Models of Gene Regulatory Networks and Other Biological Systems

Andrew Keith Miller

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Bioengineering, The University of Auckland, 2011.

Abstract

Mathematical modelling of biological processes is important in systems biology, because it facilitates understanding of the nature and behaviour of biological systems. This thesis is in two parts, the first on constructing and validating Gene Regulatory Network (GRN) models, and the second on mathematical model representation.

Gene Regulatory Network Models

A method, BaSeTraM, for identifying transcription factor (TF) binding sites from position weight matrices was developed. The sites identified were used to build a GRN by identifying genes near each site. BaSeTraM performed comparably to a widely used method when validated against experimental data, with the advantage that the selectivity-sensitivity trade-off is controlled by adjusting the posterior probability.

A classifier for detecting genes with missing regulators in a GRN model was developed, using regression to convert a qualitative model into a quantitative model, with an iterative method to predict expression levels in gene knock-out strains. Errors for each gene were used to predict which genes were missing regulators. Validation of the classifier to detect regulators deleted from a yeast GRN model showed that it out-performed random guessing.

Finally, a method was developed for validating entire models by converting to a quantitative model, and predicting gene expression by regulator levels. Validating models built using BaSeTraM against human microarray data showed that degraded models had lower errors than the original in $> 50\%$ of all predictions. The bimodal distribution of per-gene proportion of higher errors suggests that the original model described some genes more accurately. This

method provides a general framework within which to validate GRN models against genome-wide gene expression data sets.

Model representation

An API for working with CellML models was developed, allowing applications to process mathematical models more easily.

A model representation language, ModML, for representing models as a transformation from a domain specific language (DSL) into a data structure describing differential-algebraic equations was developed, along with tools for performing numerical simulations from models.

Two DSLs based on ModML were developed; ModML Units, for equations with physical units, and ModML Reactions, for reaction systems. The utility of the DSLs has been demonstrated by expressing existing models in them.

The development of ModML and DSLs built on top of it mean that models describing components of a system in different ways can be more easily composed to facilitate understanding of the system.

Acknowledgements

This thesis owes a great deal to the help of my primary supervisor, A/Prof Edmund Crampin, and my co-supervisors A/Prof Cristin Print and A/Prof Poul Nielsen, all of whom provided helpful insight, input, and inspiration at all stages of the PhD project, from the initial planning stages right through to now, and so my foremost thanks goes to them. I would additionally like to thank Prof Peter Hunter for serving on the advisory board and providing advice on the project, and for encouraging me to pursue a PhD in Bioengineering in the first place.

This thesis was funded from several sources. Three years of funding was provided by a Top Achievers Doctoral Scholarship from the Tertiary Education Commission. A further six months of funding was provided by a University of Auckland University Doctoral Extension, with funding for university fees for the final six months provided by the Auckland Bioengineering Institute. I would like to thank these organisations for providing funding.

The CellML team at the Auckland Bioengineering Institute provided helpful feedback on the development of the CellML API and on ModML. In addition, various people have given me helpful feedback on the development of these projects by e-mail correspondence and at various presentations and conferences; I'd like to thank everyone who provided feedback on my work.

I'd like to thank Keith Miller for proof reading this thesis.

I'd finally like to thank my fiancée, Donna Harland, for her great patience during the course of this PhD project, and for proof-reading some chapters.

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Gene Regulation

Gene regulation is crucial to all biological systems, from single celled organisms, where the organism must rapidly adjust to a changing external environment by changing the gene expression profile, to multicellular organisms where gene regulation is crucial in many processes in both development and differentiation and the response of cells to internal and external stimuli.

The field of Systems Biology is concerned with the study of how the parts of biological systems combine and interact to create the emergent behaviour of the biological system; systems biology is an integrative discipline, merging the study of the parts of a system (reductionism) and how the system as whole behaves (holism).

Gene expression (and its regulation) is a complex process involving many chemical species. Due to the importance of gene regulation to biology, fully understanding the link between the underlying molecular processes and the emergent behaviour when all these processes are combined to form a so-called Gene Regulatory Network (GRN) is an important open area of study in systems biology.

One of the major objectives of this research was therefore to extend the methodologies available for building and extending models of gene regulatory networks.

However, simply building models of gene regulatory networks is not very useful if these methods cannot be easily tested against different types of experimental data. Testing models of gene regulatory networks is useful not just for testing

specific models fitted to data, but also for testing and refining the theory underlying the method for generating the network.

Developing methodologies for validating models of gene regulatory networks was therefore another major objective of this research.

1.2 Mathematical models of biological systems

One important aspect of systems biology is the ability to describe hypotheses about systems as mathematical models. A mathematical model is a model of a system expressed as mathematical relationships. Mathematical models are useful as they allow assumptions made when building a system to be checked for consistency with themselves and with data.

One widely used mathematical model formulation is as a system of differential-algebraic equations. This formulation is commonly used to model how a finite number of variables evolve with respect to time (or another single dimension). For example, a differential-algebraic equation might be used to describe how the expression levels of different genes, represented by a variable each (perhaps measured by protein concentration, or by an mRNA count), evolve over time.

For systems of differential-algebraic equations to be useful for computer simulation, however, they need to be in some machine readable form. This form can then be used by computational tools for processing models.

The above example of modelling gene expression is only one of many applications for models of differential-algebraic equations. There is considerable benefit to providing tools for working with systems of differential-algebraic equations that are independent of the domain of application; this means that more effort can be focused on creating additional tools to support new types of problem, rather than reinventing the same tools for each problem domain. In this thesis, existing formats for exchanging such models are reviewed.

However, despite the benefits of domain-independent representations, there are often benefits to describing systems in terminology specific to a particular

domain of application. Domain specific languages for model representation mean that information needed to interpret the variables in a model are presented together with information needed to generate a model.

These problems can be circumvented by representing models as a functional transformation from a domain specific language to a data structure representing differential-algebraic equations. One of the major objectives of this research was to develop such a representation for mathematical models.

Chapter 2

Literature Review: Modelling gene regulation

2.1 The strategic context

One of the ultimate goals of research into the life sciences is to understand human physiology in sufficient detail to allow rational treatment design and decision making on issues related to human health. In order to further our understanding of physiology, there are two types of approach that can be taken. Holistic approaches study the organism, or some organ system, as a whole, and deduce the functionality in a top-down approach. On the other hand, reductionist approaches start from the behaviour of the underlying components of the system, such as the behaviour of individual molecules, genes, or cells, and from that, take a bottom-up approach to deduce the behaviour of the system. It has been argued recently [Schuster, 2007] that to fully understand a natural system, it is necessary that both approaches be maintained; the combination of a holistic and a reductionist approach is referred to as an integrative approach.

Hunter and Nielsen [2005a] have devised a strategy for integrative computational physiology. This strategy calls for the use of computer based modelling at multiple scales, to study how, at each scale, reductionist predictions can result in the holistic observations of the next largest scale. The ultimate aim of this multiscale approach is to understand the total physiology, from molecule through to whole organism.

The role that models and their validation play in the philosophy and methodology of science is an area which has attracted considerable attention in recent times.

Barlas [1996] draws a distinction between causal-descriptive models (also known as white-box or theory-like models), and correlative models (black-box or data driven models). The paper then argues that because causal-descriptive models "are statements as to how real systems actually operate in some aspects", "generating an 'accurate' output behaviour is not sufficient for model validity; what is crucial is the validity of the internal structure of the model". While the direct outputs of the model cannot be used, the author argues for structure-orientated behaviour tests, which "assess the validity of the structure indirectly, by applying certain behaviour tests on the model-generated behaviour patterns". One example of such a test is modified behaviour prediction, in which the real system is modified, information is collected from the modified system, and a similar modification is made to the model structure. The simulated results of the modified structure are compared against the data from the real system.

2.2 Relevant aspects from molecular biology

The primary information store for genetic information is DNA. The central dogma of molecular biology, as clarified in Crick [1970], states that there are three 'general' transfers of this information which occur in every cell. These are: the replication of DNA, the synthesis of RNA from DNA (transcription), and the synthesis of protein from DNA (translation).

The majority of all catalytic activity known to occur in cells is driven by proteins (known as enzymes), and much of the remaining known catalytic activity is carried out by functionally active RNA molecules called ribozymes [Davies, 1984].

One active area of research is on how the genes present in individual cells (the genotype) gives rise to the behaviour of the cell (the phenotype). The systematic study of this problem is referred to as systems biology [Ideker et al., 2001a]. One common theme in systems biology is the representation of cellular physiology as arising from networks of entities with interactions between them.

The metabolic activities in a cell can be viewed as the result of a network of

substances, connected by the relationships these substances play (for example, as a catalyst, reactant, product, or inhibitor) in reactions occurring in the cell. A network of this type is referred to as a metabolic network [Ideker et al., 2001b]. Due to the level of effort currently involved in constructing metabolic networks, most models of metabolic networks focus on a small number of pathways at a time.

Cells do not exist in isolation, but rather, they exist in a dynamic environment, often surrounded by other cells. Therefore, cells need to adjust their RNA and protein levels to suit the environment. External stimuli from the cell's environment can set off a complex network of responses involving the binding and modification of proteins, the release of second messenger compounds, and the transcription of genes. These networks are referred to as cell signalling networks (for an example of a complex cell signalling network, see Oda and Kitano [2006]).

Another active area of research (and the one in which this body of research is primarily focused) has been on the mechanisms by which regulatory processes occurring in the cell control the levels of RNA and protein in the cell (that is, the expression of genes).

There are several ways in which the levels of each specific protein are regulated by cells. Over 70% of the variation in protein levels is explained by variation in mRNA levels [Lu et al., 2007]. There are, in turn, a number of mechanisms which regulate mRNA levels in the cell. One well studied mechanism is the regulation of transcription (for example, through control of the amount and activity of transcription factors). However, various kinds of non-coding RNA (ncRNA) such as microRNA (miRNA) are also believed to play a major role [Mattick and Makunin, 2006].

The transcription of RNA requires a number of general transcription factors, including the enzyme RNA polymerase. While these general transcription factors do play some role in differentially regulating genes [Veenstra and Wolffe, 2001], they can generally be treated as being an invariant, basal part of gene regulation. In addition to the general transcription factors, specific transcription

factors, which are generally proteins, are also required to initiate transcription. Specific transcription factors have highly specific affinities to particular regions of the DNA (called promoter regions). Because there are a great number of distinct specific transcription factors, and different combinations are present in any one cell at a particular point in time, the promoter sequences close to a gene on the DNA strand will affect the conditions under which the gene is transcribed.

Transcription regulation can involve a set of nearby promoter sites, known as a composite element. In order to activate transcription, not only must transcription factors be able to bind to the promoters in the composite element, but they must also interact with the other bound transcription factors in an appropriate way, therefore allowing for more specific regulation [Kel et al., 1995].

Transcriptional regulation is mediated primarily by proteins, but proteins are produced from mRNA, which is in turn transcribed from genes. This means that the proteins involved in transcriptional regulation are themselves controlled in part by transcriptional regulation. As a result, one can represent gene regulation as a gene regulatory network (GRN), which consists of genes, with relationships between genes indicating how the gene products from one gene interact with DNA or other gene products in the cell to control the expression of a gene. A single relationship between interactions in this network may result from any number of different types of interactions between gene products.

2.3 The structural nature of gene regulatory networks

There are a number of processes which ultimately control the levels of each protein present in a cell, and therefore define the metabolic activities occurring in the cell. Examples of such processes include:

- epigenetic modifications to the DNA and the histone proteins associated with it.
- the rate at which transcription is initiated and transcripts are elongated.

- the rate at which transcripts are degraded.
- the rate of transport of transcripts around the cell.
- the rate and type of post-transcriptional modifications to mRNA.
- the rate at which translation is initiated and polypeptides are elongated.
- the availability of monomers for mRNA and protein bio-synthesis.
- the rate and type of post-translational modifications to proteins.
- the rate of protein degradation.

There is evidence that all of these types of regulation occur in biology. However, most research focuses on a few key types of gene regulation which differ the most from gene to gene.

Transcription factors are genes which bind to DNA and allow RNA polymerase to initiate transcription. They are said to be *cis*-acting, meaning that they affect genes on the same chromosome to where they bind. In yeast, most transcription factors bind between 50 and 600 base pairs upstream of the transcription start site for the gene [Harbison et al., 2004]. They also generally have different sequence specificities, and so different transcription factors regulate different genes.

There are several databases of transcription factors and their binding sites available, such as TRANSFAC [Matys et al., 2006] and JASPAR [Bryne et al., 2007]. These databases are curated from biological literature. This improves their accuracy, but means they do not necessarily have a good coverage of transcriptional regulation. Harbison et al. [2004] experimentally determined the sequence specificity of all yeast transcription factors, and used this to predict the likely binding sites, thereby providing a more complete picture of transcription factor binding in yeast.

Transcription factors are often affected by other transcription factors bound nearby. Two closely situated binding sites which produce a particular expression pattern are referred to as composite regulatory elements [Miner and Yamamoto,

1991]. TRANSCompel, which is also described in Matys et al. [2006], contains a database of these composite elements.

Another type of gene regulation is referred to as RNA silencing. RNA silencing is the generic term for a number of related pathways, which refers to the reduction of expression by double-stranded or hairpin RNA. These interfering RNA molecules are referred to as small interfering RNA (siRNA) when they originate from outside of the cell, and are believed to be a defence against viruses. However, RNAi also occurs due to intrinsically produced RNA. MicroRNAs (miRNA) are an example of non-coding RNA (ncRNA), RNA which are transcribed from the genome, but do not code for proteins. MicroRNAs are single stranded RNA molecules which hybridise with mRNA, forming a stretch of double stranded RNA [Lau et al., 2001]. There are two mechanisms by which this reduces the level of protein produced from the mRNA; the predominant mechanism is through reducing mRNA levels [Guo et al., 2010], and a secondary mechanism is through translational repression. The reduction in mRNA levels is believed to be because the miRNA promotes mRNA destabilisation by removing the poly-(A) tail from the 3' end and the 7-methylguanosine cap from the 5' end, causing the mRNA to be digested [Eulalio et al., 2009].

Gene regulatory networks are complex systems which result from the numerous RNA and protein molecules which can affect the expression of other genes. Some of the proteins involved are basal in the sense that they do not vary from gene to gene, while others, such as certain DNA binding transcription factors, have different effects for different genes.

There are a large number of chemical species affecting gene regulatory networks besides the DNA and mRNAs, such as transcription factors. These proteins often bind to other proteins or metabolites, and so the gene-specific rate of transcription depends on substances from a wide range of cellular activities. As many of these substances are produced in the cell through transcription, translation, and protein-mediated biosynthesis, transcript concentrations are related to each other. GRNs generally collapse all of these complex relationships involving many derived species down to the relationships between different transcript abundances, as shown in Figure 2.1.

