex as the types of the arguments—and we return to them later.

Furthermore, the **length** function introduces us to notation for describing lists. An empty list is represented by '[]', and the ':' operator[3] prepends an item to the beginning of a list.[4] Lists are a central data type in most functional programming languages, and Haskell is no exception, providing much syntactic sugar to make lists easy to represent in programs. Rather than using : and [] to construct a list explicitly, the members may be enumerated directly in square brackets; [1, 2, 3] is equivalent to 1:2:3:[]. There are also some special shorthands for numeric lists; for example [*n*..] represents the infinite list [*n*, *n*+1, *n*+2, ...]. We also see in the **length** example a common naming convention for lists: *xs* is the plural of *x*, and thus represents a list, whereas *x* represents an element of the list.

The expression '1 + **length** *xs*' gives an example of function application. Spaces are used instead of brackets and commas to delineate the function arguments; multiple arguments are separated by spaces, e.g. *add* 1 2. Function application also has a very high precedence, binding tighter than any infix operator (such as +). Round brackets are used to override the normal precedence rules.

Another important feature of Haskell is the use of *partial function application*. A function of two arguments may be applied to a single input to obtain a function of one argument. For example, the *increment* function above could also be defined by

```
increment = add 1
```

This technique is closely related to the fact that functions are 'first-class' in Haskell, and may themselves be passed as arguments to functions or returned as results. Functions that are designed to take functions as arguments are known as *higher-order* functions.

[3]: is read as *cons*.

[4]Actually, : is a data constructor—see Section A.1.1.

Operators are essentially the same as functions, except that their names consist of symbols rather than alphanumeric characters, and by default they are used (and defined—see the '.' example below) in an infix fashion. However, operators may be used just like normal functions by writing them in brackets, for example (+) 1 2, and functions may be written in infix notation by surrounding them with backticks: 1 `*add*` 2. Since operators are just functions, it also makes sense to partially apply them, and this is known as *sectioning*. The *increment* function could thus also be defined as *increment* = (+1); the brackets here are mandatory.

As well as named function definitions, functions may be defined anonymously using *lambda abstractions*, thus allowing small functions to be defined at the point of use, since such a definition is syntactically an expression. A function equivalent to *add* could be written as

```
    λ x y → x + y
```

Function composition also exists, and works in the same way as for mathematical functions. It is implemented by the infix operator '.' which is defined by

```
f . g = λ x → f (g x)
```

This says that, for any functions *f* and *g*, the function "*f* composed with *g*" is defined as a function mapping a single parameter *x* to the value *f* (*g x*).

There are two mechanisms provided for creating a local nested scope for definitions: the **let** expression and **where** clause. In each case multiple local definitions may be made. The main difference is that a **let** expression is actually an expression, whereas a **where** clause is a syntactic construct. For an example, consider finding the roots of the quadratic polynomial $ax^2 + bx + c$:

```
roots a b c =
    let det = sqrt (b∗b − 4∗a∗c)
        twice_a = 2∗a
    in ((−b + det) / twice_a ,
        (−b − det) / twice_a)
```

This could also be written using a **where** clause:

```
roots a b c = ((−b + det) / twice_a , (−b − det) / twice_a)
  where det = sqrt (b∗b − 4∗a∗c)
        twice_a = 2∗a
```

These examples also illustrate the use of indentation to structure code without the need for explicit braces and semicolons.

Pattern matching provides a way to 'dispatch control' based on structural properties of a value—selecting a different expression to evaluate depending on whether the argument list is empty or not, for example. The **case** expression provides a means to utilise this without defining a function.[5] Using **case** we could define **length** as

```
length xs = case xs of
     []      → 0
     (y:ys)  → 1 + length ys
```

Patterns may use the underscore character as a wildcard, where we do not use one of the values being matched against. For example, the **head** function returns the first element of a list, and hence does not care about the remainder of the list; it is defined by

```
head (x:_) = x
```

We can read this as "the head of a list that starts with some datum *x* and continues with any list, is *x*." There are also *as-patterns* for use when we want to bind names to the whole of a value as well as constituents of it, for instance if we want to use the whole of a list in a function definition as well as its first element. One such example is the standard function **dropWhile**, which takes a predicate function and a list as input, and returns a new list in which any initial elements that satisfy the predicate do not appear.

```
dropWhile p []          = []
dropWhile p xs@(x:xs')  = if p x then dropWhile p xs' else xs
```

This also demonstrates the **if-then-else** control construct, which has the obvious semantics. Note that an **else** clause is mandatory.

Finally, Haskell programs may also contain comments. Single line comments begin with a double dash (−−) and extend to the end of the line. Multi-line comments are enclosed in a brace and a dash:

```
{- for example like this -}
```

[5] In fact, the meaning of pattern matching in function definitions is defined in terms of **case** expressions.

A.1.1 Types

So far we have not specified the types of any of our example functions. Due to the powerful type inference of Haskell, this is usually not a problem. However, it is often useful for the programmer to specify types explicitly, both as a form of documentation, and as a statement of intent about program meaning that the compiler can check—this can then help in catching programming errors.

The *increment* function could be given the following type in a *type signature declaration*.

```
increment :: Integer → Integer
increment n = n + 1
```

Here '::' is read as 'has type,' in this case a function type which takes an **Integer** and returns an **Integer**. Simple values can also have their type specified, for example

```
1.0 :: Double
```

Note that type names always start with upper-case letters in Haskell, and variables always start with lower-case letters—this is a rule of the language, not a naming convention.

There are many built-in types, including atomic types such as **Integer**, **Double** and **Char**, as well as compound types such as lists (for example [1, 2, 3] :: [**Integer**]) and tuples (like (1, '*a*') :: (**Integer**, **Char**)).[6] Function types are specified using the *type constructor* → to create the function type from an argument and result type. Functions of multiple arguments use multiple arrows, for example the *add* function:

```
add :: Integer → Integer → Integer
add x y = x + y
```

The → operator on types associates to the right, so this type signature could be written as

```
add :: Integer → (Integer → Integer)
```

This links with partial function application—if the *add* function is applied to a single **Integer**, the result is a function mapping **Integer** to **Integer**.

Haskell also incorporates *polymorphic types*, which allow us to discuss families of types

[6]Note that lists hold *arbitrarily many* values of the *same* type, whereas tuples hold a *fixed* number of values which may be of *different* types.

universally quantified over some *type variable*. For example, $[a]$ is the type family containing, for any type a, lists of values of type a.[7] This allows us to give a type to the **length** function. Since it can determine the length of any list, no matter what type of data it contains, it has the type $[a] \rightarrow$**Integer**.

User-defined types may also be declared. There are two mechanisms for doing so. The simplest is to declare an alias for an existing type, to reduce typing and aid readability; this is accomplished by the **type** keyword. The **String** type is one example—a string is simply a list of characters.

```
type String = [Char]
```

Defining entirely new types is done using the **data** keyword. Types can be straightforward enumerations, for example the **Bool** type of truth values:

```
data Bool = False | True
```

New types may also be based on existing types, as is the case for the type we gave for assignments in a CellML model:

```
data MathAssignment = Assign EnvKey MathTree
```

Here *Assign* is a *data constructor*, which is applied to values of types *EnvKey* and *MathTree* to produce a value of type *MathAssignment*.

Polymorphic types may also be defined, and types may be recursive. An example of both of these is the type of binary trees:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

The *type constructor Tree* constructs a new type from any type a: the type of trees containing elements of type a. A tree may either be a leaf element (of type a) or an internal branch consisting of two sub-trees. Note that type constructors and data constructors exist in separate namespaces, so we can use the same name for both, as was done for cases in piecewise expressions:

```
data Case = Case MathTree MathTree
```

In the next section we consider various standard library functions used in this work. The stan-

[7]Type variables are written in lower case to distinguish them from specific types such as **Integer**.

dard library also defines many useful types, including such types as lists and strings mentioned already. Three other types deserve mention.

The **Maybe** type is used when we are not sure if we will have a value of a certain type or not, and is defined as

```
data Maybe a = Nothing | Just a
```

There are also functions **isJust** and **isNothing** to test for whether a value of type *a* is present or not. The **Either** type allows us to choose between two types for representing a datum:

```
data Either a b = Left a | Right b
```

Finally, Haskell code may also be organised into modules. We do not discuss the full syntax here, however, the *Set* datatype which we use is defined in a separate module '*Set*' which is imported. Functions and types in this module can be accessed using *qualified* names, such as *Set.fromList* (which constructs a set from a list of data) or *Set.member* (a membership test for sets).

A.2 Standard Haskell functions

Since lists are a fundamental data structure in Haskell, many of the standard functions involve processing lists. The most important of these are the higher-order functions **map**, **foldr**, and **foldl**, which can be defined as follows:

```
map :: (a → b) → [a] → [b]
map f []     = []
map f (x:xs) = f x : map f xs

foldr :: (a → b → b) → b → [a] → b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl :: (a → b → a) → a → [b] → a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Each of these uses the function supplied as its first argument to process the input list. In the case of **map** the function is simply applied to each member of the list, thus producing a list as output. The fold functions are somewhat more complex. They 'fold' the list up using the supplied function in order to obtain a single value (although this value may have an arbitrarily complex type). One can think of them as replacing the list constructor with the function parameter, and the empty list by the initial value parameter; for instance:

```
foldr (+) 0 [3, 4, 5] = 3 + 4 + 5
```

The distinction between **foldr** and **foldl** is in the way the result is bracketed: **foldr** assumes the function is right-associative (so gives 3 + (4 + 5)) whereas **foldl** assumes left-associativity (i.e. (3 + 4) + 5). There are also variants which assume the input list is non-empty, and so do not require an initial value to be supplied. For example:

```
foldl1 f (x:xs) = foldl f x xs
```

Other list processing functions can often be defined in two forms: either directly using recursion, or in terms of folds and maps. The **sum** function, which sums the elements of a list, is a good example. It can be defined directly as

```
sum []     = 0
sum (x:xs) = x + sum xs
```

or using **foldl** as

```
sum = foldl (+) 0
```

(it could also be defined using **foldr**, since + is associative).

The following are definitions for some of the other standard list processing functions that we use. The first, the ++ operator, is used to concatenate two lists. The **and** and **or** functions apply the logical operators && and || (also defined below) to a list of booleans to determine if all are **True** or at least one is **True**, respectively.

```
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

and, or :: [Bool] → Bool
and = foldr (&&) True
or  = foldr (||) False

(&&), (||) :: Bool → Bool → Bool
False && x = False
True  && x = x
False || x = x
True  || x = True
```

The **iterate** function defined below is typically used in a context where we wish to apply a function repeatedly to some initial value until a condition is met. Given a function f and value x it (lazily) constructs the infinite list $[x, f\,x, f\,(f\,x), f\,(f\,(f\,x)), ...]$. The functions **dropWhile** and **head** (defined earlier) can then be applied to this list to select the first element satisfying a given condition.

```
iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)
```

Another standard function, shown below, finds the maximum element of a list. This uses a feature of Haskell not mentioned above, namely *type classes*, in this case the class **Ord** of types which possess an ordering relation (and hence values of such a type may be compared using the $<$ operator). We do not discuss type classes in any detail here, except to mention the existence of a few key ones. The **Show** class contains types for which the function **show** may be used to obtain a string representation of values of that type, while the **Eq** class contains types whose values may be compared for equality (using ==). There are also a range of numeric classes (such as **Integral**, **Rational**, **Floating**) for which many of the standard mathematical operators and functions are defined.

```
maximum :: Ord a ⇒ [a] → a
maximum = foldl1 max
```

Finally, we end with the **error** function. This has the curious type **String** $\to a$, implying that it can return any type at all, even though it is always given a **String** argument. When evaluated it throws an exception, which will typically halt program execution and display the argument as an error message.

B

Partial Evaluator Correctness Proof Details

This appendix contains further details of the correctness proof for our partial evaluator, which were omitted from the main text for conciseness.

B.1 Binding time analysis

The crucial property of BTA is stated in Theorem 5.2, namely that evaluation of a static expression may be performed solely within the static portion of the model environment. It is thus required to prove that for any environment *env* and expression *expr* within it,

$$\mathit{bta\ env\ expr} = S \Rightarrow \mathit{eval\ env_s\ expr} = \mathit{eval}\ (\mathit{env_s} \oplus \mathit{env_d})\ \mathit{expr}$$

where

$$(\mathit{env_s}, \mathit{env_d}) = \mathit{partition\ env}.$$

Proof (Theorem 5.2). The proof proceeds by structural induction on the form of *expr*. We denote the inductive hypothesis by IH. Most cases are simple to prove, and so we focus here on those involving short-circuiting. Firstly however we briefly consider the case of a variable lookup.

Suppose $\mathit{expr} = \mathit{Variable}\ v$. Since the expression is static, by Lemma 5.1(iii) we have that $\mathit{Var}\ v \in \mathit{env_s}$. There are thus two possibilities to consider.

i. *find* env_s (*Var v*) = *Expr e*:

By the definition of *bta_key*, *bta env e* = S. Thus,

$$\begin{array}{ll} & eval\ env_s\ expr \\ = & \quad \{\text{definition of } eval\} \\ & elookup\ env_s\ (Var\ v) \\ = & \quad \{\text{definition of } elookup\} \\ & eval\ env_s\ e \\ = & \quad \{\text{IH on } e\} \\ & eval\ env\ e \\ = & \quad \{\text{definition of } elookup \text{ and Lemma 5.1(i)}\} \\ & elookup\ env\ (Var\ v) \\ = & \quad \{\text{definition of } eval\} \\ & eval\ env\ expr \end{array}$$

ii. *find* env_s (*Var v*) = *Val val*:

This case is even more simple.

$$\begin{array}{ll} & eval\ env_s\ expr \\ = & \quad \{\text{definition of } eval\} \\ & elookup\ env_s\ (Var\ v) \\ = & \quad \{\text{definition of } elookup\} \\ & val \\ = & \quad \{\text{definition of } elookup \text{ and Lemma 5.1(i)}\} \\ & elookup\ env\ (Var\ v) \\ = & \quad \{\text{definition of } eval\} \\ & eval\ env\ expr \end{array}$$

The case for an ODE lookup, *expr* = *Diff* v_1 v_2, is very similar.

Suppose *expr* = *Apply operator operands*. In the general case, for each operand *e*, *bta env e* = S, since *bta env expr* = **maximum** (**map** (*bta env*) *operands*). We thus have that

$$\begin{array}{ll} & eval\ env_s\ expr \\ = & \quad \{\text{definition of } eval\} \\ & apply\ operator\ (\mathbf{map}\ (eval\ env_s)\ operands) \\ = & \quad \{\text{IH on each operand } e\} \\ & apply\ operator\ (\mathbf{map}\ (eval\ env)\ operands) \\ = & \quad \{\text{definition of } eval\} \\ & eval\ env\ expr \end{array}$$

For some operators, however, we follow an online strategy, short-circuiting the binding time analysis when we have enough information to determine that evaluation of the remaining operands will never be needed. The following lemma proves some useful properties of the BTA's short-circuiting.

Lemma B.1 If *bta_short_circuit env pred* $[t_1, t_2, \ldots, t_n] = S$ then $\exists m \leq n$ such that

(i) $\forall i \leq m$, *bta env* $t_i = S$,

(ii) $\forall i < m$, *pred* (*eval env* t_i) $=$ **False**, and

(iii) $m = n$ or *pred* (*eval env* t_m) $=$ **True**.

Proof. We proceed by induction on n. If $n = 0$ then $m = 0$ gives us the result vacuously. For the inductive step suppose the result holds for $n = k$ and the precondition is satisfied for $n = k + 1$. Hence we must have *bta env* $t_1 = S$. If *pred* (*eval env*$_s$ t_1) $=$ **True** then, by IH on t_1, we can take $m = 1$. Otherwise *bta_short_circuit env pred* $[t_1, \ldots, t_{k+1}] =$ *bta_short_circuit env pred* $[t_2, \ldots, t_{k+1}]$, and we can then apply the inductive hypothesis of this lemma to obtain a suitable m. □

Now when *operator* = *And*, using the lemma with *pred* = **not**,

$$\begin{array}{cl}
 & \textit{eval env}_s\ \textit{expr} \\
= & \quad \{\text{definition of } \textit{eval}\} \\
 & \textbf{foldr}\ (\&\&)\ \textbf{True}\ (\textbf{map}\ (\textit{eval env}_s)\ [t_1, \ldots, t_n]) \\
= & \quad \{*,\ \text{using IH on } t_i\ \forall i \leq m \text{ by lemma part (i)}\} \\
 & \textbf{foldr}\ (\&\&)\ \textbf{True}\ (\textbf{map}\ (\textit{eval env}_s)\ [t_1, \ldots, t_m]) \\
= & \quad \{\text{IH on } t_i\ \forall i \leq m\} \\
 & \textbf{foldr}\ (\&\&)\ \textbf{True}\ (\textbf{map}\ (\textit{eval env})\ [t_1, \ldots, t_m]) \\
= & \quad \{*\} \\
 & \textbf{foldr}\ (\&\&)\ \textbf{True}\ (\textbf{map}\ (\textit{eval env})\ [t_1, \ldots, t_n]) \\
= & \quad \{\text{definition of } \textit{eval}\} \\
 & \textit{eval env expr}
\end{array}$$

where (*) uses the lemma part (iii) together with properties of && and **foldr**.

For *Or* the argument is identical, using || in place of &&, **False** in place of **True**, and *pred* = **id**.

For *Piecewise* expressions the style of the argument is basically the same as for *Apply*, since there is some short-circuiting of static conditions where appropriate. The only detail that differs in more than the names of functions is that if a static condition is found to evaluate to true, the BTA must analyse the result associated with that condition, rather than simply returning S as for *Apply*. This detail does not affect the flow of the proof, however. $\square$

B.2 Partial evaluation of a single expression

Theorem 5.3 states that partial evaluation of a single expression does not change its meaning, i.e. for any environment *env* and expression *expr* within it,

$$\textit{eval env expr} = \textit{eval env } (\textit{reduce env expr}).$$

For the case where *expr* is dynamic the proof proceeded by structural induction on the form of *expr*, and the details of two cases were deferred to this appendix. Recall that in each case we write $(\textit{env}_s, \textit{env}_d) = \textit{partition env}$.

iii. Case $\textit{expr} = \textit{Apply operator operands}$:

For most operators,

$$\begin{array}{ll}
 & \textit{eval env } (\textit{reduce env expr}) \\
= & \quad \{\text{definition of } \textit{reduce}\} \\
 & \textit{eval env } (\textit{Apply operator } (\textbf{map } (\textit{reduce env}) \textit{ operands})) \\
= & \quad \{\text{definition of } \textit{eval}\} \\
 & \textit{apply operator } (\textbf{map } (\textit{eval env}) (\textbf{map } (\textit{reduce env}) \textit{ operands})) \\
= & \quad \{\text{property of } \textbf{map}\} \\
 & \textit{apply operator } (\textbf{map } ((\textit{eval env}).(\textit{reduce env})) \textit{ operands}) \\
= & \quad \{\text{IH on each operand}\} \\
 & \textit{apply operator } (\textbf{map } (\textit{eval env}) \textit{ operands}) \\
= & \quad \{\text{definition of } \textit{eval}\} \\
 & \textit{eval env expr}
\end{array}$$

There are special cases for *And*, *Or*, and *Divide*. Evaluation of the first two can short-circuit, and *reduce* applies such short-circuiting where possible. The following lemma, a kind of converse to Lemma B.1, identifies which operands can be discarded.

Lemma B.2 If *bta_short_circuit env pred* $[t_1, t_2, \ldots, t_n] = D$ then $\exists m \leq n$ such that

(i) $\forall i < m$, *bta env* $t_i = S$,

(ii) $\forall i < m$, *pred* (*eval env* t_i) $=$ **False**,

(iii) *bta env* $t_m = D$.