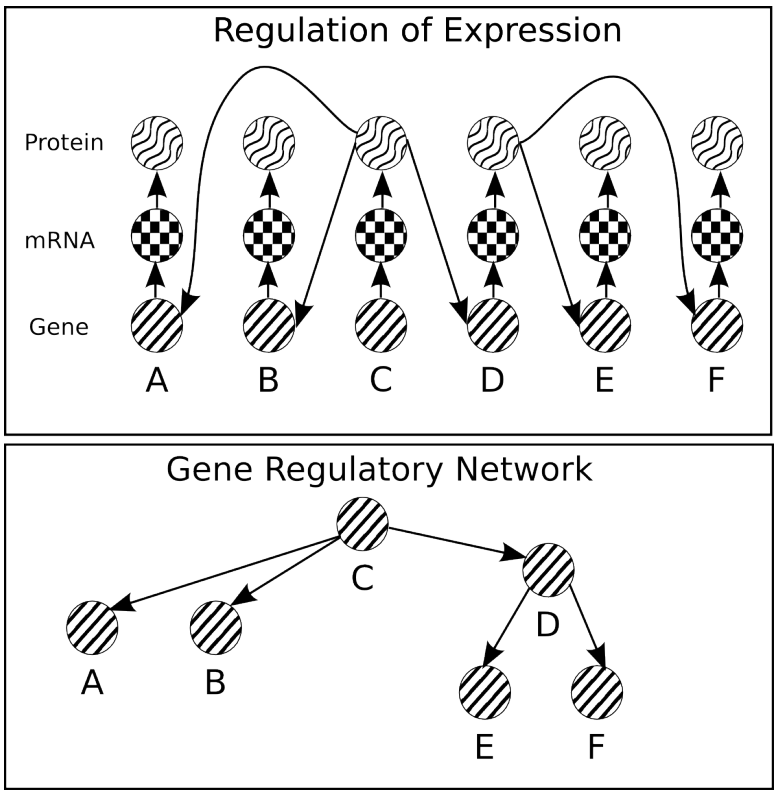

Figure 2.1: A graphical representation of how networks of genes, mRNA and proteins are collapsed to form a Gene Regulatory Network, describing relationships between genes.

2.4 Dynamic and steady-state models of GRNs

There are a wide range of different ways of modelling gene regulatory networks. A very good review of GRN modelling is available [Schlitt and Brazma, 2007], and provides a summary of numerous different modelling techniques that have been employed.

GRN models can be classified as either dynamic or steady-state models. Dynamic models describe the change in expression over time, while steady-state models work on the assumption that eventually, the concentrations of species will reach an equilibrium point, and therefore, try to describe what the cells, under a particular condition, will do in that state. In practice, a number of network motifs common in GRNs can lead to oscillatory behaviour, and these

can be quite complex as the cell goes through cyclic processes such as the cell cycle. Nevertheless, the steady-state assumption can still be used as an approximation.

One way to model gene regulatory networks is to build a kinetic model. A kinetic model describes how the concentrations of transcripts (and often other species as well) vary over time. This type of modelling is a simple extension of the modelling of individual genetic elements to a genomic scale.

2.5 Representations of gene networks

A basic aspect of gene regulatory networks is their topology. A network topology can be represented as a graph. This representation can be thought of as a set of nodes (representing genes), and a set of edges connecting pairs of nodes, indicating that the genes are somehow linked.

The literature network of genes is an example of such a topology. Jenssen et al. [2001] used MEDLINE, a database of abstracts from biomedical literature, to identify human gene names which appear together in the titles or abstracts of the same journal article. There are a number of complex reasons why two gene names might appear together in a paper, but in at least some cases, it will be because the paper has identified that one gene is involved in the regulation of the other.

Another example of such a topological graph representation can be created using synthetic lethality experiments. These experiments aim to identify pairs of genes such that if the cell is prevented from expressing one or the other of the genes, the cell is viable, but if it is prevented from expressing both genes, the cell is not viable. These types of experiments can be carried out in yeast by creating, for every gene in the genome, a strain mutant (non-functional) in that gene, and then mating those strains with a strain mutant in a different gene (using antibiotic resistance reporter genes to select out strains which didn't mate). This procedure is referred to as synthetic genetic array (SGA) analysis [Tong et al., 2001]. More recently, RNA interference has been used [Kamath

and Ahringer, 2003] to knock down the expression of genes to perform large scale synthetic lethal experiments in metazoans. The result of synthetic lethal experiments is a graph with edges between genes which are synthetically lethal. This type of graph does not directly represent genes and their regulators, but instead indicates that the two genes may be part of a common pathway, and therefore have a common regulator.

More specific topologies can be created by extending from a graph representation to a digraph representation, in which ordered pairs of nodes are connected by arcs (that is, arcs have a direction, instead of treating the two connected nodes as being equivalent with respect to an edge). It is interesting to compare the interpretation of these topologies to causal models (Definition 1 of Pearl and Verma [1991]). Under that definition, causal models are always directed acyclic graphs, while the above digraph representation allows cycles to be present. The presence of network motifs such as feed-forward loops in gene networks [Shen-Orr et al., 2002] provides an example of such a deviation. These motifs are contrasted to acyclic GRNs in Figure 2.2.

A good example of a topological digraph of a gene network is contained in the YEASTRACT database [Teixeira et al., 2006a]. The database describes regulatory associations between transcription factors and their target genes. The regulatory associations are curated from published experimental evidence. Because each transcription factor is a gene product, this can be thought of as a digraph, with an arc for each regulatory association, between the gene corresponding to the transcription factor and the target gene. The database consists of transcription factors collected from a variety of sources.

2.6 Predicting interactions *in silico* from structure and sequence

Any type of digraph derived exclusively from experimental evidence is likely to give an incomplete picture of the actual topology of gene regulatory relationships in a cell. Furthermore, there is no guarantee that regulatory relationships are

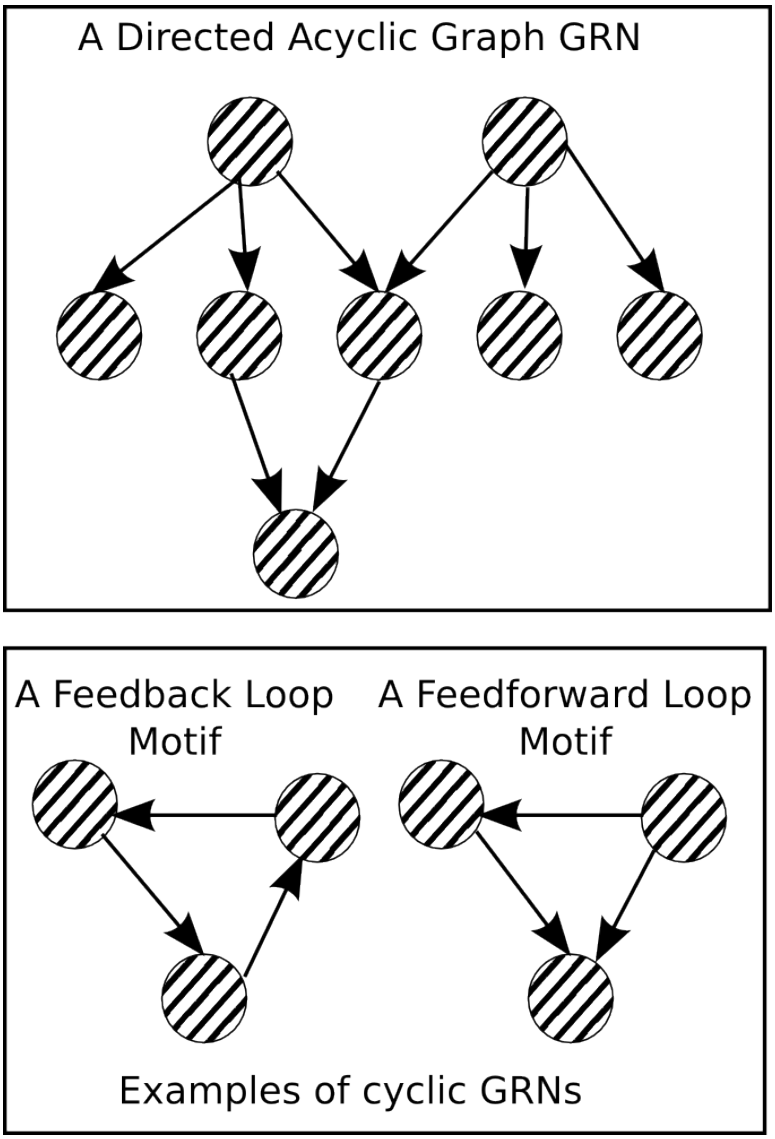

Figure 2.2: Examples of acyclic and cyclic gene regulatory networks. Feed-forward and feed-back loops, which require that cycles be present, have been hypothesised to play an important role in biological systems.

conserved between species. It is therefore beneficial to be able to determine regulatory relationships by genomic sequence analysis. In order to do this, it becomes necessary to computationally identify the transcription-factor specific promoter motifs. Ideally, it would be possible to start from the protein sequence or structure, and from that work out the DNA sequences it is likely to bind to. Morozov et al. [2005] created an interaction energy function for the binding of DNA and protein. This function requires, as an input, the structure of a homologous protein with DNA bound. Under the model, DNA binding interactions are nearly additive, which means that position-specific weight matrices for the base pairs at each position can be computed. Liu and Bader [2007] has extended the method to an all-atom simulation based approach with explicit solvent molecules. However, these *ab initio* methods are very computationally expensive, with the Liu and Bader [2007] method taking 400 CPU days on a 3.0 GHz Intel processor, for a single transcription factor.

However, models like the above have demonstrated that there is very little dependence between the nucleotide bound at one site and the affinity for nucleotides at another site (or in other words, the protein-DNA interaction energies are nearly additive). By assuming that the protein-DNA interaction energies are additive, less complex approaches become possible. These approaches do not consider the protein structure, but instead, identify potential binding specificities from DNA sequences. This problem is referred to as motif finding; the aim is to identify, from a set of sequences, any patterns, called motifs, which occur more commonly than would be expected by chance. This problem is closely related to multiple local alignment – the aim being to align a series of sequences with each other along the common points. However, [Just, 2001] and [Elias, 2006] have shown that exact multiple sequence alignments under a variety of common scoring systems is NP-hard, meaning that unless $P = NP$, there is no polynomial time algorithm which can find the most optimal multiple local alignment. For this reason, practical motif finding algorithms all place restrictions on the types of motifs they identify. Because these restrictions may not be biologically realistic, there are a large number of different motif finding programs with different restrictions on the types of motifs they can find.

Types of algorithms

There is a strong relationship between local sequence alignment and motif finding. The Smith-Waterman algorithm [Smith and Waterman, 1981] finds the globally optimal local alignment, and was presented as a method for identifying common molecular subsequences. However, the time and space required by the algorithm is polynomial in the sequence lengths, and exponential in the number of sequences to be aligned, and so it would not be feasible to use this for genome scale studies.

Practical motif finding algorithms can roughly be divided into three key types: those based on the expectation maximisation (EM) algorithm, those based on Gibbs Sampling, and the enumerative methods. Some algorithms combine aspects of more than one of these types in phases.

Methods based on Expectation Maximisation (EM)

Lawrence and Reilly [1990] proposed one of the earliest heuristic algorithms for motif finding. The algorithm is derived by framing the motif finding problem as a missing information problem; the missing information is the position at which the motif starts in each of the different sequences assumed to contain the motif (the motif model assumes the motif is of a fixed length, and has a certain probability distribution at each position). The EM algorithm then works to deduce this information by alternating between E and M steps. The E step takes as input frequency estimates for different bases outside of the motif, and at each position from the start of the motif (the model parameters), and uses these to compute, for each sequence, the posterior probability that the motif starts at each position. These posterior probabilities are then used to compute the expected numbers of each base at each position in the motif (across all sequences). The M step then uses the expected number of bases at each position to compute the frequency estimates for the next step. The algorithm stops when the estimates have converged. Lawrence and Reilly [1990] also discusses the use of different models, including gapped models, and proposes that progressively more restrictive models be tested until the decrease in predictive ability is

greater than would be expected by chance.

This algorithm was improved upon in MEME [Bailey and Elkan, 1995]. In MEME, there is no assumption that the motif occurs exactly once in each input sequence. This distinction is achieved because MEME does not normalise the offset probabilities for each sequence to sum to one, but rather, normalises the sum of all probabilities across all sequences to sum to a user-supplied estimate of the number of motif occurrences, constrained so no probability exceeds one. If a particular sequence has no occurrence at any position, it is possible for the assigned probabilities to be consistently low, rather than having to sum to one, and likewise, if there are two occurrences of the motif, they can both have high probabilities. However, this creates the problem that motifs longer than the word length would result in multiple overlapping results. MEME avoids this by constraining the sum of probabilities for adjacently positioned motif occurrences. Another innovation in MEME is that MEME starts the EM algorithm from the sequence derived frequencies rather than random inputs. The best starting point is selected by running one EM iteration, and computing the log-likelihoods after this iteration. Furthermore, MEME can report more than one different motif; it does this by making subsequent passes after finding the first motif, but down-weighting the probabilities of motifs starting at the positions involved in the first hit.

Gibbs Sampling Methods

It is also common to use Gibbs sampling to align motifs. Like the EM based methods, Gibbs sampling also treats motif finding as a missing information problem, where the missing information is the positions of the motifs, and the model, which consists of background base frequencies and base frequencies at each position of the motif, is sampled. Unlike the EM algorithm, Gibbs sampling is a stochastic approach in which motif models are randomly sampled. Lawrence et al. [1993] introduced Gibbs sampling for multiple alignment. Their algorithm starts off with a random set of positions, and samples the base frequency matrix given the starting positions, by computing the frequency matrix using the motif instances at all but one randomly sampled starting position. Then,

the algorithm, samples from the distribution of sets of positions given the model, by weighting each position in the sequence by how well it fits the model. This algorithm repeats iteratively, sampling from the distribution of models. After sufficient iterations, the optimal sampled motif model is selected using a log-likelihood criterion. One problem which was identified by the authors was that the Gibbs sampling algorithm could get stuck on a local optimum because once the frequency matrix is 'phase-shifted', that is, corresponds to a position at some small offset to the true motif start, then the algorithm is unlikely to ever sample the actual motif, because the frequency matrix for a shifted motif does not result in a high probability for the true motif positions. To avoid this problem, the authors added in an additional, periodically run step which recomputes the frequency matrix for phase-shifted starting positions, and compares them using the log-likelihood criterion. In addition, because real motif widths are not known, the algorithm also tries different widths. Because a direct log-likelihood criterion will always favour longer sequences, the authors found an empirical criterion to avoid this, and used it to find the optimal motif of any length.

Neuwald et al. [1995] created the Gibbs Motif Sampler (GMS), which (in a similar vein to how MEME improved the original EM approach), removes the assumption that the actual number of motif occurrences per sequence is known. It achieves this by sampling from the posterior distribution that a particular motif occurs at a particular site, as opposed to sampling the site locations conditional on the motif model.

Liu et al. [1995] provides an in depth theoretical discussion of motif alignments. The authors developed a Bayesian model of multiple sequence alignment, and in addition, discuss the background to removing the one occurrence per sequence assumption. The authors then present an extension which allows for fragmentation; that is, for there to be gaps in a particular motif match.