Proof. We proceed by induction on n. It must be the case that $n \geq 1$ else the precondition is false. If $n = 1$ then we must have *bta env* $t_1 = D$ and so we can take $m = n$. For the inductive step suppose the result holds for $n = k$ and the precondition holds for $n = k + 1$. Now if *bta env* $t_1 = D$ then we can take $m = 1$. If not we must have *pred* (*eval env$_s$* t_1) $=$ **False**, since the precondition would not hold otherwise. Hence by Theorem 5.2 *pred* (*eval env* t_1) $=$ **False**. Also,

$$\textit{bta\_short\_circuit env pred } [t_1, \ldots, t_{k+1}] = \textit{bta\_short\_circuit env pred } [t_2, \ldots, t_{k+1}],$$

and so by our inductive hypothesis $\exists m \in [2, k + 1]$ satisfying (i)–(iii), since we have seen above that the first two clauses also hold for t_1. □

Let us then consider the case of operator *And*. By the above lemma, with *pred* $=$ **not**, we have that $\exists m \leq n$ satisfying (i)–(iii). For $i < m$, since *bta env* $t_i = S$ we can apply Theorem 5.2 to (ii) to see that *pred* (*eval env$_s$* t_i) $=$ **False**. Hence

$$\begin{array}{cl}
& \textit{eval env } (\textit{reduce env expr}) \\
= & \qquad \{\text{expand } \textit{expr}\} \\
& \textit{eval env } (\textit{reduce env } (\textit{Apply And } [t_1, \ldots, t_n])) \\
= & \qquad \{\text{definition of } \textit{reduce}\text{, Lemma B.2, and the above}\} \\
& \textit{eval env } (\textit{Apply And } (\mathbf{map}\ (\textit{reduce env})\ [t_m, \ldots, t_n])) \\
= & \qquad \{\text{definition of } \textit{eval}\} \\
& \textit{apply And } (\mathbf{map}\ (\textit{eval env})\ (\mathbf{map}\ (\textit{reduce env})\ [t_m, \ldots, t_n])) \\
= & \qquad \{\text{property of } \mathbf{map}\} \\
& \textit{apply And } (\mathbf{map}\ ((\textit{eval env}).(\textit{reduce env}))\ [t_m, \ldots, t_n]) \\
= & \qquad \{\text{IH on each } t_i,\ m \leq i \leq n\} \\
& \textit{apply And } (\mathbf{map}\ (\textit{eval env})\ [t_m, \ldots, t_n]) \\
= & \qquad \{\text{definition of } \textit{apply}\} \\
& \mathbf{foldr}\ (\&\&)\ \mathbf{True}\ (\mathbf{map}\ (\textit{eval env})\ [t_m, \ldots, t_n]) \\
= & \qquad \{\text{Lemma B.2(ii) and definition of } \&\&\} \\
& \mathbf{foldr}\ (\&\&)\ \mathbf{True}\ (\mathbf{map}\ (\textit{eval env})\ [t_1, \ldots, t_n])
\end{array}$$

$$\begin{aligned}
& \textbf{foldr}\ (\&\&)\ \textbf{True}\ (\textbf{map}\ (\mathit{eval\ env})\ [t_1, \ldots, t_n]) \\
= & \quad \{\text{definition of } \mathit{apply}\} \\
& \mathit{apply\ And}\ (\textbf{map}\ (\mathit{eval\ env})[t_1, \ldots, t_n]) \\
= & \quad \{\text{definition of } \mathit{eval}\} \\
& \mathit{eval\ env\ expr}
\end{aligned}$$

Operator *Or* follows the same reasoning, replacing *And* with *Or*, && with ||, switching **True** and **False**, and using $\mathit{pred} = \textbf{id}$.

Division is handled differently—we replace division by a static expression with a multiplication by the reciprocal, since multiplication is faster to compute than division. If the denominator is dynamic, therefore, the proof is identical to the general case. With a static denominator, we have $\mathit{expr} = \mathit{Apply\ Divide}\ [n, d]$ and $\mathit{bta\ env\ d} = S$. Also, since *expr* is dynamic, $\mathit{bta\ env\ n} = D$. Thus, writing 1 for *Num* 1 ”dimensionless”, and ignoring wrapping/unwrapping of values as constant expressions,

$$\begin{aligned}
& \mathit{reduce\ \ env\ \ expr} \\
= & \quad \{\text{definition of } \mathit{reduce}\} \\
& \mathit{reduce\ env}\ (\mathit{Apply\ Times}\ [n, \mathit{Apply\ Divide}\ [1, d]]) \\
= & \quad \{\text{definition of } \mathit{reduce}\} \\
& \mathit{Apply\ Times}\ [\mathit{reduce\ env\ n}, \mathit{eval\ env}_s\ (\mathit{Apply\ Divide}\ [1, d])] \\
= & \quad \{\text{Theorem 5.2}\} \\
& \mathit{Apply\ Times}\ [\mathit{reduce\ env\ n}, \mathit{eval\ env}\ (\mathit{Apply\ Divide}\ [1, d])] \\
= & \quad \{\text{definition of } \mathit{eval}\} \\
& \mathit{Apply\ Times}\ [\mathit{reduce\ env\ n}, 1/(\mathit{eval\ env\ d})]
\end{aligned}$$

Therefore,

$$\begin{aligned}
& \mathit{eval\ env}\ (\mathit{reduce\ env\ expr}) \\
= & \quad \{\text{definition of } \mathit{eval} \text{ and the above}\} \\
& \textbf{foldl1}\ (*)\ [\mathit{eval\ env}\ (\mathit{reduce\ env\ n}), \mathit{eval\ env}\ (1/(\mathit{eval\ env\ d}))] \\
= & \quad \{\text{IH on } n\} \\
& \textbf{foldl1}\ (*)\ [\mathit{eval\ env\ n}, \mathit{eval\ env}\ (1/(\mathit{eval\ env\ d}))] \\
= & \quad \{\text{evaluation of a constant}\} \\
& \textbf{foldl1}\ (*)\ [\mathit{eval\ env\ n}, 1/(\mathit{eval\ env\ d})] \\
= & \quad \{\text{definition of } \textbf{foldl1}\} \\
& (\mathit{eval\ env\ n}) * (1/(\mathit{eval\ env\ d})) \\
= & \quad \{\text{properties of arithmetic}\} \\
& (\mathit{eval\ env\ n})\ /\ (\mathit{eval\ env\ d}) \\
= & \quad \{\text{definitions of } \mathit{eval} \text{ and } \mathit{apply}\} \\
& \mathit{eval\ env}\ (\mathit{Apply\ Divide}\ [n, d]) \\
= & \quad \{\text{definition of } \mathit{eval}\} \\
& \mathit{eval\ \ env\ \ expr}
\end{aligned}$$

iv. Case *expr* = *Piecewise cases* **Nothing**:

This case is made more complex by the partially-online short-circuiting of the reduction, where initial cases with static conditions evaluating to **False** are discarded, and if the then initial case has a static **True** condition the whole expression is replaced by its associated result.

Suppose for a contradiction that all the conditions are static, and for each condition /tc, *eval env c* = **False**. Then from the definition of *bta*, *bta env expr* = S. However, *expr* is dynamic, so it must consist of at least 1 condition which is not statically **False**. We therefore have

1) $n \geq 0$ cases with static conditions which evaluate to **False**;
2) a case (*Case cond res*) that either
   a) has a static condition evaluating to **True**, or
   b) has a dynamic condition; and
3) $m \geq 0$ further cases.

It is straightforward to show by induction on n that we can assume w.l.o.g. that $n = 0$. We then have two possibilities to consider.

a) In this case

$$\begin{array}{cl} & \textit{eval env } (\textit{reduce env expr}) \\ = & \qquad \{\text{definition of } \textit{reduce}\} \\ & \textit{eval env } (\textit{reduce env res}) \\ = & \qquad \{\text{IH on } \textit{res}\} \\ & \textit{eval env res} \\ = & \qquad \{\text{definition of } \textit{eval}\} \\ & \textit{eval env expr} \end{array}$$

b) Let the case expressions in (2) and (3) together be named *cases*. We first need a subsidiary result, which essentially proves the theorem for a single case expression.

Lemma B.3 Let c = *Case cond res* be any case expression in *cases*. Then for any *x*, *ecase env* (*rcase env c*) *x* = *ecase env c x*.

Proof. There are three possibilities to consider:

(1) $eval\ env\ cond = \mathbf{True}$

$$\begin{array}{ll}
 & ecase\ env\ (rcase\ env\ c)\ x \\
= & \quad \{\text{definition of } rcase\} \\
 & ecase\ env\ (Case\ (reduce\ env\ cond)\ (reduce\ env\ res))\ x \\
= & \quad \{\text{IH on } cond \text{ and definition of } ecase\} \\
 & \mathbf{Just}\ (eval\ env\ (reduce\ env\ res)) \\
= & \quad \{\text{IH on } res\} \\
 & \mathbf{Just}\ (eval\ env\ res) \\
= & \quad \{\text{definition of } ecase\} \\
 & ecase\ env\ c\ x
\end{array}$$

(2) $eval\ env\ cond = \mathbf{False}$

$$\begin{array}{ll}
 & ecase\ env\ (rcase\ env\ c)\ x \\
= & \quad \{\text{definition of } rcase\} \\
 & ecase\ env\ (Case\ (reduce\ env\ cond)\ (reduce\ env\ res))\ x \\
= & \quad \{\text{IH on } cond \text{ and definition of } ecase\} \\
 & x \\
= & \quad \{\text{definition of } ecase\} \\
 & ecase\ \ env\ \ c\ \ x
\end{array}$$

(3) $eval\ env\ cond = \bot$

$$\begin{array}{ll}
 & ecase\ env\ (rcase\ env\ c)\ x \\
= & \quad \{\text{definition of } rcase\} \\
 & ecase\ env\ (Case\ (reduce\ env\ cond)\ (reduce\ env\ res))\ x \\
= & \quad \{\text{IH on } cond \text{ and definition of } ecase\} \\
 & \bot \\
= & \quad \{\text{definition of } ecase\} \\
 & ecase\ \ env\ \ c\ \ x
\end{array}$$

□

We can now proceed to the main result.

$$\begin{array}{ll}
 & eval\ env\ (reduce\ env\ expr) \\
= & \quad \{\text{definition of } reduce\} \\
 & eval\ env\ (Piecewise\ (\mathbf{map}\ (rcase\ env)\ cases)\ \mathbf{Nothing}) \\
= & \quad \{\text{definition of } eval\} \\
 & \mathbf{foldr}\ (ecase\ env)\ \mathbf{Nothing}\ (\mathbf{map}\ (rcase\ env)\ cases) \\
= & \quad \{\text{fold-map fusion}\} \\
 & \mathbf{foldr}\ (\lambda c\ x \rightarrow ecase\ env\ (rcase\ env\ c)\ x)\ \mathbf{Nothing}\ cases \\
= & \quad \{\text{Lemma B.3}\} \\
 & \mathbf{foldr}\ (ecase\ env)\ \mathbf{Nothing}\ cases \\
= & \quad \{\text{definition of } eval\} \\
 & eval\ env\ expr
\end{array}$$

C

CellML Interpreter

```
module CellML where

import Environment
import Units
import Data.List (nub)
import Maybe

-- Definition of CellML in Haskell.

-- This makes various simplifying assumptions.
-- Further work could include removing these.
-- Examples:
--   All equations are explicit (this is the main one)
--   The CellML file describes an ODE system
--     (it must not be degenerate, ie have only algebraic equations)
--   Every variable is assumed to be real valued (ie a Double)
--     (this assumption is in CellML, too)
--     (results of logical operators can be boolean, however)
--   No support for (definite) integrals
--   Only allows first derivatives

-- The style of semantics is that of a lazy functional language, with
-- expressions evaluated on demand, ie when the value of the variable
-- to which the expression is bound is required.
-- This reflects the declarative nature of CellML, and contrasts with
-- generated procedural code, where a topological sort is used to
-- determine an evaluation order for expressions.

-- ------------------------------------------------------------
-- CellML datatypes
-- ------------------------------------------------------------

-- Identifiers are considered to be strings - we don't restrict the
-- allowable characters at all, since we're not concerned here about
-- validation of the XML representation.
type Ident = String

-- The top level datatype for a CellML model
data CellML
    = Model Ident [UDef] [Component] [Connection]
  deriving (Eq, Show)

-- Here, connections are directional, so we don't need to know about
-- the encapsulation hierarchy.  Map goes from left to right.
data Connection = VarMap (Ident, Ident) (Ident, Ident)
  deriving (Eq, Show)

-- A component, identified by name, contains units definitions and mathematics.
-- Constant variables are represented by an assignment expression.
data Component
    = Component Ident [UDef] [VarDecl] [MathAssignment]
  deriving (Eq, Show)

-- A variable declaration, associating some units with the variable name.
data VarDecl
    = VarDecl Ident UName
  deriving (Eq, Show)

-- Reference to units of constants:
-- either a name lookup or anonymous units.
type URef = Either UName Units

-- ----------------------------------------------------------
-- Mathematics datatypes
-- ----------------------------------------------------------


data MathAssignment
    = Assign EnvKey MathTree
  deriving (Eq, Show)

data MathTree
    = Num Double URef    -- A cn element
    | Bool Bool          -- The result of a relational or logical operator
    | Variable Ident     -- A ci element
    | Apply Operator [MathTree]            -- An apply element
    | Piecewise [Case] (Maybe MathTree) -- A piecewise element
    | Diff Ident Ident                     -- Diff v1 v2 = d(v1)/d(v2)
  deriving (Eq, Show)

-- Piecewise cases: Case condition result
data Case
    = Case MathTree MathTree
  deriving (Eq, Show)

-- Operator names here are capitalised MathML element names.
data Operator
    = Plus | Minus | Times | Divide
    | Exp | Log | Ln
    | Sin | Cos | Tan
    | Sqrt | Root | Power
    | And | Or | Not | Xor
    | Lt | Gt | Leq | Geq
    | Floor
  deriving (Eq, Show)

-- ----------------------------------------------------------
-- The CellML value space
-- ----------------------------------------------------------

-- The DynamicMarker is used to indicate to the partial evaluator that
-- a variable is explicitly marked dynamic, without overriding its
-- definition in the model.
data Value
    = Number Double
    | Boolean Bool
    | DynamicMarker
  deriving (Eq, Show)

-- ----------------------------------------------------------
-- A CellML environment
-- ------------------------------------------------------------

type Env = Environment EnvKey EnvValue
data EnvKey
     = Var Ident
     | Ode Ident Ident
  deriving (Eq, Show, Ord)
data EnvValue
     = Expr MathTree
     | Val Value
     | InternalData (VarUnitsEnv, UnitsEnvs, Env)

-- ------------------------------------------------------------
-- Some convenience functions for Env and Value
-- ------------------------------------------------------------

-- Helper functions to extract the Haskell value from a CellML value.
get_num :: Value → Double
get_num (Number n) = n
get_num v = error ("not a number: " ++ show v)
get_bool :: Value → Bool
get_bool (Boolean b) = b
get_bool v = error ("not a boolean: " ++ show v)

mgn = map get_num
mgb = map get_bool

-- Extracting info from an EnvValue
get_expr :: EnvValue → MathTree
get_expr (Expr t) = t
get_expr ev = error ("not an expression: " ++ show ev)
get_val :: EnvValue → Value
get_val (Val v) = v
get_val ev = error ("not a value: " ++ show ev)

-- Display of CellML environments
instance Show (EnvValue) where
  show (Expr t) = "Expr: " ++ show t
  show (Val v)  = "Val: " ++ show v
  show _ = error "unable to show internal data"

show_env :: Env → IO()
show_env env = putStr (env2str env)
env2str :: Env → String
env2str env = concat (map show_item (names env))
     where
       show_map :: EnvKey → EnvValue → String
       show_map k v = show k ++ " → " ++ show v ++ "λn"
       show_item :: EnvKey → String
       show_item k = case k of
         Var ""  → ""
         _       → show_map k (find env k)


-- ------------------------------------------------------------
-- Routines for interpreting a CellML model
-- ------------------------------------------------------------

-- Run a CellML model, in an environment giving values for the state variables and time,
-- returning an environment in which each deriviative is bound to a value.
-- In other words, we define the meaning of a model as evaluating the RHS of the ODE
-- system in a suitable environment.
run_cellml :: CellML → Env → Env
run_cellml model init_env
     = run_env model_env derivs
     where (derivs, model_env) = load_cellml model init_env
```

```
-- Evaluate an environment representing a CellML model.
run_env :: Env → [EnvKey] → Env
run_env model_env derivs
  = foldr eval_deriv empty_env derivs
  where -- Evaluate the RHS of a single ODE
        eval_deriv :: EnvKey → Env → Env
        eval_deriv d env = define env d (Val (elookup model_env d))

-- Build an environment to evaluate the model in,
-- and find the ODEs.
load_cellml :: CellML → Env → ([EnvKey], Env)
load_cellml model' init_env
    = (derivs, ext_env_u)
    where model_env :: Env
          model_env = process_cellml model init_env
          derivs :: [EnvKey]
          derivs = find_derivs model_env
          ext_env :: Env
          ext_env = foldr add_src_deriv model_env derivs
          -- Apply model transformations
          model = ((xform_math expand_unames) .
                   (xform_math expand_names) .
                   (xform_math (forget transform_piecewise))) model'
          -- Add in units information
          units_env = process_units model
          var_units = process_var_units model units_env
          ext_env_u = define ext_env (Var "")
                          (InternalData (var_units, units_env, init_env))

-- ------------------------------------------------------------
-- The core mathematics interpreter
-- ------------------------------------------------------------

-- Evaluate an expression to obtain its value
eval :: Env → MathTree → Value
eval env (Num n _)
    = Number n
eval env (Bool b)
    = Boolean b
-- Lookup the variable's definition and evaluate it
eval env (Variable v)
    = elookup env (Var v)
-- Lookup the ODE's definition and evaluate it
eval env (Diff var bvar)
    = elookup env (Ode var bvar)
-- Evaluation of apply depends on the operator.
-- Lazily evaluate the operands.
eval env (Apply operator operands)
    = apply operator (map (eval env) operands)
-- Evaluation of a piecewise expression short-circuits when a True
-- condition is found.
eval env (Piecewise cases Nothing)
    = case foldr ecase Nothing cases of
          Just v   → v
          Nothing → error "fallen off end of piecewise"
    where
      ecase :: Case → Maybe Value → Maybe Value
      ecase (Case cond res) rest = case eval env cond of
            Boolean False → rest
            Boolean True  → Just (eval env res) -- short-circuit
            _ → error ("conditional does not evaluate to a boolean: "
                         ++ show cond)

apply :: Operator → [Value] → Value
apply Plus operands       -- nary addition
    = Number (sum (mgn operands))
apply Minus [operand]     -- unary minus
```

```
    = Number (0 − (get_num operand))
apply Minus [a, b]          −− binary minus
    = Number ((get_num a) − (get_num b))
apply Divide [a, b]         −− binary divide
    = Number ((get_num a) / (get_num b))
apply Times operands        −− nary multiplication
    = Number (foldl1 (∗) (mgn operands))
apply Exp [operand]         −− unary exponential fn
    = Number (exp (get_num operand))
apply Log [operand, base] −− logarithm to given base
    = Number ((log (get_num operand)) / (log (get_num base)))
apply Ln [operand]          −− natural logarithm
    = Number (log (get_num operand))
apply Sin [operand]
    = Number (sin (get_num operand))
apply Cos [operand]
    = Number (cos (get_num operand))
apply Tan [operand]
    = Number (tan (get_num operand))
apply Sqrt [operand]
    = Number (sqrt (get_num operand))
apply Root [operand, degree]
    = Number (get_num operand ∗∗ (1/(get_num degree)))
apply Power [operand, degree]
    = Number (get_num operand ∗∗ (get_num degree))
apply And operands     −− nary logical and
    = Boolean (and (mgb operands))
apply Or operands      −− nary logical inclusive or
    = Boolean (or (mgb operands))
apply Not [operand]  −− logical negation
    = Boolean (not (get_bool operand))
apply Xor operands     −− nary logical exclusive or
    = Boolean (foldl xor False (mgb operands))
    where xor a b = (a && (not b)) || (not a && b)
apply Lt [a, b]
    = Boolean (get_num a < (get_num b))
apply Gt [a, b]
    = Boolean (get_num a > (get_num b))
apply Leq [a, b]
    = Boolean (get_num a ≤ (get_num b))
apply Geq [a, b]
    = Boolean (get_num a ≥ (get_num b))
apply Floor [a]
    = Number (fromInteger (floor (get_num a)))
apply op _
    = error ("unknown operator " ++ show op)


−− Lookup a value from an environment.  If the given name maps to
−− an expression, evaluate the expression in this environment to
−− find its value.
−− For an ODE we may need to look up using source variables if it
−− is defined in a different component from that in which it is used.
elookup :: Env → EnvKey → Value
elookup env (Ode var bvar)
    = case maybe_find env (Ode var bvar) of
        Just (Expr e) → eval env e
        Just (Val v) → v
        −− If we assume no invalid models, then we could do
        −− Nothing → elookup env ode_src
        Nothing → case find env ode_src of
          Expr e → eval env e
          Val v   → v
    where ode_src = (Ode (find_src env var) (find_src env bvar))
elookup env key
    = case find env key of
        Expr e → eval env e
        Val v → v

-- ---------------------------------------------------------------
-- Utility functions used by load_cellml, for
-- converting a CellML model into a canonical
-- CellML environment.
-- ---------------------------------------------------------------

-- Build up an environment containing all the definitions in a model.
process_cellml :: CellML → Env → Env
process_cellml (Model name us comps conns) env
    = let env' = foldr process_component env comps
          in foldr process_connection env' conns

-- Add a component's definitions to an environment.
process_component :: Component → Env → Env
process_component (Component cname _ _ assigns) env
    = foldr add_assignment env assigns
  where
    -- Add an assignment expression to an environment.
    add_assignment :: MathAssignment → Env → Env
    add_assignment (Assign key expr) env
        = define env key' (Expr expr)
        where key' = case key of
                          Var vname → Var (full_ident cname vname)
                          Ode v1 v2 → Ode (full_ident cname v1)
                                          (full_ident cname v2)

-- Add a connection to an environment, as an assignment expression.
process_connection :: Connection → Env → Env
process_connection (VarMap (cname1, vname1) (cname2, vname2)) env
    = define env key val
    where key = Var (full_ident cname2 vname2)
          val = Expr (Variable (full_ident cname1 vname1))


-- Identify the ODEs defined in an environment.
find_derivs :: Env → [EnvKey]
find_derivs e = fd (names e)
  where fd :: [EnvKey] → [EnvKey]
        fd [] = []
        fd ((Ode v1 v2):ks) = (Ode v1 v2) : fd ks
        fd (_:ks) = fd ks

-- For each derivative defined, add a definition to the environment
-- using the src variables for both independent and dependent vars.
-- This makes allowing for derivatives to appear on the RHS of
-- expressions easier.
add_src_deriv :: EnvKey → Env → Env
add_src_deriv (Ode v1 v2) env
    = if v1 == v1' && v2 == v2'
        then env
        else define env (Ode v1' v2') alias
    where v1' = find_src env v1
          v2' = find_src env v2
          alias = Expr (Diff v1 v2)


-- Apply a transformation to the (explicit) mathematics within a model.
-- The transformation of a tree also takes in the name of the component
-- within which the tree occurs.
xform_math :: (Ident → MathTree → MathTree) → CellML → CellML
xform_math xform (Model name udefs comps conns)
  = Model name udefs comps' conns
  where
    comps' = map xform_comp comps
    xform_comp (Component cname udefs vdecls assigns)
```