Roth et al. [1998] further extended the Gibbs sampling approach in a program called AlignACE. AlignACE extends Neuwald et al. [1995] by adding iterative sequence masking, allowing it to find multiple different types of motif automatically. In addition, AlignACE is specifically targeted at nucleotide sequences

(unlike GMS, which was primarily designed for use on proteins) and so supports features relevant to nucleotide sequences, such as searching on both strands. In Roth et al. [1998], AlignACE was combined with expression microarray data to identify genes or open reading frames (ORFs) which contained a particular potential motif and also showed a large fold-change under a particular treatment.

Liu et al. [2001] introduced BioProspector, another Gibbs sampling approach, based off the algorithm in Lawrence et al. [1993]. Unlike the above Gibbs sampling and EM algorithms, BioProspector uses a Markovian background model, in which the probability of a base appearing in the background sequence is dependent on the one, two, or three bases preceding it in the sequence. Like GMS, BioProspector allows for different numbers of occurrences of the motif in the same sequence. However, unlike GMS, BioProspector achieves this by sampling sets of alignments conditional on a motif model. To find the alignments to add to the set, BioProspector requires an upper and a lower log-likelihood threshold. Alignments above the upper threshold are unconditionally included in the set of alignments, while those below the lower threshold are unconditionally excluded. A single alignment between the thresholds is sampled (weighted by the log-likelihood measuring how well it fits the model). The lower threshold is gradually raised as the algorithm proceeds. In addition, BioProspector allows for motifs to be modelled in two parts with a fixed width gap between the parts.

Thompson et al. [2003] introduced a number of further enhancements to motif sampling. The background model of the previously mentioned methods has been either homogeneous or low order Markovian. Thompson et al. [2003] instead pre-processes the sequences using a Bayesian segmentation algorithm, to identify segments of the sequence which are, say, GC rich, thereby creating a position-specific background model. In addition, the method, known as the Gibbs Recursive Sampler (GRS), searches for multiple different types of motifs by including information about multiple different motifs across the sequences. The method also allows for fragmentation as in Liu et al. [1995]. In addition, it includes models specifically for palindromic sequences (which are constrained to only allow fragmentation in the central positions), and reduced alphabet models for transcription factor binding sites that don't distinguish AT from TA,

or GC from CG.

Other non-enumerative methods

There have also been some non-enumerative methods which are not directly based on either EM or Gibbs sampling. Hertz and Stormo [1999] introduced a greedy hill-climbing algorithm in which sequences are progressively added to the alignment. This is done based on an efficient numerical approximation for computing the p-value from an information content score.

Enumerative methods

In addition to the statistical methods based on expectation maximisation or Gibbs sampling, another major approach to the problem has been to enumerate all possible motifs, subject to a given constraint. The advantage of enumerative methods is that because all motifs are tested, the algorithm is guaranteed to find the globally optimal motif (as opposed to a locally optimal motif). However, for this approach to be feasible, there needs to be some structural limitations placed on the forms of motifs searched for, because the number of possible motifs increases exponentially with motif length.

Galas et al. [1985] introduced an algorithm for sequences which are imperfectly aligned in columns (for example, relative to the start codon). The algorithm requires the width of the window to search in (the maximum amount by which sequences can be misaligned by) as well as a fixed length of motifs, and a neighbourhood specification, which is the number of bases by which two sequences can differ to still be considered occurrences of the same motif. The algorithm enumerates all possible ungapped motifs within the window. As the time of the algorithm grows exponentially with the length of the sequences and the neighbourhood of sequences to search in, like all purely enumerative algorithms, it is limited to small sequence lengths.

van Helden et al. [1998] described a simple oligonucleotide frequency search algorithm. In the algorithm, every possible oligonucleotide of a fixed length

(the authors repeated the method with 4 through to 9 nucleotides) was searched for, and the sequences that showed a statistically significant over-representation were identified and reported. This algorithm was extended in van Helden et al. [2000] to allow for dyad analysis, where the sequence consists of a short word, followed by a spacer 0 to 16 base pairs wide each of which matches any base, followed by another short word.

Brazma et al. [1998] developed another enumerative method. The model of a match is a regular expression of the form $A_1A_2 \ldots A_p$, where each A_i is the set of bases which are permissible under the pattern at the given position. The algorithm does not set an upper limit on the length of patterns which can be found, but does set a lower limit, and a limit on the number of permissible wildcard matches, in order to limit the complexity. Pattern occurrences in the sequence are stored in a modified suffix trie, but as part of the algorithm, patterns with identical occurrence lists (for example, patterns like AGCTACTG and GCTACTGA, if both of them only match on the same occurrence of AGCTACTGA) are identified to avoid wasted computation time.

Another enumerative algorithm for identifying motifs while allowing some mismatches was described in Pavesi et al. [2001]. The algorithm takes two parameters: q, which is the minimum number of occurrences of the pattern which are required, and e, the maximum number of base mismatches which are allowed. The algorithm starts by constructing a trie containing all 4^e possible subsequences of length e. The algorithm then relies on the following to prune the search space: if there are not q or more matches found for a subsequence, then there cannot be q or more matches for any sequence starting with that subsequence. Matches do not need to be exact, because there can be up to e mismatches, and so all subsequences in the initial trie match everything. The algorithm then iteratively increases the length of the subsequences by appending each possible base to the end of each branch in the trie, and determining how many matches there are in the input sequences. If there are more than q matches, then the branch is added. This process repeats until the desired motif length is reached. After the iterations have been completed, the next step is to sort the output. This is done based on one of several different significance

scores presented in the paper.

One of the more popular enumerative methods is YMF, the yeast motif finder [Sinha and Tompa, 2002] [Sinha and Tompa, 2003]. YMF uses a similar process to that employed by van Helden et al. [2000], except it uses in a third order Markov chain background model to determine the significance of the motif. YMF also allows a limited number of degenerate characters which match more than one base to appear in motifs.

Hybrid approaches

All of the methods mentioned so far either make heuristic assumptions right from the beginning when looking for overrepresented motifs, or are completely enumerative and therefore limited in the number of patterns they can feasibly identify. The initial enumerative step allows good seed motifs to be found, and these are then expanded upon to attain significance and refine the profile. Chakravarty et al. [2007] created a program called SPACER (separated pattern-based algorithm for cis-element recognition) using such an approach. SPACER initially searches for elements using the algorithm from BEAM [Carlson et al., 2006] (which is, in many respects, similar to Pavesi et al. [2001] except it uses a Markov chain background model). This identifies a pattern A-S_N-B, where S_N is a spacer region, and A and B are short sequences of bases. However, a heuristic search then modifies and extends the ends from the global optimum (allowing for degenerate bases, for example), and finally, the degenerate spacer region is analysed to identify weak base preferences.

Phylogenetic fingerprinting

Many of the newer motif search methods make use of aligned genomes from two or more species, in order to determine which bases are conserved. The most basic approach to this problem is called phylogenetic footprinting, and can be used to identify both coding and non-coding sites that are conserved across species [Gumucio et al., 1992]. Principled algorithms for phylogenetic

footprinting were introduced somewhat later, in Blanchette et al. [2002].

Finding motifs from position weight matrices

Another slightly different, and somewhat easier problem is, given existing position-by-position frequencies for each base in hand-curated motif occurrences, find any occurrences of this motif in the genome. The MATCH [Kel et al., 2003] algorithm achieves this type of search using an *ad hoc* algorithm which weights motif occurrences in a given position higher for bases with a higher information content (that is, for less degenerate bases).

Lähdesmäki et al. [2008] developed a more principled algorithm for the same problem solved by the MATCH algorithm, based on Bayesian statistics. This model computes a posterior probability that a group of transcription factors are present in a given promoter region, taking into account uncertainty in both the background model probabilities and the foreground model counts. As a Bayesian approach, the prior probability can be adjusted to take into account other evidence (for example, experimental evidence of binding). There is no closed form for the posterior probability, but Markov Chain Monte Carlo (MCMC) techniques can be used to find promoter regions.

In the course of this study, an alternative Bayesian approach, called BaSeTraM (Bayesian Search for Transcriptional Motifs) [Miller et al., 2010b] was developed; the paper is reproduced as Chapter 3. The method uses a different derivation for foreground model uncertainty, and assumes no uncertainty in the first-order Markovian background model. This allows a closed form for the posterior probability in terms of the sequence being searched, and makes it computationally feasible to search the entire human genome against the entire TRANSFAC motif database in only weeks of single-core commodity processor time. A benefit of using a Bayesian approach over MATCH is that the trade off between false positives and false negatives can be adjusted by changing the posterior probability cut-off.

While TRANSFAC matrices only describe raw counts at each position in the motif, more accurate algorithms for searching for existing motifs have been

developed using more sophisticated motif models. Public availability of large databases of such motifs is currently limited. Marinescu et al. [2005] introduced MAPPER, which models motifs as Hidden Markov Models, achieving a better selectivity and sensitivity than with MATCH.

2.7 Reverse engineering gene regulatory networks

The problem of building steady state models of gene regulatory networks can be decomposed into building a topological relationship between regulator and regulated genes, and determining the form of the functions from the quantitative expression levels of the regulator genes to the expression levels of the regulated genes.

When building topologies and representations of quantitative relationships, it becomes necessary to trade off improved fit to experimental data with increased complexity of the models. The objective of most approaches is, more or less, to minimise the actual risk, that is, the expected loss, which measures the deviation of the prediction from the truth, over the distribution of possible true values. It is possible to estimate the actual risk using empirical data, but minimising this empirical risk (or similarly, choosing the model which maximises the likelihood of predicting a particular set of data conditional on the model) will favour increasingly more complex models, even when these models do not generalise to new data well. This problem is known as overfitting in machine learning literature. The problem can also be thought of in terms of Occam's Razor; that all other things being equal, the simplest possible solution is the best. One of the keys to developing good network reconstruction methods is therefore to find a principled way to trade off increased complexity against better empirical fit to data.

In addition to the above problem, another key concern is the time complexity of algorithms. Even for the most simple representation of gene networks, using digraphs, which can only capture two-way interactions between genes, there are $n(n-1)$ possible edges between n genes, and because each of these edges can be present or absent, there are 2^{n^2-n} possible graphs. It is therefore important

to choose algorithms which can scale to all the genes in the genome (high n).

Gene clustering

As early as 1998, hierarchical structures of genes were being created from expression microarray data [Wen et al., 1998, Eisen et al., 1998] (for a review of other early microarray analysis methods, see Quackenbush [2001]). However, the focus of these analyses was to cluster genes with similar expression patterns together, and not to determine the actual structure of a gene regulatory network. As such, the methods were used for exploratory data analysis.

Tavazoie et al. [1999] also made use of hierarchical clustering methods (in this case using the k-means algorithm), but then used the AlignACE motif finding algorithm, as discussed in section 2.6, to find explanations for why genes cluster together.

Maximum parsimony

Wagner [2001] introduced one of the first methods aimed at reconstructing the causal structure of a gene regulatory network. This algorithm requires, as input, microarray data for a perturbation of every gene to be considered as part of the gene regulatory network. From this information, an accessiblity list (that is, the set of genes which change as a result of a perturbation to a gene) is constructed. Construction of the digraph representation of the regulatory network then occurs from the accessiblity list. The method aims to construct the most parsimonious digraph representation, and from the data used as input, this means there cannot be any shortcut edges, that is, there can only be one directed path between any pair of genes.

In another key network reconstruction study, Ideker et al. [2001b] perturbed 20 genes related to galactose utilisation in *Saccharomyces cerevisiae*, chosen based on an existing model, and measured the transcript levels using expression microarrays. The authors then carried out isotope coded affinity tag tandem mass spectrometry experiments to compare protein levels in wildtype strains

in the presence and absence of galactose. The authors did not propose a completely automated algorithm for the analysis of their data, but rather, generated hypotheses by the manual inspection of results in the data.

Combining experimentation with model building

Davidson et al. [2002] took another approach to gene regulatory network reconstruction, which they applied to sea urchin development. Rather than taking the most simple model consistent with perturbative data, the authors used potential regulations identified from quantitative real-time PCR data to motivate further experimentation. One type of further information collected was evidence of cis-regulatory elements such as transcription factor binding sites. This allowed direct and indirect regulation to be distinguished. They also made use of double knockout experiments to see if one knockout can interact with the effects of another knockout.

Bayesian networks

Friedman et al. [2000] introduced the use of Bayesian networks for expression data analysis, using the Spellman et al. [1998] time series data. Expression values were represented either as samples from the multinomial distribution (with relationships modelled as probabilistic dependencies) with expression values discretised into upregulated, downregulated and no change bins, or from the multivariate normal distribution. The method did not find a single Bayesian network, but instead used a Markov Chain Monte Carlo (MCMC) algorithm to compute properties of the posterior distribution of Bayesian networks. This can be used to compute properties such as the probability of an edge between two different nodes in the gene network.

Pe'er et al. [2001] applied the framework from Friedman et al. [2000] in order to pick out subnetworks from the model, that is, networks with edges which have strong experimental support. These networks are made up of edges which are present in a large proportion of the bootstrapped networks (which are sampled

as part of the MCMC process from the posterior distribution of Bayesian network representations).

Imoto et al. [2002] also extended the ideas from Friedman et al. [2000], by creating a Bayesian network model which allowed for non-parametric relationships between genes, by representing the expression response to the gene parents as a non-parametric regression (for example, based off B-splines) with a Gaussian noise component. The method makes the assumption that the microarray readings for a gene are a linear sum of functions of that gene's expression in terms of a single parent at a time, plus the noise. This means that three-way interactions between genes cannot be represented in the model.

Unlike Friedman et al. [2000], which aimed to find properties of the posterior distribution of models, Imoto et al. [2002] aimed to find the *maximum a posteriori* (MAP) Bayesian network. Finding the globally optimal Bayesian network has been shown to be NP-complete [Chickering et al., 2004]. Instead, the paper makes use of a greedy heuristic algorithm, with added model permutation steps designed to allow escape from some local minima.

The BNRC score, which is used to select the optimal Bayesian network in Imoto et al. [2002], is an approximation of the posterior probability of a graph, given the prior probability of the graph (which takes into account the complexity of the graph) and parameters, and the likelihood of the data given the B-splines.

Imoto et al. [2004] adapted the methodology of Imoto et al. [2002] to steady state microarray data. The problem of trading off model complexity with empirical fit is achieved through the use of Bayesian priors (which is justified through the explanation that models with more edges are, *a priori*, less likely), giving more complex models a progressively lower prior probability. The authors also discuss the possibility of using priors to take into account other types of biological information, such as known protein-DNA interactions.

Tamada et al. [2003] developed a different Bayesian networks based approach for incorporating biological data. The paper defines an iterative EM type algorithm (although the paper does not refer to the algorithm as an EM algorithm). Initially, no prior information is assumed, and the *maximum a*

posteriori network is computed using the microarray data. Next, the initial estimate of the network structure is used to guide a genome search for motifs between gene parents and children. The results from this search are fed back into the first step, and the algorithm repeats until convergence is reached.

Mutual Information Relevance Networks

Butte and Kohane [2000] introduced the concept of mutual information based relevance networks, that is, networks of genes which tend to have similar expression levels (or in other words, knowledge of the expression level of one gene provides a high level of knowledge about the expression level of another gene). Genes which do not interact, either directly or indirectly (including through connection with a common parent) have a true mutual information of zero (although estimations of the mutual information may be non-zero due to prediction error).