```
      = Component cname udefs vdecls (map (xform_assign cname) assigns)
    xform_assign cname (Assign k t)
      = Assign k (xform cname t)

-- Useful function in connection with xform_math: changes a transformation
-- that is independent of component to the necessary form.
forget :: (a → b) → (c → a → b)
forget f = λ _ → f

-- Transform piecewise expressions to remove the need for special-case handling
-- of the 'otherwise' clause, by adding it in (if present) as a case with
-- condition True.
transform_piecewise :: MathTree → MathTree
transform_piecewise (Piecewise cases otherwise)
  = Piecewise new_cases Nothing
  where new_cases = case otherwise of
                      Nothing → cases
                      Just t  → cases ++ [Case (Bool True) t]
transform_piecewise (Apply operator operands)
  = Apply operator (map transform_piecewise operands)
transform_piecewise leaf = leaf

-- Modify variable names in the given expression to include the name of
-- the component in which they occur.
expand_names :: Ident → MathTree → MathTree
expand_names cname = modify_leaves f
  where
    f (Variable v) = Variable (full_ident cname v)
    f (Diff v1 v2) = Diff (full_ident cname v1) (full_ident cname v2)
    f leaf         = leaf


-- ----------------------------------------------------------------
-- Some more generally useful utility functions
-- ----------------------------------------------------------------

-- Find the name of the variable from which a variable obtains its
-- value, by following connections.
-- If this isn't a 'mapped' variable, will just return its name.
find_src :: Env → Ident → Ident
find_src env v
  = case maybe_find env (Var v) of
      Just (Expr (Variable sv)) → find_src env sv
      _ → v

-- Compute the full name of a variable, including its component's name.
full_ident :: Ident → Ident → Ident
full_ident "c" n = n -- Hack to allow comparison of PE implementations
full_ident cname vname = cname ++ "," ++ vname

-- Get the component name from a full identifier.
-- If given a local identifier, returns "".
get_component_name :: Ident → Ident
get_component_name n
  = if pre_comma == n then "" else pre_comma
    where pre_comma = takeWhile (≠ ',') n

-- Get the variable name from a full identifier.
-- If given a local identifier, returns it verbatim.
get_variable_name :: Ident → Ident
get_variable_name n
  = if post_comma == "" then n else post_comma
    where post_comma = ((drop 1) . (dropWhile (≠ ','))) n

-- Generate a list of variables/ODEs looked up in an environment.
lookups :: Env → [EnvKey]
lookups env = foldr_expr lookups_expr [] env
```

```
  where
    lookups_expr :: MathTree → [EnvKey] → [EnvKey]
    lookups_expr (Variable v) ls = (Var v):ls
    lookups_expr (Diff v1 v2) ls = (Ode v1 v2):ls
    lookups_expr (Apply _ ts) ls = foldr lookups_expr ls ts
    lookups_expr (Piecewise cases Nothing) ls
      = foldr lookups_expr ls (cases2list cases)
    lookups_expr _ ls = ls

-- Conversion between lists of cases and expressions.
cases2list :: [Case] → [MathTree]
cases2list [] = []
cases2list (Case cond res:cs) = cond:res:(cases2list cs)
list2cases :: [MathTree] → [Case]
list2cases [] = []
list2cases (cond:res:ts) = (Case cond res) : (list2cases ts)


-- ------------------------------------------------------------
-- Various higher-order utility functions
-- for processing (CellML) environments.
-- ------------------------------------------------------------


-- A right fold over an Environment.
foldr_env :: (Ord k, Show k) ⇒
             (k → v → b → b) → b → Environment k v → b
foldr_env f z env = foldr g z (names env)
  where g key z = f key (find env key) z

-- Filter an environment according to some predicate on keys or values
-- (or both).
filter_env :: (EnvKey → EnvValue → Bool) → Env → Env
filter_env pred env = foldr_env f empty_env env
  where f :: EnvKey → EnvValue → Env → Env
        f k v e = if pred k v then define e k v else e

-- Apply a function to each 'real' entry in an environment.
-- The first argument determines what is a 'real' entry.
map_env :: (Ord k, Eq k, Show k) ⇒
           (k → Bool) → (k → v → a) → Environment k v → [a]
map_env pred f env = map do_f (filter pred (names env))
  where do_f key = f key (find env key)

-- Fold a function over all expressions in an environment.
foldr_expr :: (MathTree → a → a) → a → Env → a
foldr_expr f a env
  = foldr f a (map get_expr
                 (filter real_expr
                    (map (find env) (names env))))
  where real_expr :: EnvValue → Bool
        real_expr (Expr _) = True
        real_expr _        = False

-- Fold a function over all expressions in an environment
-- (where the function also takes the key).
foldr_expr_key :: (MathTree → EnvKey → a → a) → a → Env → a
foldr_expr_key f a env
  = foldr (uncurry f) a
      (map get_expr_key (filter real_expr (names env)))
  where real_expr :: EnvKey → Bool
        real_expr key = case find env key of
                          (Expr _) → True
                          _        → False
        get_expr_key :: EnvKey → (MathTree, EnvKey)
        get_expr_key key = (get_expr (find env key), key)

-- Apply a transformation to all expressions in an environment.
```

```
modify_exprs :: (MathTree → MathTree) → Env → Env
modify_exprs f env = foldr_expr_key g env env
  where g expr key env' = define env' key (Expr (f expr))

–– Apply a transformation to all leaf nodes in an expression tree.
modify_leaves :: (MathTree → MathTree) → MathTree → MathTree
modify_leaves f (Apply operator operands)
  = Apply operator (map (modify_leaves f) operands)
modify_leaves f (Piecewise cases Nothing)
  = Piecewise (map modify_case cases) Nothing
  where modify_case (Case t1 t2)
          = Case (modify_leaves f t1) (modify_leaves f t2)
modify_leaves f leaf = f leaf


–– ––––––––––––––––––––––––––––––––––––––––––––––––
–– Working with units in CellML
–– ––––––––––––––––––––––––––––––––––––––––––––––––

–– Units can be defined in either components, or at the model level.
–– When a units definition is looked up, we must first look to see if
–– it was defined in this component, and only if not do we look at the
–– model level, then in the standard units.

–– The different scopes for units definitions.
–– The keys are component names, with special names starting with a .
type UnitsEnvs = Environment Ident UnitsEnv

–– An environment containing just the pre–defined units.
standard_uenvs = define empty_env ".standard" standard_units


–– A mapping between variable names and their units
type VarUnitsEnv = Environment Ident (UName, Units)

–– Extracting different units environments:
–– Model level definitions.
model_units :: UnitsEnvs → UnitsEnv
model_units uenvs = find uenvs ".model"
–– Component level definitions.
component_units :: UnitsEnvs → Ident → UnitsEnv
component_units uenvs cname
  = if cname == "" then empty_env
    else case maybe_find uenvs cname of
      Just env → env
      Nothing  → error (show cname ++ " is not a component")

–– Get the units associated with a variable
variable_units :: VarUnitsEnv → Ident → Units
variable_units vuenv vname = snd (find vuenv vname)

–– Get the name of the units associated with a variable
variable_units_name :: VarUnitsEnv → Ident → UName
variable_units_name vuenv vname = fst (find vuenv vname)

–– Lookup a units definition from within the given component.
lookup_units :: UnitsEnvs → Ident → UName → Units
lookup_units envs cname uname
  = case maybe_find (component_units envs cname) uname of
        Just u  → u
        Nothing → case maybe_find (model_units envs) uname of
                      Just u  → u
                      Nothing → find standard_units uname

lookup_uref :: UnitsEnvs → URef → Units
lookup_uref uenvs = either lookup_uname id
  where lookup_uname uname
```

```
        = lookup_units uenvs (get_component_name uname)
                             (get_variable_name uname)

-- Extract the units definitions from a CellML model
process_units :: CellML → UnitsEnvs
process_units (Model _ m_udefs comps _)
  = foldr proc_comp model_env comps
  where
    model_env = define standard_uenvs ".model"
                (foldr proc_udef empty_env m_udefs)
    proc_comp :: Component → UnitsEnvs → UnitsEnvs
    proc_comp (Component cname c_udefs _ _) env
        = define env cname (foldr proc_udef empty_env c_udefs)
    proc_udef :: UDef → UnitsEnv → UnitsEnv
    proc_udef (UDef uname u) env = define env uname u

-- Create a mapping between variables and their units
process_var_units :: CellML → UnitsEnvs → VarUnitsEnv
process_var_units (Model _ _ comps _) uenv
  = foldr proc_comp empty_env comps
  where proc_comp :: Component → VarUnitsEnv → VarUnitsEnv
        proc_comp (Component cname _ vdecls _) env
          = foldr (proc_decl cname) env vdecls
        proc_decl :: Ident → VarDecl → VarUnitsEnv → VarUnitsEnv
        proc_decl cname (VarDecl vname uname) env
          = define env (full_ident cname vname)
                       (full_ident cname uname,
                        (lookup_units uenv cname uname))

-- Expand units names in numbers within mathematics, so they include the
-- component name.
expand_unames :: Ident → MathTree → MathTree
expand_unames cname = modify_leaves f
  where
    f (Num n uref) = Num n (expand_uname cname uref)
    f leaf         = leaf

expand_uname :: Ident → URef → URef
expand_uname cname
  = either (Left . full_ident cname) (Right . id)

-- Evaluate the units of an expression
eval_units_in :: Env → MathTree → Units
eval_units_in env expr = eval_units vuenv uenv env expr
  where InternalData (vuenv, uenv, _) = find env (Var "")

eval_units :: VarUnitsEnv → UnitsEnvs → Env → MathTree → Units
eval_units vuenv uenv env expr = case expr of
 (Num _ uref)     → lookup_uref uenv uref
 (Bool _)         → boolean
 (Variable v)     → variable_units vuenv v
 (Apply operator operands)
                  → apply_units operator
                                (map (eval_units vuenv uenv env) operands)
                                operands
 (Piecewise cases Nothing)
                  → piecewise_units cases
 (Diff var bvar) → quotient (variable_units vuenv var)
                            (variable_units vuenv bvar)
 where
    eqp :: Units → Units → Units
    eqp u1 u2 = if dim_equiv u1 u2
                then u1 else error "dimension mismatch"
    boolp :: Units → Units → Units
    boolp u1 u2 = if u1 == boolean && u2 == boolean then boolean
                  else error "operands are not boolean"
    ifu :: Units → [Units] → Units → Units
ifu u us res = if foldr1 eqp (u:us) == u
                 then res else error "dimension error"
dlessp :: [Units] → Units
dlessp us = ifu dimensionless us dimensionless

apply_units :: Operator → [Units] → [MathTree] → Units
apply_units Plus us _  = foldr1 eqp us
apply_units Minus us _ = foldr1 eqp us
apply_units Times us _ = foldr1 otimes us
apply_units Divide [u1, u2] _ = simplify (quotient u1 u2)
apply_units Exp [u] _    = dlessp [u]
apply_units Log [u, b] _ = dlessp [u, b]
apply_units Ln [u] _     = dlessp [u]
apply_units Sin [u] _    = dlessp [u]
apply_units Cos [u] _    = dlessp [u]
apply_units Tan [u] _    = dlessp [u]
apply_units Sqrt [u] _   = exponentiate u (0.5)
apply_units Root [u, du] [_, d]
  = ifu dimensionless [du] (exponentiate u (1 / get_num (eval env d)))
apply_units Power [u, du] [_, d]
  = ifu dimensionless [du] (exponentiate u (get_num (eval env d)))
apply_units And us _ = foldr1 boolp us
apply_units Or  us _ = foldr1 boolp us
apply_units Not us _ = foldr1 boolp us
apply_units Xor us _ = foldr1 boolp us
apply_units Lt  (u:us) _ = ifu u us boolean
apply_units Gt  (u:us) _ = ifu u us boolean
apply_units Leq (u:us) _ = ifu u us boolean
apply_units Geq (u:us) _ = ifu u us boolean

piecewise_units :: [Case] → Units
piecewise_units cases
  = if ifu boolean conds boolean == boolean then foldr1 eqp us
    else error "piecewise conditions are not boolean"
  where (conds, us) = unzip (map get_upair cases)
        get_upair (Case cond res) = (e cond, e res)
        e = eval_units vuenv uenv env
```

D

Units Algorithms

```
module Units (
  Units (BaseUnits, SimpleUnits, ComplexUnits),
  UnitsReference (Unit),
  UDef (UDef), UName,
  dimensionless, boolean,
  expand, simplify, otimes, dim_equiv, exponentiate, quotient, equal_units,
  UnitsEnv, standard_units
) where

-- A formalisation of the CellML model of physical units.

import Environment


-- The type for units
data Units = ComplexUnits [UnitsReference]
      | SimpleUnits Multiplier Prefix Units Offset
      | BaseUnits UName
  deriving (Show, Ord)

data UnitsReference = Unit Multiplier Prefix Units Exponent
  deriving (Show, Ord)

type Multiplier = Double
type Prefix     = Double
type Exponent   = Double
type Offset     = Double
type UName      = String

-- A units definition for use in a CellML model,
-- associating an identifier with some units.
data UDef = UDef UName Units
  deriving (Eq, Show)

-- An environment of such units definitions,
-- mapping identifiers to units.
type UnitsEnv = Environment UName Units


-- Equality of units definitions.
-- We need to compare floating point values within a tolerance.
instance Eq (Units) where
  (BaseUnits n1) == (BaseUnits n2)
```

```
    = n1 == n2
  (SimpleUnits m1 p1 u1 o1) == (SimpleUnits m2 p2 u2 o2)
    = (eq_delta m1 m2) && (eq_delta p1 p2) && (eq_delta o1 o2) && (u1 == u2)
  (ComplexUnits us1) == (ComplexUnits us2)
    = us1 == us2
  _ == _ = False

instance Eq (UitsReference) where
  (Unit m1 p1 u1 e1) == (Unit m2 p2 u2 e2)
    = (eq_delta m1 m2) && (eq_delta p1 p2) && (eq_delta e1 e2) && (u1 == u2)

-- Approximate equality of two doubles
eq_delta :: Double → Double → Bool
eq_delta d1 d2 = abs(d1 - d2) < 1e-12


-- Whether two definitions are the same, when canonicalised
equal_units :: Units → Units → Bool
equal_units u1 u2
    = simplify (expand u1) == simplify (expand u2)


-- Special units
dimensionless = BaseUnits "dimensionless"
boolean = BaseUnits "cellml:boolean"

-- Multiplicative factor for a units reference
mfac :: Double → Double → Double → Double
mfac m p e = m * (10**p)**e


-- Canonicalisation: expansion to products of powers of base units
expand :: Units → Units
expand (BaseUnits n) = BaseUnits n
expand (SimpleUnits m p u o) =
  case expand u of
      BaseUnits n
        → SimpleUnits m p (BaseUnits n) o
      SimpleUnits m' p' u' o'
        → SimpleUnits (m*10**p * m'*10**p') 0 u' (o + o'/(m*10**p))
expand (ComplexUnits us) = ComplexUnits (expand' us)
  where
    expand' [] = []
    expand' ((Unit m p u e):urefs) = new_urefs ++ expand' urefs
      where
        new_urefs = case expand u of
          BaseUnits n → [Unit m p u e]
          SimpleUnits m' p' u' _ → prop_mf [Unit (m'**e) p' u' e]
          ComplexUnits urefs' → prop_mf (mergesort (<) (map prop_e urefs'))
            where prop_e (Unit m' p' u' e')
                    = Unit ((mfac m' p' e')**e) 0 u' (e*e')
        prop_mf [] = []
        prop_mf ((Unit m' p' u' e'):us)
          = (Unit ((mfac m p e) * m') p' u' e'):us

-- Canonicalisation: collection of references to same units
-- The Multiplier in a SimplifyEnv stores the whole multiplicative factor
-- (i.e. mfac m p e).
type SimplifyEnv = Environment Units (Exponent, Multiplier)
simplify :: Units → Units
simplify (BaseUnits n) = BaseUnits n
simplify (SimpleUnits m p u o)
  = SimpleUnits m p u o
simplify (ComplexUnits urefs)
  = new_units
  where
    -- Step 1
    units_map :: SimplifyEnv
    units_map = foldr add_ref empty_env urefs
    add_ref :: UnitsReference → SimplifyEnv → SimplifyEnv
    add_ref (Unit m p u e) env = case maybe_find env u of
        Nothing → define env u (e, mfac m p e)
        Just (e', m') → define env u (e+e', m'*(mfac m p e))
    −− Step 1(b) − convert u^0 to d'less
    units_map' = foldr remove_zero_powers empty_env (names units_map)
    remove_zero_powers :: Units → SimplifyEnv → SimplifyEnv
    remove_zero_powers u env = case find units_map u of
        (0, m) → add_ref (Unit m 0 dimensionless 1) env
        (e, m) → define env u (e, m)
    −− Step 2 − remove 1*d'less if present
    units_map'' = case maybe_find units_map' dimensionless of
        Just (e, 1) → copy_without dimensionless units_map'
        _ → units_map'
    copy_without :: Units → SimplifyEnv → SimplifyEnv
    copy_without u env = foldr copy_item empty_env (names env)
      where copy_item u' env' = if u == u' then env'
                                else define env' u' (find env u')
    −− Step 3 − generate new units.  List of references is sorted according
    −− to the ordering on the Units datatype.
    new_units = if is_empty_env units_map'' then dimensionless else
                ComplexUnits new_urefs
    new_urefs = map new_uref (names units_map'')
    new_uref :: Units → UnitsReference
    new_uref u = Unit m 0 u e
      where (e, m) = find units_map'' u


−− Convert any units definition to be a ComplexUnits definition
make_complex (SimpleUnits m p u o) =
  ComplexUnits [Unit m p u 1]
make_complex (BaseUnits n) =
  ComplexUnits [Unit 1 0 (BaseUnits n) 1]
make_complex u = u

−− Multiplication of units definitions
otimes :: Units → Units → Units
otimes u1 u2 = simplify u
  where
    ComplexUnits u1' = make_complex u1
    ComplexUnits u2' = make_complex u2
    u = ComplexUnits (u1' ++ u2')

−− Dimensional equivalence of units
dim_equiv :: Units → Units → Bool
dim_equiv u1 u2 = sorted_exps1 == sorted_exps2
  where
    ComplexUnits urefs1 = make_complex (simplify (expand u1))
    ComplexUnits urefs2 = make_complex (simplify (expand u2))
    exps1 = map extract urefs1
    exps2 = map extract urefs2
    sorted_exps1 = mergesort pred exps1
    sorted_exps2 = mergesort pred exps2
    extract (Unit _ _ (BaseUnits n) e) = (n, e)
    pred (n1, _) (n2, _) = n1 ≤ n2

−− Exponentiation of units
exponentiate :: Units → Exponent → Units
exponentiate (BaseUnits n) exp =
  ComplexUnits [Unit 1 0 (BaseUnits n) exp]
exponentiate (SimpleUnits m p u o) exp =
  ComplexUnits [Unit m p u exp]
exponentiate (ComplexUnits urefs) exp =
  ComplexUnits (map expo urefs)
   where
```

```
    expo (Unit m p u e)
         = Unit (m**exp) p u (e*exp)

-- The quotient of two units
quotient :: Units → Units → Units
quotient u1 u2 = ComplexUnits [Unit 1 0 u1 1, Unit 1 0 u2 (-1)]


-- SI prefixes
tera  =  12 :: Prefix
giga  =  9  :: Prefix
mega  =  6  :: Prefix
kilo  =  3  :: Prefix
hecto =  2  :: Prefix
deka  =  1  :: Prefix
deci  = -1  :: Prefix
centi = -2  :: Prefix
milli = -3  :: Prefix
micro = -6  :: Prefix
nano  = -9  :: Prefix
pico  = -12 :: Prefix

-- Units from the CellML standard dictionary
kelvin = BaseUnits "kelvin"
metre = BaseUnits "metre"
second = BaseUnits "second"
kilogram = BaseUnits "kilogram"
ampere = BaseUnits "ampere"
candela = BaseUnits "candela"
mole = BaseUnits "mole"

radian = ComplexUnits [Unit 1 0 metre 1, Unit 1 0 metre (-1)]
steradian = ComplexUnits [Unit 1 0 metre 2, Unit 1 0 metre (-2)]
hertz = ComplexUnits [Unit 1 0 second (-1)]
newton = ComplexUnits [Unit 1 0 kilogram 1, Unit 1 0 metre 1,
                       Unit 1 0 second (-2)]
pascal = ComplexUnits [Unit 1 0 newton 1, Unit 1 0 metre (-2)]
joule = ComplexUnits [Unit 1 0 newton 1, Unit 1 0 metre 1]
watt = ComplexUnits [Unit 1 0 joule 1, Unit 1 0 second (-1)]
coulomb = ComplexUnits [Unit 1 0 second 1, Unit 1 0 ampere 1]
volt = ComplexUnits [Unit 1 0 watt 1, Unit 1 0 ampere (-1)]
farad = ComplexUnits [Unit 1 0 coulomb 1, Unit 1 0 volt (-1)]
ohm = ComplexUnits [Unit 1 0 volt 1, Unit 1 0 ampere (-1)]
siemens = ComplexUnits [Unit 1 0 ampere 1, Unit 1 0 volt (-1)]
weber = ComplexUnits [Unit 1 0 volt 1, Unit 1 0 second 1]
tesla = ComplexUnits [Unit 1 0 weber 1, Unit 1 0 metre (-2)]
henry = ComplexUnits [Unit 1 0 weber 1, Unit 1 0 ampere (-1)]
celsius = SimpleUnits 1 0 kelvin (-273.15)
lumen = ComplexUnits [Unit 1 0 candela 1, Unit 1 0 steradian 1]
lux = ComplexUnits [Unit 1 0 lumen 1, Unit 1 0 metre (-2)]
becquerel = ComplexUnits [Unit 1 0 second (-1)]
gray = ComplexUnits [Unit 1 0 joule 1, Unit 1 0 kilogram (-1)]
sievert = ComplexUnits [Unit 1 0 joule 1, Unit 1 0 kilogram (-1)]
katal = ComplexUnits [Unit 1 0 second (-1), Unit 1 0 mole 1]