ARACNE [Margolin et al., 2006] is another steady-state model reconstruction method. ARACNE does not directly construct the quantitative relationships between genes, but rather, extends Butte and Kohane [2000] by computing mutual information for a topology by transforming all quantitative expression data to values between 0 and 1 based on their rank order, and estimates the entropy based on sample spacing (utilising the standard normal distribution). Like the Bayesian network approaches discussed above, ARACNE also only considers two-way interactions between genes, and not higher interactions. The decision to only allow up to two-way interactions was deliberately made in the design of ARACNE. This was necessary because the mutual information estimator relies on a kernel density estimation of the joint distributions.

ARACNE considers the mutual information for each pair of genes, and uses this value to decide which edges to add into the network. It then completes a second step, which involves going through each set of three genes where there is an edge between all three pairs, and removing extraneous edges using the data processing inequality (in other words, when there are edges between every pair from three nodes, then the edge with the lowest mutual information is treated

as an indirect interaction, and therefore deleted).

The Context Likelihood of Relatedness (CLR) algorithm [Faith et al., 2007] extends the concept of relevance networks introduced in Butte and Kohane [2000]. Like ARACNE, it uses mutual information to build gene networks. However, the algorithm evaluates the significance of each transcription factor / target gene pair in terms of the distribution of the mutual information values for all possible pairs involving either the same transcription factor or the same target gene. The authors argue that this is sufficient to remove indirect interactions, and so no further attempts (such as using the data processing inequality, as in ARACNE) are made to remove such interactions.

2.8 Synthetic gene networks

One important consideration for gene network reconstruction tools is the need to be able to test how well the algorithm can reconstruct functional relationships in the data. Because the ground truth is not known for any biological data set, and is rather what we aim to reconstruct using these methods, it is desirable to be able to create simulated results for gene networks. As part of this process, it is important that the gene network includes a level of complexity comparable to that found in biological systems.

Mendes et al. [2003] generated synthetic gene regulatory networks, which can be used to validate a range of different algorithms which operate on gene regulatory networks. One key aspect of this algorithm is the production of initial gene networks with different types of properties. The simplest option, from a computational standpoint, is an Erdős-Rényi network model (in which the presence or absence of every possible edge is decided at random, independently of other edges, and with fixed probability). Another possible network model is the small-world model introduced in Watts and Strogatz [1998]. In this model, the expected distance between any two genes is lower than the expected distance in an Erdős-Rényi model. This type of model is created by starting with a lattice model in which genes are connected to 'neighbouring' genes, and then adding short-cut interactions. In addition, the authors also provided an

option for generating scale free networks [Barabasi and Albert, 1999], using the algorithm of Albert and Barabasi [2000].

The synthetic GRN approach then takes the generated network topologies, and uses this to add in a rate law of the form:

$$\frac{dG_i}{dt} = V_i \prod_j \frac{Ki_j^{n_j}}{I_j^{n_j} + Ki_j^{n_j}} \times \prod_k \frac{A_k^{n_k}}{A_k^{n_k} + Ka_k^{n_k}} - d_i(G_i) \tag{2.1}$$

In this equation, G_i represents the concentrations of the ith gene, while I_j and A_k, respectively, represent the concentrations of the activators and inhibitors of the gene concerned. $d_i(G_i)$ represents the rate of mRNA degradation, and V_i and Ki_j and Ka_j are constants. The constants for the rate law are, in the normal instance, held constant across all genes to avoid the question of which values should be used, but can be manually edited after the model has been generated.

On top of any measurements of gene levels taken, random noise is added in. This is important in the simulation of gene networks, because expression microarray experiments are vulnerable to a certain level of random noise, and good techniques must be able to deal with this. The Mendes et al. [2003] simulator can add either Gaussian or γ-distributed noise.

Using these synthetic gene regulatory network models, a number of different *in silico* experiments can be performed. One type of experimental simulation is of a knockout experiment. To simulate this, the constant V_i for the gene concerned is set to zero, meaning that no mRNA for that gene is produced. The simulation is run until the results converge on a steady state, and the results are taken from there.

Another type of simulated experiment is an environmental perturbation time course. To simulate this, a number of genes are chosen at random, and the rate constants for these genes have a random change in magnitude applied to them, and the simulation is run, with simulated samples taken at a small number of time points after the perturbation.

Another study, by Ramsey et al. [2005], introduced a software framework called Dizzy. This framework divides simulations up into modules, which includes

representations of the biochemical reactions making up the framework. One key enhancement that this study provided over early work is the ability to perform stochastic simulations, based on Gillespie's direct method.

2.9 Experimental Measurements of Transcription Factor Binding

Chromatin immunoprecipitation

Experimental evidence about transcriptional regulation can be obtained from chromatin immunoprecipitation (ChIP) studies. In the most basic form, chromatin immunoprecipitation allows proteins bound to DNA to be precipitated, therefore allowing DNA-protein binding to be detected. Chromatin immunoprecipitation can be combined with microarray technology (a combination known as ChIP-chip) to identify the DNA to which the protein has bound.

Harbison et al. [2004] created a fusion transcription factor protein containing a c-Myc tag, for all transcription factors in yeast (one transcription factor at a time). The transcription factor fusion protein is assumed to bind DNA at the normal sites, and is then chemically cross-linked *in vivo*. The DNA-tagged transcription factor is precipitated from the cell lysate using an antibody for the c-Myc tag. The DNA is then released from the precipitated transcription factor, copied by PCR, and identified using a genomic microarray. This study allowed Harbison et al. [2004] to identify the binding specificity of every single transcription factor in *Sacchromyces cerevisiae*. From this, the genes with a coding region starting within a threshold distance downstream of the binding site are treated, putatively, as being regulated by the binding transcription factor.

To date, there has not been a similar study to identify the binding specificity of transcription factors in any metazoan. This is because of the increased complexity of gene regulation required to code for different types of differentiated cells. According to Babu et al. [2004], there are 2604 DNA binding proteins in *Homo sapiens*, compared to 267 in *Saccharomyces cerevisiae*, and so repeating

the Harbison et al. [2004] study for human would be an order of magnitude more difficult.

2.10 Microarrays

Using DNA microarray technology [Schena et al., 1995], it is possible to get an indication of the levels of different transcripts present in a cell extract. DNA microarrays are glass slides with a large number of tiny spots of DNA strands, known as the probes, attached to the surface. RNA in a cell extract is reverse transcribed to complementary DNA (cDNA) and the cDNA is chemically labelled with a fluorescent dye. The dye-labelled cDNA is allowed to flow over the microarray surface and hybridise with any probes having the reverse complement sequence. Any unhybridised cDNA is washed off, and the microarray is then scanned to form an image showing the intensity of the fluorescent dye at different locations on the microarray. This image is analysed to estimate the abundance of the transcripts expected to bind to each probe.

Spotted microarrays were the first type of DNA microarray to be developed. Probes are firstly produced and put into wells on a plate. They are then printed on the glass slides using a spotting robot. This type of microarray is generally then hybridised with a mixture of cDNA from two different samples, each of which was labelled with a different dye. By comparing the intensity of the two signals, it is possible to estimate whether a particular transcript was more or less abundant in one sample than the other.

Higher density microarrays can be produced though a photolithographic process [Fodor et al., 1991]. Oligonucleotides are synthesised *in situ* by the cyclic addition of nucleotides. In each cycle, light-sensitive protecting reagents are added, and whether or not a base is added in each step is controlled by light falling on the spot. By using a series of light masks to control where light falls, the oligonucleotide sequence synthesised at a spot is controlled. *In situ* oligonucleotide arrays can contain far more probes per array (in the order of one million oligonucleotide features). This allows them to contain multiple probes for the same gene at different locations on the same microarray. By

using multiple probes, and by normalising using ubiquitous maintenance genes, it is possible to more reliably compare between arrays. For this reason, only one sample is hybridised to each array.

DNA microarrays can estimate the quantities of many different transcripts present in a cell type. However, for such data to improve our understanding of the underlying biology, it needs to be interpreted. Models of gene regulatory networks (GRNs) attempt to explain how the processes occurring in the cell produce the observed transcript abundances.

DNA microarrays contain a large amount of data about transcript levels in cells. However, the exact relationship between the expression microarray readings to the activities of the cell is not fully known.

For example, Griffin et al. [2002] performed both an expression microarray measurement of transcript abundance between two conditions, and an isotope coded affinity tag (ICAT) mass spectrometric measurement of protein abundances between the same two conditions, and found that there is a correlation between transcript and protein abundance, but it is not an entirely accurate predictor.

Expression microarray data inherently contains noise from a variety of sources, and appropriate normalisation can often reduce the impact of this noise. For this reason, there has been a great deal of research into microarray normalisation and post-processing methods.

Hegde et al. [2000] provided a description of a protocol for two-colour expression microarray data. Several steps are involved in the data analysis. Firstly, the microarray is scanned using lasers at the appropriate frequency for the two dyes hybridised to the sample. This produces two images indicating the readings for each dye at different points on the microarray. This image is then passed into software to determine spot locations. This produces both a foreground (spot) intensity, and a background intensity. Because there can be regional variation across different parts of the microarray, background intensities used are always local to the region around the spot.

In order to take into account regional variations in background intensity on the

array, background correction is performed. One of the simplest methods, as used in the Hegde et al. [2000] protocol, is to simply subtract the background intensity from the foreground intensity. However, this type of background correction does result in some problems for the analysis. One major problem is that, for low intensity signals, the background reading may happen to be greater than the foreground reading, giving a negative corrected value. Another problem is that subtractive background correction results in increased variation for low intensity samples, and means that the ratios between the two dyes are systematically underestimated [Qin and Kerr, 2004]. Edwards [2003] described a method which smooths out values where the intensity difference is less than some cut-off using an exponential model. Ritchie et al. [2007] introduced a more principled model based approach to background correction, based off the normal exponential distribution.

Because two different dyes are used to label the RNA from the different conditions on two colour microarrays, the values on each channel are not directly comparable, and so ratios between the two channels cannot be taken directly. Instead, normalisation between channels needs to be performed. This can be done by assuming that most genes on the array, across a range of different average intensities, are unlikely to have changed between the two conditions, and so performing a locally weighted normalisation such as LOESS [Jacoby, 2000] or LOWESS. Similarly, there can also be effects between microarrays and between print-tips on the same spotted microarray, which may need to be normalised out.

One colour array systems, such as the high density *in situ* arrays produced by Affymetrix, require different types of normalisation technique. Affymetrix arrays have two different types of probe, referred to as perfect match (PM) and mismatch (MM) probes, which match and only partly match sequences from the gene target, respectively. One of the methods which has been widely used is MAS 5.0, an algorithm provided by Affymetrix for the analysis of the arrays. However, these methods have been shown, using RNA spike-in experiments, to not perform as well as alternatives [Irizarry et al., 2003]. Particular problems are that the mismatch probes do not only measure non-specific binding, they also

measure, to some extent, mostly specific binding by the partially matching probe. The Robust Microarray Analysis (RMA) algorithm, also defined in Irizarry et al. [2003], makes use of a linear effects model in order to achieve much better performance at detecting spike-ins than any of the earlier methods. These methods have been implemented in a common software framework [Gautier et al., 2004].

Harr and Schlotterer [2006] carried out an analysis of different Affymetrix microarray normalisation methods by exploiting the fact that in *E. coli*, genes are organised into operons, which are co-transcribed. As a result of this, normalisation methods which are performing well should provide a high correlation between the expression level measurements for the genes on the operon. They also tested correlations between replicates. They split the process of Affymetrix data processing into four stages: correction for non-specific binding, normalisation, correction between PM and MM (positive and mismatch) probes, and the summarisation method for combining multiple probes into a single probeset expression level. Different permutations of the methods at each step were tested. The results showed that the summary method made the largest contribution to the correlation coefficients, but that the median polishing algorithm used by RMA and GCRMA [Wu et al., 2004] produces the highest correlation coefficients between replicates, while the approach of Li and Wong [2001] produces higher correlation coefficients between genes in the same operon.

2.11 Sequence based measurements of mRNA abundance

SAGE and CAGE

SAGE, or Serial Analysis of Gene Expression [Velculescu et al., 1995] measures the abundance of mRNAs as follows: mRNAs are purified by selecting for the poly(A)-tail with affinity chromatography (using poly(T) beads). The 5' end is removed with a nuclease, leaving only a short sequence of the original mRNA sequence from the 3' end, called a tag. The tags are ligated to primers and

the sequences are ligated together, amplified, and inserted into a plasmid for cloning in bacteria. The resulting DNA sequence is sequenced, and mRNAs are identified by the tags.

CAGE, or Cap Analysis of Gene Expression, is an alternative protocol which instead uses sequence from the 5' end of mRNAs as a tag, allowing identification of the transcription start sites associated with mRNAs.

MPSS

MPSS, or Massively Parallel Signature Sequencing attaches mRNA sequences to beads. The beads are arranged in a two dimensional array, and sequencing occurs on the beads, allowing the mRNA sequence attached to each bead to be sequenced in parallel.

RNA-Seq

RNA-Seq [Mortazavi et al., 2008] uses next-generation sequencing technologies to directly sequence cDNA fragments obtained from purifying, digesting, and reverse transcribing RNA from an extract. Unlike SAGE, CAGE, and MPSS, RNA-Seq does not require that cDNA sequence be assembled into plasmids and cloned in bacteria. In addition, the entire transcriptome is used, rather than only the 5' or 3' ends; the full sequence can be assembled, and the levels of different variants can be estimated. Massively parallel pyrosequencing, rather than Sanger sequencing, is used [Margulies et al., 2005].

Sequencing occurs by allowing the DNA Polymerase complex to bind to cDNA strands on an array. The arrays are exposed to mixtures containing one nucleotide at a time, along with luciferin and luciferase (which generates light when exposed to luciferin and ATP), adenylyl sulfate, and sulfate adenylyltransferase (which produces ATP from adenylyl sulfate and diphosphate). The incorporation of the nucleotide releases diphosphate, which combines with adenylyl sulfate to produce ATP, which in turn triggers the luciferase reaction, producing a flash of light. This light is detected by a fibre optic array. By cycling through

different nucleotides and detecting which nucleotide is incorporated into the newly synthesised strain, the reverse complement of the sequence is detected. Unlike Sanger sequencing, which typically sequences at most 96 sequence fragments per machine, Margulies et al. [2005] were able to sequence 1.6 million fragments (albeit of a shorter read length) in parallel on one machine, using only a small quantity of DNA.

2.12 Validating gene regulatory networks against expression microarray data

A number of methods for building gene regulatory networks from expression microarray data, or taking into account expression microarray data, have been discussed in this chapter. An alternative approach is to build gene regulatory networks based on other data sources, and then validate those networks for consistency against expression microarray data. This approach is useful because it allows the quality of gene regulatory networks (and the hypotheses used to build them) to be tested against relatively large data sets.

There are two major approaches to validation of GRNs against expression microarray data that have been adopted. The first approach is to treat GRN models as a set of rules, and test those rules against data, while the second is to convert a GRN model into a quantitative function describing gene expression, and compute an error residual between the levels predicted by the function, and the actual level.