gram = SimpleUnits (0.001) 0 kilogram 0
litre = ComplexUnits [Unit 1000 centi metre 3]

standard_dictionary_list
  = [ -- Special units
      UDef "dimensionless"  dimensionless
    , UDef "cellml:boolean" boolean
    -- Base units
    , UDef "kelvin"   kelvin
    , UDef "metre"    metre
    , UDef "second"   second
    , UDef "kilogram" kilogram
    , UDef "ampere"    ampere
    , UDef "candela"   candela
    , UDef "mole"      mole
    -- Other SI units
    , UDef "radian"     radian
    , UDef "steradian" steradian
    , UDef "hertz"    hertz
    , UDef "newton"   newton
    , UDef "pascal"   pascal
    , UDef "joule"    joule
    , UDef "watt"     watt
    , UDef "coulomb"  coulomb
    , UDef "volt"     volt
    , UDef "farad"    farad
    , UDef "ohm"      ohm
    , UDef "siemens"  siemens
    , UDef "weber"    weber
    , UDef "tesla"    tesla
    , UDef "henry"    henry
    , UDef "celsius"  celsius
    , UDef "lumen"    lumen
    , UDef "lux"      lux
    , UDef "becquerel" becquerel
    , UDef "gray"     gray
    , UDef "sievert"  sievert
    , UDef "katal"    katal
    , UDef "gram"  gram
    , UDef "litre" litre
    ]
standard_dictionary_aliases
  = [ ("meter", "metre"), ("liter", "litre") ]
standard_units :: UnitsEnv
standard_units = foldr add_alias initial_env standard_dictionary_aliases
    where initial_env = foldr add_udef empty_env standard_dictionary_list
          add_udef :: UDef → UnitsEnv → UnitsEnv
          add_udef (UDef name u) env = define env name u
          add_alias :: (UName, UName) → UnitsEnv → UnitsEnv
          add_alias (new_name, existing_name) env
              = define env new_name (find env existing_name)



-- Parameterised mergesort
-- Takes in a comparison function
mergesort :: (a → a → Bool) → [a] → [a]
mergesort pred []   = []
mergesort pred [x]  = [x]
mergesort pred xxs = merge (mergesort pred xs1) (mergesort pred xs2)
  where
    (xs1, xs2) = split xxs
    split xs = splitrec xs xs []
    splitrec [] ys zs             = (reverse zs, ys)
    splitrec [x] ys zs            = (reverse zs, ys)
    splitrec (x1:x2:xs) (y:ys) zs = splitrec xs ys (y:zs)
    merge xs []          = xs
    merge [] ys          = ys
    merge (x:xs) (y:ys) =
      case pred x y of
        True  → x: merge xs (y:ys)
        False → y: merge (x:xs) ys


-- ------------------------------------------------------------
-- Testing
-- ------------------------------------------------------------

-- Some useful units
```

```
liter = find standard_units "liter"
meter = find standard_units "meter"
fahrenheit = SimpleUnits (5/9) 0 celsius 32
kPa = ComplexUnits [Unit 1 kilo newton 1, Unit 1 0 metre (−2)]

−− The tests to run
tests = [ ( expand (ComplexUnits [Unit 10 0 second 2, Unit 2 0 gram (−1)])
              == ComplexUnits [Unit 10.0 0.0 (BaseUnits "second") 2.0,
                               Unit 2000.0 0.0 (BaseUnits "kilogram") (−1.0)]
            ),
            ( expand fahrenheit
              == SimpleUnits (5/9) 0.0 (BaseUnits "kelvin") (32 − 273.15 ∗ (9/5))
            ),
            ( otimes (ComplexUnits [Unit 1 0 coulomb 1, Unit 1 0 volt (−1)])
                     (ComplexUnits [Unit 1 0 volt 1])
              == ComplexUnits [Unit 1 0 coulomb 1]
            ),
            simplify radian == dimensionless,
            dim_equiv liter (ComplexUnits [Unit 1 0 metre 3]),
            ( otimes kilogram (expand (ComplexUnits [Unit 1 0 gram (−1)]))
              == ComplexUnits [Unit 1000.0 0.0 (BaseUnits "dimensionless") 1.0]
            ),
            ( expand (ComplexUnits [Unit 1 1 litre 2])
              == ComplexUnits [Unit ((mfac 1000 (−2) 3)∗∗2 ∗ mfac 1 1 2) 0 meter 6.0]
            ),
            ( expand (ComplexUnits [Unit 1 1 litre 2])
              == ComplexUnits [Unit 1e−4 0 meter 6.0]
            ),
            otimes (ComplexUnits [Unit 1 0 meter (−1)]) metre == dimensionless,
            ( simplify(expand(ComplexUnits
                [Unit 1 kilo (ComplexUnits [Unit 1 0 metre 1,
                                            Unit 1 0 kilogram 1,
                                            Unit 1 0 second (−2)]) 1,
                 Unit 1 0 metre (−2)]))
             == simplify(expand(ComplexUnits
                 [Unit 1 kilo (ComplexUnits [Unit 1 0 kilogram 1,
                                             Unit 1 0 metre 1,
                                             Unit 1 0 second (−2)]) 1,
                  Unit 1 0 metre (−2)]))
            ),
            ( simplify (ComplexUnits [Unit 1 centi metre 2, Unit 10 0 metre 1])
              == ComplexUnits [Unit 0.001 0 metre 3]
            ),
            ( otimes (ComplexUnits [Unit 1 centi meter 2, Unit 1 0 kilogram 1])
                     (ComplexUnits [Unit 10 0 metre (−1)])
              == ComplexUnits [Unit 1 0 kilogram 1, Unit 0.001 0 metre 1]
            )
          ]
run_tests = (and tests, tests)
```

E

Haskell Partial Evaluator for CellML

```
module PE where

import CellML
import Environment
import Units
import qualified Data.Set as Set
import Maybe

-- Haskell implementation of the Partial Evaluation of CellML.


-- ------------------------------------------------------------
-- Binding time analysis
-- ------------------------------------------------------------

-- This is a partially-online analysis, relying on the dependency graph
-- having no cycles.

-- A datatype for binding times.  The derived ordering gives us
-- Static < Dynamic.
data BindingTime = Static | Dynamic
  deriving (Eq, Show, Ord)

-- Partition an environment into static and dynamic portions.
partition :: Env → (Env, Env)
partition env = foldr f (empty_env, empty_env) (names env)
  where
    f :: EnvKey → (Env, Env) → (Env, Env)
    f k (env_s, env_d) =
      case bt of
        Static  → (define env_s k v, env_d)
        Dynamic → (env_s, define env_d k v)
      where
        v = find env k
        bt = bta_key env k

-- Compute the binding time of an expression.
-- This implementation is inefficient, since we do not annotate expression
-- trees with a binding time to avoid recomputation.  Termination is only
-- guaranteed if the dependency graph has no cycles.
-- We require an environment describing the whole CellML model, as well as the
-- expression to be analysed.
bta :: Env → MathTree → BindingTime
bta env (Num _ _) = Static -- constants are always static
bta env (Bool _) = Static  -- constants are always static
bta env (Variable v) -- look up the definition of the variable and analyse that
  = bta_key env (Var v)
bta env (Diff v1 v2) -- analyse the ODE definition
  = bta_key env (Ode v1 v2)
bta env (Apply operator operands)
  = bta_apply env operator operands -- see below
bta env (Piecewise cases Nothing)
  = bta_piecewise env cases          -- see below

-- Compute the binding time of whatever is bound to the given key in
-- the given environment.
bta_key :: Env → EnvKey → BindingTime
bta_key env (Var "") = Static -- arbitrary choice
bta_key env key
  = case maybe_find dyn_env key of
      Just _   → Dynamic -- state or free variable, or user annotated
      Nothing → case find env key of
                    Expr t → bta env t -- analyse the defining expression
                    Val _  → Static    -- constants are static
  where (InternalData (_,_,dyn_env)) = find env (Var "")

-- The binding time of an operator application is special-cased for
-- some operators.
bta_apply :: Env → Operator → [MathTree] → BindingTime
bta_apply env And operands -- short-circuit if static operand is False
  = bta_short_circuit env (not . get_bool) operands
bta_apply env Or operands -- short-circuit if static operand is True
  = bta_short_circuit env get_bool operands
bta_apply env _ operands -- general case: maximum of operand binding times
  = maximum (map (bta env) operands)

-- Short circuit binding time analysis if a static operand satisfies the
-- given predicate.
bta_short_circuit :: Env → (Value → Bool) → [MathTree] → BindingTime
bta_short_circuit env pred (t:ts)
  = if bta env t == Static
      then if pred (eval env t) then Static
                                else bta_short_circuit env pred ts
      else Dynamic
bta_short_circuit env pred [] = Static

-- In general we take the maximum of child binding times, but there is some
-- short-circuiting and evaluation of static conditions.
bta_piecewise :: Env → [Case] → BindingTime
bta_piecewise env (Case cond res : cs)
  = if bta env cond == Static
      then if get_bool (eval env cond) then bta env res
                                       else bta_piecewise env cs
      else Dynamic
bta_piecewise env [] = Static


-- ------------------------------------------------------------
-- Partial evaluation
-- ------------------------------------------------------------

-- Partial evaluation of a model simply reduces each ODE.
-- The provided environment contains entries for each dynamic variable.
reduce_cellml :: CellML → Env → Env
reduce_cellml model dyn_env
  = reduce_env model_env derivs dyn_env
  where (derivs, model_env) = load_cellml model dyn_env

reduce_and_run_cellml :: CellML → Env → Env
reduce_and_run_cellml model dyn_env
  = run_env reduced_model derivs
  where (derivs, model_env) = load_cellml model dyn_env
        reduced_model = reduce_env model_env derivs dyn_env

-- Partial evaluation of a CellML environment.
reduce_env :: Env → [EnvKey] → Env → Env
reduce_env model_env derivs dyn_env
  = (move_component_units . define_pe_units)
        (rec_reduce_derivs model_env)
  where idata = find model_env (Var "")
        reduce_derivs env
          = foldr (reduce_deriv env) init_env derivs
        init_env = filter_env real_value dyn_env
          where real_value _ (Val DynamicMarker) = False
                real_value _ _ = True
        rec_reduce_derivs env
          = if has_instantiable_key new_env
               then rec_reduce_derivs new_env
               else new_env
          where new_env = define
                  (head (dropWhile has_undefined_var
                          (iterate (add_reduced_definitions env)
                                   (reduce_derivs env))))
                  (Var "") idata
        -- Reduce the RHS of a single ODE
        reduce_deriv :: Env → EnvKey → Env → Env
        reduce_deriv menv d env = define env d (reduce_key menv d)