In the first approach each gene on each microarray is classified as upregulated, downregulated, or change unknown, based on whether a statistically significant change can be detected; the model (which annotates each edge from regulator to regulated gene to indicate if the regulator inhibits or activates expression) is tested against the rules for consistency. The consistency rules are introduced as a *sign algebra* in Siegel et al. [2007]; a sign is a member of $+, -, 0, ?$. The predicted expression of a gene product is the sum, over each regulator, of the direction of up- or down-regulation of the regulator, multiplied by the direction

of regulation (- for inhibition, + for activation) of the regulator on the regulated gene. Addition is defined so that adding a + and a + or a - and a - will produce a + or - respectively, but adding a + and a - will produce a ? (representing uncertainty).

This sign algebra is utilised in Guziolowski et al. [2009a] to validate models using a simple consistency rule: if a + or - is predicted, but the actual value is a - or +, respectively, then that represents a failure of validation. The authors tested this method on E. coli using an expression microarray data set and RegulonDB, a set of interactions, and found 80% agreement. A Cytoscape plugin to simplify the use of this method is described in Guziolowski et al. [2009b].

The quantitative approach is described in Chapter 6 of this thesis. Rather than giving a definitive answer of whether data is consistent or not, it computes an error value giving the difference between a predicted value and the actual microarray value. While this makes interpreting the results more difficult, it has the benefit that more marginal variation can be taken into account, and the method is less vulnerable to mis-classification, because statistical testing occurs on the final error only, rather than for each classification of a gene as being up- or down-regulated. The method also does not require an explicit specification of whether a regulatory interaction activates or inhibits expression; instead, the general statement is that the expression of the regulated gene is a function of the regulator genes. Under this model, complex interactions are possible; for example, a regulator may switch from activating expression to inhibiting it under different conditions.

As discussed in more detail in Chapter 6, the method determines the shape of the function from regulator gene expression levels to regulated gene expression levels using a regression method, support vector regression. Some of the expression microarray data is used to train the support vector regression function, while the remainder of the data is held back and used for the validation, in which predictions are made for each regulated gene from the regulator levels, and a residual is computed.

Chapter 3

BaSeTraM - A Bayesian Search for Transcriptional Motifs

This chapter is based closely on a peer-reviewed paper that was published in PLoS One. The research described in the manuscript and the first draft of the manuscript was completed by me, while my supervisors, Cristin G Print, Poul M F Nielsen, and Edmund J Crampin provided advisory input into the research methodologies, suggested refinements to the manuscript, and were listed as coauthors on the submitted paper. Further refinements to the manuscript, as well as to the graphical presentation of the results, were suggested by the two anonymous peer reviewers.

3.1 Abstract

Identifying transcription factor (TF) binding sites (TFBSs) is an important step towards understanding transcriptional regulation. A common approach is to use gaplessly aligned, experimentally supported TFBSs for a particular TF, and algorithmically search for more occurrences of the same TFBSs.

The largest publicly available databases of TF binding specificities contain models which are represented as position weight matrices (PWM). There are other methods using more sophisticated representations, but these have more limited databases, or aren't publicly available. Therefore, this paper focuses on methods that search using one PWM per TF.

An algorithm, MATCHTM, for identifying TFBSs corresponding to a particular PWM is available, but is not based on a rigorous statistical model of TF

binding, making it difficult to interpret or adjust the parameters and output of the algorithm. Furthermore, there is no public description of the algorithm sufficient to exactly reproduce it. Another algorithm, MAST, computes a p-value for the presence of a TFBS using true probabilities of finding each base at each offset from that position.

We developed a statistical model, BaSeTraM, for the binding of TFs to TFBSs, taking into account random variation in the base present at each position within a TFBS. Treating the counts in the matrices and the sequences of sites as random variables, we combine this TFBS composition model with a background model to obtain a Bayesian classifier. We implemented our classifier in a package (SBaSeTraM).

We tested SBaSeTraM against a MATCHTM implementation by searching all probes used in an experimental *Saccharomyces cerevisiae* TF binding dataset, and comparing our predictions to the data. We found no statistically significant differences in sensitivity between the algorithms (at fixed selectivity), indicating that SBaSeTraM's performance is at least comparable to the leading currently available algorithm.

Our software is freely available at:

http://wiki.github.com/A1kmm/sbasetram/building-the-tools

3.2 Introduction

Identifying which transcription factors bind to which promoters is an important step towards understanding the transcriptional regulatory code. This identification process can be divided into two parts: determining the binding specificity of specific transcription factors, and then identifying TFBSs in a sequence using the binding specificity information.

There have been a number of papers proposing methods for one or both parts of the problem. Methods for finding transcription factors (as motifs which are statistically over-represented in sequences) can be broadly classified as those

based on phylogenetic footprinting, and those which are not. These methods have been widely compared [Tompa et al., 2005, Mahony et al., 2007] and reviewed [Das and Dai, 2007]. The software implementations associated with many of these methods often also include software to use the motifs to identify TFBSs. For example, the popular motif finding software MEME [Bailey et al., 2006] is packaged with the MAST software [Bailey and Gribskov, 1998].

The link between determining binding specificity and finding sites where the transcription factor is likely to bind is the way in which binding specificity is represented. At present, the largest databases which are generally available, such as TRANSFAC [Matys et al., 2006] and JASPAR [Bryne et al., 2007], represent binding specificity using an ungapped position weight matrix (PWM) representation. Each entry in an ungapped PWM, $\mathbf{F}$, is a weight for finding a particular base at a particular position from the start of the motif. There are several types of weights possible, but in this paper, we consider weights given as a raw count. At each aligned position i in the binding footprint, a frequency is recorded for each base b to give the matrix entry $\mathbf{F}_{ib}$. Let m denote the total number of aligned sequences. Not all TFBS sequences are aligned to both ends, and so for each i, $\sum_j F_{ij} \leq m$. Note that in algorithms such as MEME, the algorithm alternates between finding an alignment, and determining the PWM, until the algorithm meets a termination condition and the final PWM is produced.

There are more sophisticated representations for transcription factor binding specificity, such as the Hidden Markov Model (HMM) approach used by MAPPER [Bryne et al., 2007]. However, TRANSFAC and JASPAR collectively include a reasonably large number of matrices, and these are available to the public (albeit under commercial terms in the case of TRANSFAC). Other databases are either smaller in size, or as in the case of MAPPER, binding models are not available to the public (Voichita D. Marinescu, personal communication). For this reason, the focus of this paper is on methodology which uses only ungapped PWM information to search for transcription factor binding sites.

Representing transcription factor binding specificities in this form means that

no data is stored on the interaction of binding specificity between different base positions in the binding sites. This is a reasonable approximation, as molecular binding models describing the interactions between transcription factors and DNA have shown that binding energies are approximately additive between bases [Liu and Bader, 2007] (in other words, interaction of binding specificity is negligible).

Existing PWM based search methodologies, such as MATCHTM, have not been justified based on a formal statistical model. MATCHTM instead computes scores using the formula [Kel et al., 2003]

$$\sum_{i=1}^{n} \left(\sum_{b \in \{A,C,G,T\}} F_{ib} \ln 4F_{ib} \right) F_{iB_i} \tag{3.1}$$

where n is the length of the matrix, B_i is the base at position i in the sequence, and F_{ib} is the frequency of base b at position i.

3.3 Methods

Overall approach

Let **A** be a sequence of bases of length at least n (where each base can be A, G, C, or T). We aim to make a decision about whether there is an exactly aligned TFBS starting at the beginning of **A**.

We define a TFBS as a locus that is under evolutionary pressure so the sequence is one that a particular transcription factor will bind to. The sequence is used as evidence supporting the hypothesis that there is a TFBS at a particular locus. For example, the presence of a sequence exactly identical to the consensus sequence for the transcription factor is strong evidence for a TFBS. A sequence which is more distantly similar to the consensus sequence is weaker evidence for there being a TFBS. This is because there are an increasing number of possible sequences as the deviation from the consensus sequence increases, and so the null hypothesis that similarity to the consensus sequence arose by chance (as opposed to natural selection) becomes more credible.

Under this definition, a transcription factor either binds to a TFBS, or it does not; there is no attempt to model the degree of affinity, only to determine if there is evidence for an underlying process. Note that evolutionary pressure may select for a moderate TF-TFBS affinity, but against a stronger affinity. In this case, evidence for the TFBS is reduced, but may still be enough to detect the site.

We use two models of putative TFBS sequences. The foreground model describes the distribution of sequences under the alternative hypothesis that there is a TFBS at the site. The background model describes the distribution of sequences under the null hypothesis that there is no TFBS at the site.

Foreground model

Our foreground model is best introduced in terms of a matrix of hidden parameters p_{ij} which represent the probability that a true TFBS will contain base j at position i. This parameter should not be confused with $\frac{F_{ij}}{\sum_k F_{ik}}$, which is merely an estimator of p_{ij}. The true p_{ij} is unknown. For this reason, we build a statistical model of F_{ij}, so we can express the joint distribution of F_{ij} and the TFBS sequence, under the alternative hypothesis. We refer to the alternative hypothesis that this model applies as H_1.

Our foreground model requires that each base in a TFBS is independently selected in accordance with the hidden parameters. In practice, there are two ways in which new TFBSs are likely to arise. They may arise from convergent evolution, in which case the TFBS sequence is independent of all other TFBSs. Alternatively, an existing TFBS could be copied in a duplication event, creating a paralogous TFBS which is not independent of the original. Over time, however, mutations to less strongly conserved bases in the two TFBSs will reduce this dependence. For this reason, the independence assumption is reasonable except for very recently duplicated TFBSs.

If B_i is the base at position i into a TFBS, the probability of the sequence $\mathbf{B}$ is

$$P(\mathbf{B}|\mathbf{p}) = \Pi_i p_{iB_i} \tag{3.2}$$

We assume, under this same model, that $\mathbf{F}_i$ is a random variable produced by aligning n_i independent sequence samples (where $n_i = \sum_j F_{ij}$), and therefore that

$$F_{ij} \sim \mathrm{B}inomial(n_i, p_{ij}) \tag{3.3}$$

Hence,

$$P(F_{ij} = f_{ij}|p_{ij}) = \binom{n_i}{f_{ij}} p_{ij}^{f_{ij}} (1 - p_{ij})^{n_i - f_{ij}} \tag{3.4}$$

where f_{ij} is a non-negative integer representing a frequency.

Now,

$$\begin{aligned} &P(B_i = b \cap F_{ib} = f_{ib}|p_{ib}) \\ &\qquad = \binom{n_i}{f_{ib}} p_{ib}^{f_{ib}} (1 - p_{ib})^{n_i - f_{ib}} p_{ib} \end{aligned} \tag{3.5}$$

$$\begin{aligned} &P(B_i = b|F_{ib} = f_{ib}) \\ &\qquad = \frac{P(B_i = b \cap F_{ib} = f_{ib})}{P(F_{ib} = f_{ib})} \\ &\qquad = \frac{\int_0^1 \binom{n_i}{f_{ib}} p_{ib}^{f_{ib}} (1 - p_{ib})^{n_i - f_{ib}} p_{ib} dp_{ib}}{\int_0^1 \binom{n_i}{f_{ib}} p_{ib}^{f_{ib}} (1 - p_{ib})^{n_i - f_{ib}} dp_{ib}} \\ &\qquad = \frac{\beta(f_{ib} + 2, n_i - f_{ib} + 1)}{\beta(f_{ib} + 1, n_i - f_{ib} + 1)} \end{aligned} \tag{3.6}$$

where $\beta(x, y)$ is the Euler Beta function [Olver et al., 2010].

Note that we assume that $P(p_{ib}) = 1$ (*i.e.* without any samples from f_{ib}, we know nothing about p_{ib}). This is the same as the B$eta(1, 1)$ distribution, from the conjugate prior family to the Binomial distribution.

This gives us the ability to compute the probability of a given sequence under the alternative hypothesis:

$$P(\mathbf{B}|\mathbf{F}) = \Pi_{i=0}^{l} \frac{\beta(f_{iB_i} + 2, n_i - f_{iB_i} + 1)}{\beta(f_{iB_i} + 1, n_i - f_{iB_i} + 1)} \tag{3.7}$$

Background model

We used a simple first-order Markov chain model, with one parameter for each base, q_b, describing the probability that the base b occurs at a particular point

in the sequence. In addition, we introduce one parameter, $t_{b_1 b_2}$ for each pair of bases (b_1, b_2), describing the conditional probability of finding base b_2 at a point in the sequence, given that b_1 was present one base-pair earlier in the sequence. We refer to the null hypothesis that this model applies as H_0.

We will assume that the foreground and background model are complementary. This is an approximation, because sequences might have higher order interactions not explained by either the foreground or background models. Making a simplifying assumption here is unavoidable because of the high complexity of these higher order interactions. For example, polypeptide coding sequences are considered background, and the distribution of the sequence of bases is determined by the effect of the polypeptide sequence on evolutionary fitness; something which would require more knowledge about biological function than is available, and is too complex to include in the background model.

However, the model nevertheless provides a principled approach for correcting for the length of the sequence, and for differences in the frequency of bases or pairs of bases. Hence,

$$P(\mathbf{B}|H_0) = q_{B_1} \Pi_{i=2}^{n} t_{B_{i-1} B_i} \tag{3.8}$$

Recall that B_i is the ith nucleotide in the sequence being tested for a motif occurrence.

Combining the models

In order to combine the foreground and background models, we start with Bayes' theorem:

$$P(H_1|\mathbf{B}) = \frac{P(\mathbf{B}|H_1)P(H_1)}{P(\mathbf{B})} \tag{3.9}$$

We assume the foreground and background models are complementary, so

$$P(\mathbf{B} \cap H_1) + P(\mathbf{B} \cap H_0) = P(\mathbf{B} \cap (H_0 \cup H_1)) = P(\mathbf{B}) \tag{3.10}$$

$$P(\mathbf{B}) = P(\mathbf{B}|H_1)P(H_1) + P(\mathbf{B}|H_0)P(H_0) \tag{3.11}$$

Due to complementarity,

$$P(H_0) = 1 - P(H_1) \tag{3.12}$$

This leaves the prior probability $P(H_1)$ as the only remaining unknown. This should be an estimate of the rate of occurrence of the TFBS in the genome (or other set of sequences being searched). As this is not known, we make a plausible assumption about $P(H_1)$, and later discuss the sensitivity of the accuracy of the method to this parameter.

We note that this combination of foreground and background models is able to represent a number of features to the extent that the information is present in the raw counts matrix. For example, gaps in the sequence correspond to regions in which the foreground is indistinguishable from the background, in which the value of p_{ib} is identical to the probability of finding the base in the background. Similarly, palindromes can be represented merely by the incorporation of the palindromic pattern into $\mathbf{p}$. For this reason, there is no need for any special steps to be taken to allow BaSeTraM to find gapped or palindromic TFBS.

Comparison with other work

Our model shares some similarities with the model used in a previous study [Lähdesmäki et al., 2008]. However, we have taken a different approach at a number of points, as we discuss below. The most notable benefit of our approach compared to the Bayesian approach presented in the paper is that the approach of Lähdesmäki et al. requires a computationally expensive Markov Chain Monte Carlo (MCMC) procedure, while we can efficiently compute the posterior probability for a given motif being at a given position.