-- Reduce the definition of an EnvKey
reduce_key :: Env → EnvKey → EnvValue
reduce_key env k
  = reduce_val env (find env k)

reduce_val :: Env → EnvValue → EnvValue
reduce_val env (Expr t) = Expr (reduce env t)
reduce_val env value = value


-- The main workhorse of the partial evaluator:
-- Reduce an expression to a simpler form by evaluating static portions
-- within the given environment.
reduce :: Env → MathTree → MathTree
reduce env expr
  = case bta env expr of
      Static  → let (_,e) = eval_to_expr expr in e
      Dynamic → reduce' expr
  where
    -- Evaluate an expression to get a constant expression
    eval_to_expr :: MathTree → (Value, MathTree)
    eval_to_expr e
      = let v = eval env_s expr in
          (v, case v of
                Number n  → Num n (Right (eval_units_in env expr))
                Boolean b → Bool b
          )

    (env_s, env_d) = partition env

    -- Reduce an expression known to be dynamic.
    reduce' (Variable var)
      = reduce_lookup (Var var) (Variable var)
    reduce' (Diff v1 v2)
      = reduce_lookup (Ode v1 v2) (Diff v1 v2)
    reduce' (Piecewise cases Nothing) -- short-circuit static conditions
      = f cases
      where -- expr Dyn ⇒ at least 1 cond != Static False
        f allcs@(Case cond res : cs)
          = if bta env cond == Static
```

```
                then case eval env_s cond of
                        Boolean True  → reduce env res
                        Boolean False → f cs
                else Piecewise (map (rcase env) allcs) Nothing
          rcase env (Case cond res) = Case cond' res'
            where cond' = reduce env cond
                  res'  = reduce env res
      reduce' (Apply And operands) −− short−circuit if static operand is False
        = short_circuit And (not . get_bool) operands
      reduce' (Apply Or operands)  −− short−circuit if static operand is True
        = short_circuit Or get_bool operands
      reduce' (Apply Divide [n, d]) −− convert divide−by−static to times
        = if bta env d == Static
            then reduce env (Apply Times [n, Apply Divide [one, d]])
            else Apply Divide (reduce_list [n, d])
        where one = Num 1 (Left (full_ident ".model" "dimensionless"))
      reduce' (Apply op operands) −− reduce operands
        = Apply op (reduce_list operands)

      short_circuit :: Operator → (Value → Bool) → [MathTree] → MathTree
      short_circuit op pred (t:ts)
        = if bta env t == Static
            then if pred val then e −− never happens as expr is dynamic
                             else short_circuit op pred ts
            else Apply op (reduce_list (t:ts))
        where (val, e) = eval_to_expr t

      −− Reduce a list of expressions
      reduce_list :: [MathTree] → [MathTree]
      reduce_list exprs = map (reduce env) exprs

      −− Reduce an environment lookup (var/ODE)
      reduce_lookup key key_as_expr
        = if may_instantiate_key env key
          then reduce env e −− instantiate reduced definition
          else key_as_expr  −− retain lookup
        where Expr e = find env_d key
        −− It must be defined by an Expr since o/w it would be static


−− Determine whether we should instantiate the definition of the given key.
−− We should if
−−  (1) it is not explicitly marked dynamic; and
−−  (2) it is defined by an expresion; and
−−  (3a) it is only used once, or
−−  (3b) it is just an alias for another key.
−− Condition (2) is always true for dynamic keys (so could be omitted).
may_instantiate_key :: Env → EnvKey → Bool
may_instantiate_key env key
  = not annotated && is_expr && (used_once || alias)
  where
    InternalData (_, _, dyn_env) = find env (Var "")
    annotated = isJust (maybe_find dyn_env key)
    used_once = check_usage env key == 1
    defn = find env key
    is_expr = case defn of
      Expr _ → True
      _      → False
    alias  = case defn of
      Expr (Variable _) → True
      Expr (Diff _ _)   → True
      _                 → False


−− −−−−−−−−−−−−−−−−−−−−−−−−−−−−−
−− Checking variable usage
−− −−−−−−−−−−−−−−−−−−−−−−−−−−−−−
```

```
-- Environment tracking variable usage counts.  Counts simple usage and
-- ODE usage separately.
type UsageCounts = Environment EnvKey Int

-- Find how often a variable has been used.  Returns 0 if no count is found.
check_usage :: Env → EnvKey → Int
check_usage env key
  = case maybe_find counts key of
      Just n   → n
      Nothing → 0
  where counts = variable_usage env

-- Count variable usage in a whole model.
variable_usage :: Env → UsageCounts
variable_usage env
  = foldr_expr var_usage empty_env env

-- Count variable usage in an expression.
var_usage :: MathTree → UsageCounts → UsageCounts
var_usage (Variable v) env
  = incr_count env (Var v)
var_usage (Diff v1 v2) env
  = incr_count env (Ode v1 v2)
var_usage (Apply _ operands) env
  = foldr var_usage env operands
var_usage (Piecewise cases Nothing) env
  = foldr case_usage env cases
  where case_usage (Case cond res) env
          = var_usage cond (var_usage res env)
var_usage _ env = env

incr_count :: UsageCounts → EnvKey → UsageCounts
incr_count env k
  = case maybe_find env k of
      Just count → define env k (count + 1)
      Nothing     → define env k 1

-- ----------------------------
-- Utility functions for
-- avoiding code duplication
-- ----------------------------

-- Check if any key lookup in the model is permitted to be
-- instantiated by reduce.
has_instantiable_key :: Env → Bool
has_instantiable_key env
  = foldr (||) False (map (may_instantiate_key env) (lookups env))

-- Check if a variable/ODE is looked up but not defined
has_undefined_var :: Env → Bool
has_undefined_var env
  = not (used_set `Set.isSubsetOf` def_set)
  where def_set  = Set.fromList (names env)
        used_set = Set.fromList (lookups env)

-- Copy definitions used by the second Env but only provided in the
-- first to the second, reducing them in the process
add_reduced_definitions :: Env → Env → Env
add_reduced_definitions from_env to_env
  = foldr add_def to_env (Set.toList (Set.difference used_set def_set))
  where def_set  = Set.fromList (names to_env)
        used_set = Set.fromList (lookups to_env)
        add_def key env = define env key (reduce_key from_env key)


-- ----------------------------------------------------------
-- Adding appropriate units to the reduced model
-- ---------------------------------------------------------

-- Add all anonymous units to the environment, and change references
-- to look them up.
define_pe_units :: Env → Env
define_pe_units env
  = foldr_expr_key f env env
  where
    f :: MathTree → EnvKey → Env → Env
    f expr key env' = define env'' (Var "") idata'
     where
       (InternalData (vuenv, uenvs, dyn_env)) = find env' (Var "")
       idata' = InternalData (vuenv, uenvs', dyn_env)
       env'' = define env' key (Expr expr')
       (uenvs', expr') = def_units_expr uenvs expr

-- Process an expression tree, adding anonymous units to the units
-- environment, changing references to look them up.
def_units_expr :: UnitsEnvs → MathTree → (UnitsEnvs, MathTree)
def_units_expr uenvs (Num x uref)
  = case uref of
      Left uname  → (uenvs, Num x uref)
      Right units → (uenvs', Num x (Left uname'))
       where
        (uname', uenvs') = define_units uenvs units
def_units_expr uenvs (Apply op operands)
  = (uenvs', Apply op operands')
  where (uenvs', operands') = map_foldr def_units_expr uenvs operands
def_units_expr uenvs (Piecewise cases Nothing)
  = (uenvs', Piecewise cases' Nothing)
  where (uenvs', ts) = map_foldr def_units_expr uenvs (cases2list cases)
        cases' = list2cases ts
def_units_expr uenvs leaf = (uenvs, leaf)

-- Add units to the environment, if there are not already there.
-- Returns the name bound to these units, and the new environment.
-- If not already defined, units will be defined in the 'model' portion
-- of the environment.
define_units :: UnitsEnvs → Units → (UName, UnitsEnvs)
define_units uenvs units
  = case units_defined uenvs units of
      Just uname → (uname, uenvs)
      Nothing    → (uniq_uname,
                    define uenvs ".model"
                           (define menv uniq_uname units))
  where menv = model_units uenvs
        uniq_uname = uniq_key menv

-- Are the given units already defined?
units_defined :: UnitsEnvs → Units → Maybe UName
units_defined uenvs units
  = foldr ud' Nothing (map (find uenvs) (names uenvs))
  where
    ud' :: UnitsEnv → Maybe UName → Maybe UName
    ud' uenv rest = foldr (ud'' uenv) rest (names uenv)
    ud'' :: UnitsEnv → UName → Maybe UName → Maybe UName
    ud'' uenv uname rest
     = if find uenv uname == units then Just uname
                                   else rest

-- A little type used in move_component_units
type RenameEnv = Environment UName UName
-- Move any component-level units definitions into the model-level
-- environment.
move_component_units :: Env → Env
move_component_units env
```

```
  = modify_exprs (modify_leaves rename_urefs) env'
  where
    InternalData (vuenv, uenvs, dyn_env) = find env (Var "")
    idata' = InternalData (vuenv, new_uenvs, dyn_env)
    env' = define env (Var "") idata'
    new_uenvs = define standard_uenvs ".model" new_model_env
    (new_model_env, renamed)
        = foldr_env do_comp (model_units uenvs, empty_env) uenvs

    rename_urefs :: MathTree → MathTree
    rename_urefs (Num n (Left uname))
        = case maybe_find renamed uname of
            Just new_name → Num n (Left new_name)
            Nothing       → Num n (Left uname)
    rename_urefs leaf = leaf

    do_comp :: Ident → UnitsEnv → (UnitsEnv, RenameEnv)
            → (UnitsEnv, RenameEnv)
    do_comp cname c_uenv (uenv, renames)
      = if head cname == '.'
        then (uenv, renames)
        else foldr_env (do_udef cname) (uenv, renames) c_uenv
    do_udef :: Ident → UName → Units → (UnitsEnv, RenameEnv)
            → (UnitsEnv, RenameEnv)
    do_udef cname uname units (uenv, renames)
      = if new_name == full_name
        then (new_env, renames)
        else (new_env, define renames full_name new_name)
      where full_name = full_ident cname uname
            new_name = (head . dropWhile used)
                         (iterate (++"_") full_name)
            used name = isJust (maybe_find uenv name)
            new_env = define uenv new_name units


-- Find a unique key, unused in the given environment
uniq_key :: UnitsEnv → UName
uniq_key uenv
  = head (dropWhile used (map num2uname [0..]))
  where names_set = Set.fromList (names uenv)
        num2uname n = "___" ++ show n
        used n = Set.member n names_set

-- Combined map and (right) fold.
map_foldr :: (b → a → (b, a)) → b → [a] → (b, [a])
map_foldr f init [] = (init, [])
map_foldr f init (x:xs) = (acc, (x':xs'))
  where (acc, x')    = f rest x
        (rest, xs') = (map_foldr f init xs)
```