One major difference between the two approaches is that Lähdesmäki et al. aims to identify the posterior probability of alignments of one or more motifs in a given promoter region, while BaSeTraM computes the probability that a single motif is found at a given site, and uses this to annotate a sequence with probable sites. Another difference is that BaSeTraM does not take into account uncertainty in the background probabilities (and instead focuses entirely on

the uncertainties in frequencies in the foreground model). This approximation can be justified by the large quantity of data available to build the background model (as opposed to the foreground models), and the correspondingly low estimator variance. Using this simpler background model allows BaSeTraM to efficiently use a context-dependent background model.

In addition, Lähdesmäki et al. used a different derivation, by representing all foreground model frequencies at each position using a four-way multinomial distribution across all bases. In this paper we instead use a binomial distribution, where one Bernoulli outcome is that a base at position i used to build the PWM row $\mathbf{F}_i$ matches the base B_i, and the other is that it does not. In other words, we build a model of the motif matrix specific to $\mathbf{B}$, while Lähdesmäki et al. builds a general model. As discussed in the Implementation section, our formulation allows us to find a computationally efficient closed form solution (dependent on pre-computed values of the β function) for the posterior probability.

Implementation

We developed an implementation, SBaSeTraM, of the Bayesian search method, BaSeTraM, described above. We also implemented the method described in Kel et al. [2003], and refer to the implementation as GMATIM. As the implementation of MATCHTM provided by the authors of that paper is closed source, GMATIM may differ from the BioBase MATCHTM implementation. For example, that paper stated that "the core of each matrix is defined as the first five most conserved consecutive positions of a matrix". However, we have been unable to determine how the level of conservation of each group of 5 consecutive positions is measured and compared. To resolve this issue, we implemented GMATIM to simply find the 5 most conserved positions, where conservation at position i is measured as $\max_b f_{ib}$.

In addition, we have created a wrapper, called WrapMAST, around the stand-alone MAST [Bailey and Gribskov, 1998] binary, which we built from the MEME 4.4.0 source code (downloaded from http://meme.sdsc.edu/ on the 2nd of July, 2010). WrapMAST converts matrices from TRANSFAC into the form produced

by MEME. This involves converting the matrix of frequencies to a matrix of log-odds **L**. We have used the same formula used in the MEME software (using a background proportion of 0.25 for each base):

$$L_{ij} = \log_2(4\hat{p}_{ij} + 10^{-200}) \tag{3.13}$$

$$\hat{p}_{ij} = \frac{F_{ij}}{\sum_k F_{ik}} \tag{3.14}$$

The addition of 10^{-200} is used (as in MEME) in ensure that L_{ij} has a real value even when $\hat{p}_{ij} = 0$. For each PWM, WrapMAST invokes MAST in hit list mode to search all probes. It then parses the output from MAST and outputs them in the same format used by SBaSeTraM (but with the p-value from MAST used in place of the posterior probability from SBaSeTraM).

SBaSeTraM, GMATIM, and WrapMAST are written in Haskell, and we have aimed to make the source code of each program a succinct and readable description of the corresponding algorithm. SBaSeTraM, WrapMAST, and GMATIM provide a similar command line interface (and share common code), so as to simplify the design of analyses which compare the algorithms.

Due to the possibility of numerical underflow from very small probabilities, our SBaSeTraM and GMATIM implementations make use of log probabilities (base e).

It is necessary for SBaSeTraM to compute the posterior probability, $P(H_1|\mathbf{B})$, at every position in the sequence being searched, for every TFBS matrix (with the exception that there is no search for TFBS matrices of length l in a sequence of length n at starting positions $i > n - l$). For this reason, it is important that the posterior probability can be computed efficiently.

The β function has no closed form, and needs to be calculated numerically. To avoid expensive computations in our inner loop, for each matrix, we pre-compute $v_{ib} = \ln \frac{\beta(f_{ib}+2, n_i - f_{ib}+1)}{\beta(f_{ib}+1, n_i - f_{ib}+1)}$ for each i and b. We also pre-compute all partial sums of the series $0 + \sum_{i=1}^{N} \ln t_{B_{i-1}B_i}$, where N is the length of the sequence **B**. Let

s_i be the ith entry in the series, so,

$$s_1 = 0 \tag{3.15}$$

$$s_2 = \ln t_{B_1 B_2} \tag{3.16}$$

$$s_3 = \ln t_{B_1 B_2} + \ln t_{B_2 B_3} \tag{3.17}$$

and so on. This means that:

$$\ln P(B_i \cap ... \cap B_{i+l-1} | H_{0,B_i}) = \ln q_{B_i} + s_{i+l-1} - s_i \tag{3.18}$$

$$\ln P(B_i \cap ... \cap B_{i+l-1} | H_{1,B_i}) = \sum_{j=i}^{i+l-1} v_{j-i+1,B_j} \tag{3.19}$$

Note that equation 3.18 is a log-transformed equivalent of equation 3.8, and similarly, equation 3.19 is a log-transformed equivalent of equation 3.7.

$$\begin{aligned}\ln P(H_{1,B_i} | B_i \cap ... \cap B_{i+l-1}) = \ln P(B_i \cap ... \cap B_{i+l-1} | H_{1,B_i}) + \ln P(H_1) - \\ \text{logsumexp}(\ln P(B_i \cap ... \cap B_{i+l-1} | H_{1,B_i}) + \ln P(H_1), \\ \ln P(B_i \cap ... \cap B_{i+l-1} | H_{0,B_i}) + \ln P(H_0)),\end{aligned} \tag{3.20}$$

where logsumexp(x, y) is a function which computes $\ln(e^x + e^y)$ while avoiding numerical underflow for large magnitude negative values of x and y, by computing $a + \ln(\mathrm{e}^{x-a} + \mathrm{e}^{y-a})$ for $a = \max(x, y)$.

We compute the vector q and the matrix t once, across all nucleotide sequences to be processed, by counting the number of occurrences of each base and sequence of the two bases, respectively, and dividing by the pooled total number of occurrences.

For each site, we compute the log-posterior probability and test it against a cut-off (as discussed below) to decide whether the TFBS occurs at that site. We search for sites, both on the sequences provided, and on the reverse complement of those sequences.

We retrieved the online supplement for Harbison et al. [2004] at http://fraenkel.mit.edu/Harbison/release_v24/. This data describes which of 6725 probes each of 182 different transcription factors bound to in a series of

chromatin immunoprecipitation microarray (ChIP-chip) experiments. These probes were between 47 and 2764 base pairs long, with 95% between 92 and 1317 base pairs, 50% between 227 and 647 base pairs, and a median length of 359 base pairs. We also downloaded all TRANSFAC Saccharomyces Module matrices (TSM; [Matys et al., 2006]), as of 2009-06-16, from http://tsm.bioinf.med.uni-goettingen.de/.

Where a matrix used estimated rather than raw counts, as indicated by the occurrence of a decimal point in the 'frequency' matrix, that matrix was excluded (as we have assumed that raw counts will be used).

We filtered the set of probes, based on the experimental data, to only include those to which a transcription factor bound (for which we had a corresponding PWM). This left 1259 probes.

We then used each method to search the entire set of probes for TFBSs corresponding to each matrix, across all positions in the probe. Where the method detected the occurrence of a TFBS for a particular TF at any position in a probe, a positive result for that TF-probe combination was recorded. If no TFBSs were found at any position for a given TF a negative result was recorded. These results were then compared against the 'gold standard' experimental data. Only TFs which had corresponding matrices in TSM, and were also in the experimental results, were included.

We classified each included TF-probe pair into 4 categories:

- True Positive (TP) - positive prediction, and experimental determination of TF-probe interaction;
- False Positive (FP) - positive prediction, but no experimental determination of TF-probe interaction;
- True Negative (TN) - negative prediction, and no experimental determination of TF-probe interaction;
- False Negative (FN) - negative prediction, but experimental determination of TF-probe interaction.

In this paper we have used 0.001 as an approximation of the prior probability, because this value is credible as a frequency of occurrences. To determine the sensitivity of this parameter, we tested values that were one order of magnitude higher, and one and two orders of magnitude lower. The posterior probabilities obtained from doing this are increased or decreased, respectively, but once this is taken into account when selecting cut-offs, there is very little difference in the results within this range of prior probability parameters.

3.4 Results and Discussion

There were 38 different transcription factors in TRANSFAC Saccharomyces Module, of which 32 were made up of raw counts. Of these, 16 were also found in the ChIP-chip dataset. These were tested against the 1259 different probes in the chromatin immunoprecipitation experiment. This gives 20144 different TF-probe pairs where we can classify whether the TF binds to the probe, and then check the classification. These results are shown in Figure 3.1.

We generated a ROC curve (Figure 3.1) for SBaSeTraM, by varying the posterior probability cut-off, and hence the trade-off between sensitivity and selectivity.

The point on the ROC curve generated using the parameters from [Kel et al., 2003] with GMATIM appears slightly below the ROC curve for BaSeTraM (GMATIM has 71.61% true positive rate for a 53.27% false positive rate). We found a posterior cutoff that generates a FPR close to this (with a posterior probability cut-off of 0.407, BaSeTraM achieved a 72.07% TPR at a FPR of 53.25%). At this point, we tested for a significant difference in the proportion of predictions which were correct; that is, $\frac{\mathrm{TP+TN}}{\mathrm{TP+TN+FP+FN}}$. We performed a comparison of these two binomial proportions, using the prop.test function in R [R Development Core Team, 2009], and obtained a one-sided p-value of 0.4603 (*i.e.* not significant to a 95% confidence level).

SBaSeTraM outperforms MAST when used through WrapMAST. It is worth noting that MAST is not typically used with TRANSFAC PWMs, and usually, multiple PWMs are used for each TF, and so the results cannot be used to

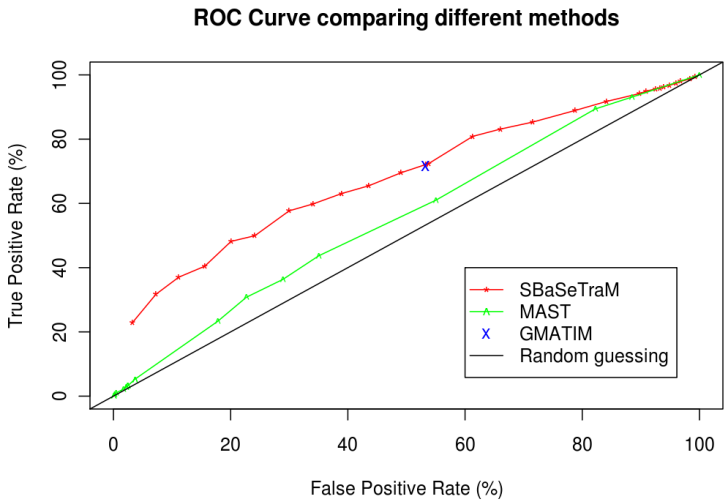

Figure 3.1: A receiver operating characteristics (ROC) curve comparing SBaSeTraM, GMATIM, and MAST. For SBaSeTraM, the posterior cut-off was varied to obtain a series of points. For MAST, the p-value cutoff was varied. For GMATIM, the parameters listed in the MATCHTM paper were used to generate the point on the curve.

make inferences about how well MAST performs together with MEME. The results do, however, illustrate the benefit of methods which take into account uncertainty in the foreground model.

We also carried out an analysis to see whether any particular TFs were making a large contribution to the overall prediction accuracy at this point. Figure 3.2 shows the differences between the two methods in the ROC space for each TF PWM. For each transcription factor, we have plotted an arrow from the point in the ROC space corresponding to the results for SBaSeTraM, to the point corresponding to the results from GMATIM. Some of the predictions are quite different; for example, for ADR1, SBaSeTraM found no occurrences, while GMATIM made numerous predictions, resulting in a true positive rate of 91.3% and a false positive rate of 96.0% (putting the accuracy for that particular TF below the line of no-discrimination). There was only one TF, GAL4, for which SBaSeTraM fell below the line of no-discrimination (which GMATIM predicted with a 17.4% true positive rate and a 0.8% false positive rate), and

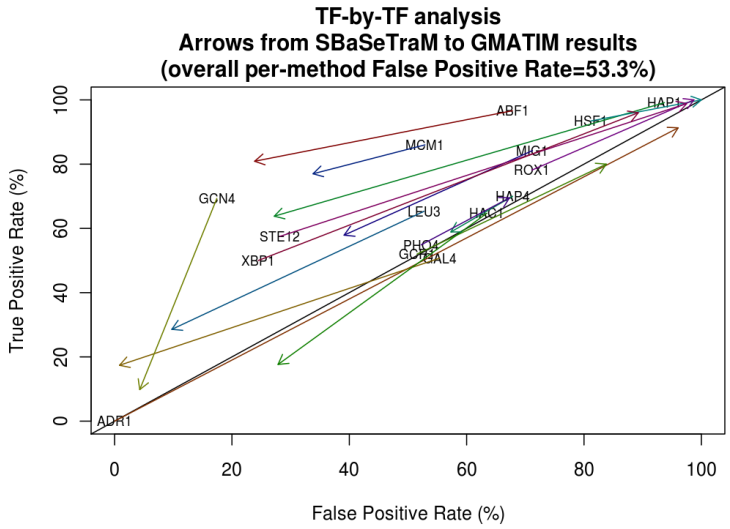

Figure 3.2: Comparing SBaSeTraM to GMATIM predictions for each transcription factor. The results are shown with the overall False Positive Rate for SBaSeTraM matched at that obtained from GMATIM with the parameters in the MATCHTM paper, namely 53.3%. Arrows run from the point obtained using SBaSeTraM to the point obtained using GMATIM.

three TFs for which GMATIM fell below the line of no-discrimination (all of which were above or on the line of no-discrimination for SBaSeTraM). Unlike for SBaSeTraM, GMATIM predictions for HSF1, ROX1, and STE12 had true and false positive rates approaching 100%.

We also analysed the spread of true and false positive rates for each method. Figure 3.3 shows box-and-whisker plots for the true and false positive rates for SBaSeTraM and GMATIM. Notably, there is a much greater distance between the upper and lower quartiles in both the true and false positive rates for GMATIM than there is for SBaSeTraM. This demonstrates that the BaSeTraM algorithm is more consistently controlling the trade-off between sensitivity and selectivity for each individual TF.

In addition, we used the bisection method to find a separate posterior probability cutoff for each of the 16 TFs that gave the SBaSeTraM method a FPR (for that TF) close to the FPR obtained with GMATIM. We allowed the method to terminate when a cutoff was found that brought the L_1 distance of the two

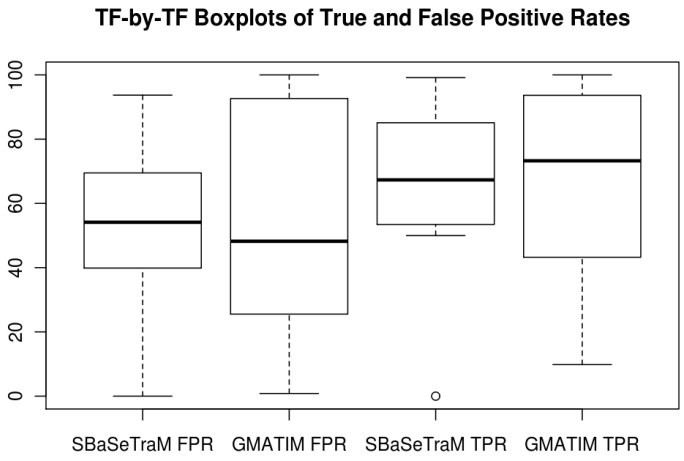

Figure 3.3: Box and whisker plot showing the spread of true and false positive rates for SBaSeTraM and GMATIM. The results are shown with the overall False Positive Rate for SBaSeTraM matched at that obtained from GMATIM with the parameters in the MATCHTM paper, namely 53.3%.

FPRs within 0.001%, when an increase in cutoff resulted in an increased FPR (or a decrease in the cutoff resulted in a decrease in the FPR), or when no improvement in FPR was achieved after 4 iterations of the algorithm. The latter two conditions are necessary because there are a finite number of probes (1259), and there is no guarantee that there will be a cutoff which brings the SBaSeTraM FPR within 0.001% of the GMATIM FPR. In practice, for 8 of the 16 TFs, the difference between the final FPRs for the two methods was less than 0.001%, for 11 it was within 0.25%, and for 13 was within 0.5%. For HAC1, the final SBaSeTraM FPR was 0.966% higher than the GMATIM one, for XBP1 the GMATIM FPR was 1.158% higher, and for HAP1, the final SBaSeTraM FPR was 2.012% higher.

Using the same methodology used on the entire dataset (as discussed above), we tested for a statistically significant difference in proportion of predictions which were correct for each transcription factor, between GMATIM and SBaSeTraM (with the posterior probability cutoffs discussed in the previous paragraph). We obtained only one result where the p-value was less than 0.05, for GCN4

(p=0.00886). For this TF, the FPR for both methods was 4.288%, the TPR for SBaSeTraM was 38.037%, while it was 9.816% for GMATIM. When we applied the Holm-Bonferroni procedure for multiple comparisons [Holm, 1979], none of the TF-by-TF results were significant to a 5% familywise error rate (FWER).

Conclusions

We have developed a Bayesian classifier for identifying TFBSs, which performs comparably to an existing algorithm, but which has a more principled statistical explanation, so that the trade-off between sensitivity and selectivity can be trivially adjusted, and the method can be altered to use different background models.

It is clear that the two methods are very similar in overall performance, and there is insufficient data in TSM to tell the two apart. The 95% confidence interval for the difference of the proportion correctly classified above runs from SBaSeTraM being 1.03% better, to GMATIM being 0.93% better. We therefore conclude that until there is more evidence that one method is better, from a performance standpoint, the two methods can be used interchangeably.

However, the fact that the statistical interpretation of BaSeTraM has been explained in rigorous terms, combined with the ease with which the posterior probability cut-off can be adjusted (as opposed to needing to adjust two separate parameters and re-run the analysis) makes the use of BaSeTraM preferable for many applications.

We note that despite the similarity in accuracy, the predictions made are not all the same; only 62.8% of all predictions of transcription factor binding made by SBaSeTraM with this posterior probability cut-off were also made by GMATIM.

The BaSeTraM statistical model includes a background model to be used. While a relatively uninformative background model is useful with the synthetic probes used in ChIP-chip analyses, using a different background model is likely to be important on genomic scale data, where there are localised variations in base frequencies.

When dealing with genomic scale data, it is also important that computation is reasonably efficient. It is also preferable that this computation can occur on modest hardware, so it is usable by groups without access to high-performance computing infrastructure.

In order to achieve these goals, we also developed a C++ implementation of BaSeTraM, called CBaSeTraM, which we optimised for the AMD64 architecture. We used Callgrind [Weidendorfer, 2008] to identify places where cache misses were occurring. We then used a customised allocator to ensure that all data which is needed in the inner loop (which is executed for each matrix for each alignment for each position) does not result in any cache misses, due to it being present in one cache page. As reading the level 1 and 2 caches are approximately 10 and 300 times faster than RAM, respectively, this leads to significant speed-ups. In this tool, we also implemented a sliding window determination of background model parameters q_b and $t_{b_1 b_2}$. Our implementation supports two distinct sliding windows; the intention is that one window is much larger than the other. The final estimate of each q_b and $t_{b_1 b_2}$ is the geometric mean of the two estimates. By default, the small window is 501 BP wide, and the large window is 2001 BP wide. Both windows are centred on the same base, which is used as the first position when testing for TFBSs. In addition, CBaSeTraM can use MPI [Gropp et al., 1999] to search multiple sequences in parallel.

GMATIM, SBaSeTraM, and CBaSeTraM, as well as the programs used to test the methods, are Free / Open Source software. Instructions for building these programs are included as an online supplement.

Chapter 4

Literature Review: Supervised Machine Learning with Support Vector Machines

In this PhD research project, supervised machine learning using support vector machines is used to make inferences about Gene Regulatory Networks. This review chapter provides background on supervised learning using support vector machines.

4.1 Introduction

A major dichotomy of machine learning techniques is between supervised machine learning and unsupervised machine learning. Supervised machine learning takes training data, for which the answer to some question is known, and from this aims to learn how to predict data for the testing set. This is contrasted to unsupervised machine learning, which identifies patterns in the data without any examples of a correct answer. The motif finding algorithms discussed in section 2.6 provide examples of unsupervised machine learning algorithms. The remainder of this section will focus on supervised machine learning algorithms.

Supervised machine learning algorithms exist for both classification and regression problems. The former problem takes some form of data for different cases as input, and classifies each case into a category. The regression problem, on the other hand, tries to predict one or more continuous variables for each case.

There has been a great deal of research into machine learning over the past 50 years or so, viewed both from an empirical standpoint of testing methods and observing how well they work, and from a theoretical standpoint in terms of moves towards more principled machine learning methods.

4.2 The regression problem

In my review of the literature, I will focus primarily on methods which can be used for approximate regression. The regression problem can be expressed as follows:

For a random variable $\mathbf{X}$ (which is a vector of dimension one or higher) and a scalar random variable Y, find $\mathbb{E}(Y|\mathbf{X})$. Algorithms for this work in two phases. In the first phase, a series of paired samples of $\mathbf{X}$ and Y are provided to the algorithm. This data is referred to as the training data. In the second phase, samples of $\mathbf{X}$ for which Y is unknown are provided, and the algorithm needs to estimate an expectation for Y. The way in which training is performed can be used to categorise different types of machine learning methods. Eager learning methods work out details of the expectation function before any expectations need to be predicted, and then use this to estimate expectations. On the other hand, lazy learning methods store all the training data, and then use the training data each time an expectation needs to be computed. A second dichotomy that is useful for eager methods is to classify them as online or offline methods. Online methods update their internal representation of the conditional expectation immediately after each case is provided. On the other hand, offline methods store all data until all training data has been provided, and then process the training data as a batch.

4.3 Local methods

Local regression methods are lazy learning methods which work by identifying samples of $\mathbf{X}$ from the training which are, by some measure, close to the $\mathbf{X}$ at which we want to predict the conditional expectation of Y.

Cleveland [1979] introduced an algorithm called LOWESS. This algorithm fits a polynomial to all points, with the contribution of each sample to the fit being in inverse proportion to how far away the point is from the point at which the regression is being found. A more precise specification of the algorithm [Cleveland and Devlin, 1988] proposed using a Euclidean distance metric on dimensions scaled by the interquartile distance, and transforming the weights by $W(u) = (1 - u^3)^3$, where u is the distance normalised by the qth nearest point (where q is a parameter, with points further away than the qth nearest point given zero weight).

The algorithm used to perform the fitting was described in Cleveland et al. [1988]. The algorithm starts by repeatedly bisecting the data by planes through the median in each dimension, and then recursively bisecting the resulting segments until segments contain less than 5% of all points (creating a k-d tree). Within each segment, nearest neighbours are identified using an efficient (linear) partial sort algorithm, and the local model is fitted.

4.4 Kernel machines

Support vector machines are a family of methods which have proved useful for a wide variety of machine learning problems. Variants of the basic concept can be used for both the supervised classification problem (predicting a categorical variable from a vector of real valued dimensions) and for regression (predicting the expected value of a real valued variable given a vector of real values). If we restrict the former problem to two classes (also known as pattern recognition, where one class indicates the presence of the pattern and the other class indicates the absence), and express class membership as an indicator function, then both problems can be seen as the problem of learning a functional relationship.

Vapnik and Lerner [1963] introduced the first maximum margin algorithm for pattern recognition. The algorithm takes all the examples (which describe whether the pattern was present, and the corresponding input vectors) and, if one exists, finds the hyperplane in input space which separates the points, and which maximises the distance to the nearest points to the hyperplane. In later

literature, these nearest points are referred to as the support vectors.

Boser et al. [1992] extended this approach by allowing for the input vectors space to be mapped to a feature space. However, to avoid the computational complexity of mapping into the feature space and then computing the optimal hyperplane in the feature space, the method instead makes use of the fact that only dot products of feature space vectors are needed. To obtain these dot products, a kernel function, which takes two input space vectors, and computes the feature space dot product, is used. Kernel functions have been derived for a wide variety of mappings, including arbitrary order polynomials and radial basis functions. This general technique is used in a wide variety of different machine learning methods, which are collectively called kernel machines.

The probably approximately correct (PAC) model of learning Valiant [1984] is a key statistical model for machine learning. Under this model, the problem of learning is defined as choosing a function f that achieves an loss of ϵ or less with probability δ or greater, for a binary function $f \in F$ (where F is the set of all possible functions) and a set of true examples sampled from a probability distribution. The training algorithm may be given access to an oracle which can determine whether a given input has the property.

There are several theoretical frameworks for machine learning algorithms. In all of the statistical algorithms, it is assumed that there is an underlying joint cumulative distribution $F(x, y)$ from which both the training examples and the points at which we need to predict are sampled. The *loss function* $L(y, x)$ measures the cost of predicting y when the true value is x. The *risk functional* is the expected value of the loss for a given function over the distribution, i.e. $R(g) = \int_{\mathbf{x},y} L(y, g(\mathbf{x}))dF(\mathbf{x}, y)$. Ideally, we would want to minimise the risk. The problem with this is the underlying joint distribution is unknown. One technique available is empirical risk minimisation. The principle is to estimate the risk using the training data, and minimise this instead, or in other words, for n samples $(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)$ to minimise $\frac{\sum_i^n L(y_i, g(\mathbf{x}_i))}{n}$. The problem with this approach is that the samples make up only a part of the full sample space, and, if too much capacity for functions to fit any data is allowed, the empirical risk will be low, but the actual risk will be higher (this is

called the overfitting problem; an example would be choosing a function which returns the correct values for the training data but completely incorrect values at other points). One way around this problem is to optimise over a set of functions which all have the same capacity to fit data. This approach is called structural risk minimisation [Vapnik and Chervonenkis, 1974]. The capacity of a set of functions is measured using the VC-dimension of the problem. The optimal VC dimension is subsequently found by optimising over held out data. The VC-dimension is defined so that for a set of real-valued functions F, the VC-dimension n of F is the maximum possible size of a set of feature space vectors $\mathbf{X}$ and true outputs Y so that for every possible $\mathbf{X}$ such that $|\mathbf{X}| = n$, there is an $f \in F$ such that $L(Y_i, f(X_i)) < \beta$, where β is some arbitrary fixed cut-off.

Support vector machine classifiers achieve structural risk minimisation because there is a relationship between the margin and the VC dimension. To extend this to learning of functions which return real values, the loss function can be changed to an ϵ-insensitive loss function. This loss function has no loss for errors less than ϵ. Larger values of ϵ correspond to smaller VC dimensions. Minimising the empirical risk for a ϵ-insensitive loss function and then optimising for ϵ using the cross-validation error is the basis of the ϵ support vector regression algorithm [Vapnik, 1999]. The algorithm can be visualised in two dimensions (one input dimension, one output) as placing a tube of diameter ϵ in the feature space, trying to include as many points as possible in the tube, and minimising the distance from the tube to the remaining points (the points in the tube contribute zero loss, and points outside of the tube contribute a loss in proportion to their distance from the outside of the tube). The line through the centre of the tube represents the regression function in the feature space. Note that the algorithm is linear in the feature space, but as with all Support Vector Regression methods, can be non-linear in the input space due to the kernel mapping.

Scholkopf et al. [1998] (as explained more extensively in Scholkopf et al. [2000]) extended the ϵ-SVR algorithm by adding a parameter ν, representing the proportion of points which are outside of the tube (taking a value $\nu \in (0, 1)$). The width of the tube (ϵ) is adjusted automatically in this method. The

parameterisation of the model in terms of ν more naturally represents the complexity of the model (having fewer points outside the tube means that the model has less capacity to fit arbitrary data). One important result proved in Scholkopf et al. [2000] is that although only the points outside of the tube contribute to the loss, the local position of the outliers (points outside the regression tube) do not affect which points are inside or outside of the tube. This explains the robustness of support vector regression to noise due to outliers.

A key aspect to the usefulness of support vector regression (and other support vector approaches) is the availability of efficient algorithms for determining the support vectors. The support vector regression problem, when converted into the dual form, becomes a quadratic optimisation problem [Boser et al., 1992]. However, this problem cannot feasibly be solved for large data sets using standard quadratic programming solution techniques when there are a large number of training examples (because a non-sparse Q matrix representation has n^2 entries for n training points). However, because most training examples do not form support vectors, the Q matrix is sparse (because entries are zero when the Lagrange multipliers of either the row or column point are zero, i.e. the point is not a support vector). Vapnik [1979] / Vapnik [1982] introduced an algorithm that is commonly referred to as chunking. This algorithm starts with an initial guess of the support vectors, and builds a Q matrix containing rows and columns for support vectors only. From this set of support vectors, the algorithm attempts to solve the QP problem. The results from this allow the algorithm to determine which points meet the Karush-Kuhn-Tucker conditions, and for the points which don't, quantify how far numerically they are from meeting the condition. The points in the solution of one iteration which don't appear to be support vectors are removed from the next iteration, while up to M points (if there are more than M candidates, the worst M) which don't meet the KKT conditions are added to the set of putative support vectors for the next iteration. The algorithm continues until a set of support vectors are found so the KKT conditions are met.

Osuna et al. [24-26 Sep 1997] noted, however, that in the case of hard problems, in which the generalisation error and hence the ratio of support vectors to

points is high, as well is in the case of very large data sets, where there are still a large number of support vectors despite a low support vector to point ratio, the chunking algorithm is insufficient. Osuna et al. [24-26 Sep 1997] addressed this problem by dividing the training points into two data sets, B and N, where B is called the working set, and proving that moving a point which violates the optimality criteria for an optimisation subproblem from N into B will always decrease the cost function. This allows for the algorithm to keep the number of active support vectors constant; one support vector is added, and one is removed, in each step.

Platt [1998] introduced a more efficient algorithm, which has been shown to be about 1000 times more efficient than chunking on practical data sets, called Sequential Minimal Optimization (SMO). The same algorithm can be used for both classification and regression [Flake and Lawrence, 2001]. The algorithm works by only ever solving quadratic programming problems with two data points at a time, and therefore optimises two Lagrange multipliers at a time. As long as at least one of the points violated the KKT conditions before the step, convergence is guaranteed by the theorem in Osuna et al. [24-26 Sep 1997]. The algorithm works by starting from an initial set of points in the working set, and then alternating between two different point selection heuristics. The first heuristic iterates over all points, and identifies those which don't meet the KKT conditions, from which two are selected for refinement of the Lagrange optimisers. The second heuristic is itself iterative. At each inner iteration, a couple of points which have non-zero Lagrange optimisers are selected. This second heuristic terminates when all KKT conditions are met within a set error tolerance. Part of the reason for the efficiency of this method is that two point QP subproblems can be optimised analytically, while the algorithms of Vapnik [1979] and Osuna et al. [24-26 Sep 1997] require a numerical solution of the larger QP subproblems.

The support vector machine algorithms discussed so far carry out a form of inference known as inductive inference; that is, they take specific training examples, and from this, estimate a general model that fits the data. They then use this general model to predict specific information for the test data. Another

form of inference that many machine learning methods can be adapted to is called transductive inference. Transductive inference takes specific training data, and a specific test point at which to predict specific information, without going through a general model. Because this allows the results of the training part of the algorithm to be focused on the required point, much more efficient use of the training data can be made. The disadvantage of transductive inference is that potentially expensive training is required for every point. Joachims [1999] introduced a transductive support vector machine (TSVM) algorithm for text classification. At the time of writing Transductive Support Vector Machines have only been used for classification problems, and not for regression. TSVM classification works by finding a classification for the test data point and a separating hyperplane so that the training and test data are best separated by the hyperplane into their respective classes.

There have also been attempts to extend support vector regression to use non-convex loss functions, instead of the convex ϵ-insensitive loss function. This allows non-convex loss functions designed to reduce the influence of outliers, such as equation 4.1 to be used. Zhao and Sun [2008] developed an approach to use the concave-convex procedure (CCCP) to transform the optimisation problem over the non-convex loss function into a convex optimisation. Unlike SMO, which optimises a subset of the variables in the dual form to get a certain answer, the Zhao and Sun [2008] approach uses a Newton-type numerical optimisation approach directly on the primal form (but uses the result from an approach like SMO as the starting point for the optimisation).

$$H_\epsilon(z) = \begin{cases} 0 & \text{if } |z| < \epsilon - h \\ -\frac{(h+|z|-\epsilon)^2}{4h} & \text{if } \epsilon - h \leq |z| \leq \epsilon + h \\ \epsilon - |z| & \text{if } |z| > \epsilon + h \end{cases} . \tag{4.1}$$

4.5 Other SVM applications to gene regulation

Brown et al. [2000] used support vector machines to classify gene functions based on expression microarray data. The method worked by predicting, using

support vector machines, whether or not genes have a particular function, based on the time course of gene expression for that gene following diauxic shift. Annotated genes were compared against an annotated database of gene functions (MYGD). The method was shown to work better overall, compared to the unsupervised classification methods available at the time.

Qian et al. [2003] used support vector machine classification in an attempt to determine whether a given transcription factor binds to the promoter for a given gene, using time-course expression microarray data. Positive training examples were obtained from TRANSFAC [Matys et al., 2006] and SCPD [Zhu and Zhang, 1999]. Negative training examples were created using two different methods: firstly, transcription factors with known binding specificities were assumed not to regulate genes that had no binding site for that transcription factor upstream of them. Secondly, transcription factors with no known binding specificities were used as non-regulators (negative examples) for scrambled expression data. This method was shown to result in comparable levels of agreement to each of two different ChIP-chip studies to the level of agreement between the two experimental studies.

Chapter 5

Predicting structural deficiencies in GRNs with SVR

5.1 Introduction

When building gene regulatory networks using a specific source of biological information, there are likely to be regulatory interactions that are not captured in a particular dataset. This could be due to random variation in the data masking signals, or alternatively, because the method is simply not capable of observing a particular type of regulatory interaction.

It would therefore be helpful to know if the topological GRN model is missing any edges into a particular regulated gene, or in other words, to determine whether there is evidence that the expression of a regulated gene is not completely described by its regulators according to a GRN model.

In this chapter, I describe a novel methodology for ranking genes by how well predictions from a topological model fit expression microarray data. In addition, I describe the experiments I performed to determine the trade off between selectivity and sensitivity to detect genes with regulators that were randomly selected for deletion, from a topological model of a GRN.

Tests on the method, using a GRN built from the interactions in YEASTRACT and expression microarray data from gene deletion strains of yeast, showed that it performs better than expected by chance, but only by a small (but statistically significant) amount.

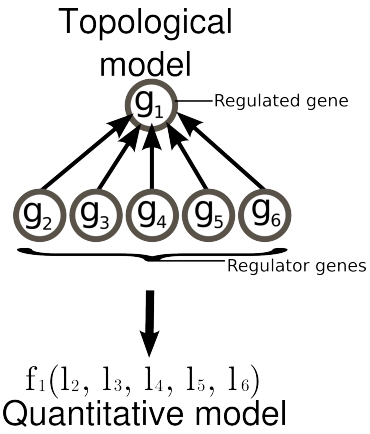

Figure 5.1: Converting a topological GRN model into a numerical one

5.2 Methodology for detecting missing regulatory interactions

The method described in this chapter assesses a topological model that describes qualitatively which genes act as regulators for which other genes. It also requires a series of models for a series of strains where the topological model has been altered (for example, by knocking out or knocking down a gene), and expression microarray data approximating the steady state expression level for each gene.

The method works by taking a qualitative topological models, and converting it into a quantitative model using a regression method (see Figure 5.1).

The numerical model relies on a numerical measure of the extent to which a gene is being expressed. In this present study, I used expression microarray readings as this indicative measure. It is worth acknowledging that the use of expression microarray data means that the method cannot take post-transcriptional regulation into account; for this reason, only models of transcriptional regulation are considered in this chapter.

Let g_i represent the ith gene, and let L_i be a random variable for a measure of the expression of gene g_i. Let n_i be the number of regulators of g_i, and let $r_{i,j} \in \mathbb{N}$ be such that for $j \in \mathbb{N} \cap [1, n_i]$, $g_{r_{i,j}}$ is a regulator of g_i. Then, we define a generic regression function f with parameters α in terms of the conditional

expectation of L_i:

$$f(\alpha_i, x_1, ..., x_{n_i}) \approx \mathbb{E}(L_i | L_{r_{i,1}} = x_1, ..., L_{r_{i,n_i}} = x_{n_i}) \tag{5.1}$$

where α_i contains parameters necessary to approximate the shape of the expectation function, and each x_i is a bound variable local to this equation.

Note, however, that when modelling gene expression in a gene deletion strain, the equation does not apply for genes which have been deleted from the strain. Instead, in a model for a strain where gene g_i has been knocked out,

$$L_i = 0 \tag{5.2}$$

For the purposes of this chapter, I used ϵ-Support Vector Regression, as reviewed in section 4.4. However, it would be possible to substitute an alternative regression method to describe f.

Let $t(\epsilon, \mathbf{y}, X_1, ..., X_{n_i})$ be the training function that estimates the parameter α of the regression function f, given a vector $\mathbf{y}$ of v samples from the output random variable, and a $v \times n_i$ matrix X, where each row X_i contains a sample of the input random variables corresponding to $\mathbf{y}_i$.

Let K be a list of sets, such that $i \in K_k$ if and only if gene g_i was knocked out in the kth microarray experiment. Let $m_{i,k}$ be the expression microarray reading (level) corresponding to gene g_i in the kth expression microarray experiment. Let $M_i = \{j | i \in K_j\}$.

To determine the gene specific error residuals, a leave-one-out cross-validation strategy (LOO-CV) is applied. For each i such that g_i is a gene and each k

such that $K_k \in K$ and $i \notin K_k$:

$$\hat{\alpha}_{i,k} = t\left(\epsilon, \begin{pmatrix} m_{i,1} \\ \vdots \\ m_{i,M_{i1}-1} \\ m_{i,M_{i1}+1} \\ \vdots \\ m_{i,k-1} \\ m_{i,k+1} \\ \vdots \\ m_{i,M_{i|M_i|}-1} \\ m_{i,M_{i|M_i|}+1} \\ \vdots \\ m_{i,|K|} \end{pmatrix}, \begin{pmatrix} m_{r_{i,1},1} & \cdots & m_{r_{i,n_i},1} \\ \vdots & \ddots & \vdots \\ m_{r_{i,1},M_{i1}-1} & \cdots & m_{r_{i,n_i},M_{i1}-1} \\ m_{r_{i,1},M_{i1}+1} & \cdots & m_{r_{i,n_i},M_{i1}+1} \\ \vdots & \ddots & \vdots \\ m_{r_{i,1},k-1} & \cdots & m_{r_{i,n_i},k-1} \\ m_{r_{i,1},k+1} & \cdots & m_{r_{i,n_i},k+1} \\ \vdots & \ddots & \vdots \\ m_{r_{i,1},M_{i|M_i|}-1} & \cdots & m_{r_{i,n_i},M_{i|M_i|}-1} \\ m_{r_{i,1},M_{i|M_i|}+1} & \cdots & m_{r_{i,n_i},M_{i|M_i|}+1} \\ \vdots & \ddots & \vdots \\ m_{r_{i,1},|K|} & \cdots & m_{r_{i,n_i},|K|} \end{pmatrix}\right) \tag{5.3}$$

The trained model is used to generate estimates of each gene in the left-out microarray as follows:

$$\hat{m}_{i,k} = f(\hat{\alpha}_{i,k}, \hat{m}_{r_{i,1},k}, ..., \hat{m}_{r_{i,n_i},k}) \tag{5.4}$$

For $i \in K_k$, instead:

$$\hat{m}_{i,k} = 0 \tag{5.5}$$

However, equation 5.4, when applied to all i (such that g_i is a gene) creates a cyclic dependency between the estimators (because regulators are typically regulated by other regulators to form a GRN with cycles), and so cannot be directly used as a closed form solution for $\hat{m}_{i,k}$.

To address this issue, an iterative algorithm is applied. The algorithm is applied separately for each left out microarray k. The algorithm starts by estimating $\hat{m}_{i,k}$ as the mean value in the other (non left-out) microarrays, $\frac{\sum_{l \in K \setminus \{k\}} m_{i,l}}{|K|-1}$. Then, it proceeds through each $i \in \mathbb{N} \cap [1, |g|]$, and updates $\hat{m}_{i,k}$ using equation 5.4. This process repeats until the convergence condition is reached. To determine if convergence has occurred, let $\hat{m}'_{i,k}$ denote the value of

$\hat{m}_{i,k}$ at the previous iteration, and check for:

$$\sum_{i \in \mathbb{N} \cap [1,|g|]} (\hat{m}_{i,k} - \hat{m}'_{i,k})^2 < \beta, \tag{5.6}$$

where β is the convergence tolerance. For the purposes of this chapter, $\beta = 1$ was used. In addition, an upper limit of ten iterations was imposed. These values were chosen to both make computation feasible, while allowing reasonably good convergence when possible (to put a value of one in perspective, each individual $\hat{m}_{i,k}$ value is typically between 0 and 65536).

The training and parameter estimation process is recalculated for each left-out array, so the left-out array is not used in the training. By summing data for particular gene over the values when each array k is left out in turn, a cumulative residual sum of squared error e_i is computed for each gene:

$$e_i = \sum_{k \in (\mathbb{N} \cap (1,|K|))} (m_{i,k} - \hat{m}_{i,k})^2 \tag{5.7}$$

These e_i are used as an indication of the quality of each model at representing a particular gene. Genes g_i can be ranked by their respective e_i values.

5.3 Methods for evaluating how well the method works

If the method described above is useful, the following prediction would be expected to be true: if we take a model M_1 which describes transcriptional regulation, and delete edges from it at random to get model M_2, we would expect that the genes into which edges have been deleted will show up as genes with high e_i values.

Because of the availability of more comprehensive gene knock-out data sets, the experiment was carried out using *Saccharomyces cerevisiae* expression data.

Hu et al. [2007] profiled the gene expression of both wild-type and knock-out (gene deletion) strains of *Saccharomyces*, including gene deletion strains for 263 different transcription factors, using spotted expression microarrays. The raw data set was downloaded from `http://lad.icmb.utexas.edu/cgi-bin/`

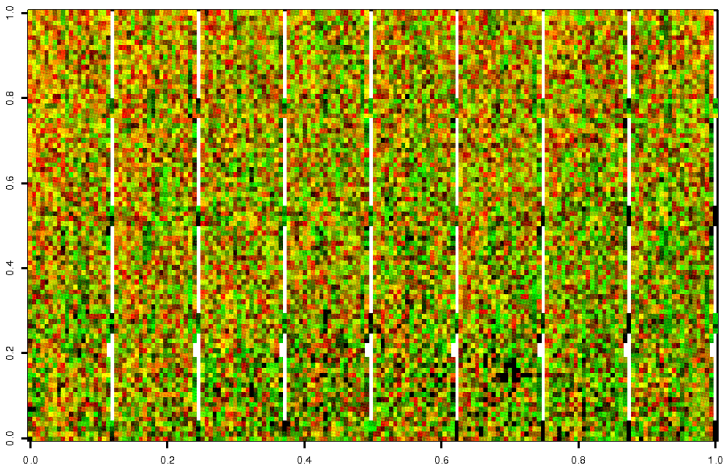

Figure 5.2: Raw microarray readings for one of the arrays from Hu et al. [2007], as a heatmap, one probe per gene, with one channel visualised in red, and the other in green, positioned according to row and column number on the microarray. The white squares show microarray locations where no probes were present. Notice the mild spatial effects such as the higher level of red in the top left corner, and the potential scratch between 0.6 and 0.8 on the X axis and at about 0.15 on the Y axis.

`publication/viewPublication.pl?pub_no=70`. Visualising the raw microarray data as a heatmap by array and spot location showed the types of artifacts commonly found in raw microarray data, as can be seen in Figure 5.2.

To normalise out spatial and dye effects, and facilitate further processing, the following changes were made:

- Data points marked as failed or where the spot flag was set, indicating a problem identifying the spot, in the data set, were excluded.
- Data points where the background levels were above the 97.5th percentile, or below the 2.5th percentile of all data points from that pin tip on either channel were excluded.
- Background correction was performed by subtracting the median background reading from the mean foreground reading (for each channel).

- Values below 0.5 were set to 0.5 to remove negative or zero values.
- MA normalisation was performed using LOESS to remove trends between M (the logged ratio between the channel values) and A (the average logged channel value). This was performed within pin-tips (pin-tip LOESS) and so removed any differences between pin tips.

A wildtype topological model of *Saccharomyces cerevisiae* gene expression was generated based on the YEASTRACT (Yeast Search for Transcriptional Regulators) [Teixeira et al., 2006b] database of known transctional regulatory interactions. YEASTRACT provides information about where known yeast transcription factors are predicted to bind on the yeast genome. Coordinates are are given relative to the sequences from the Sacchromyces Genome Database (SGD) [Cherry et al., 1998]. As SGD also includes annotations describing the mapped location of yeast genes (including the directionality, i.e. what strand the gene is on), it is possible to determine whether genes bind near the promoter region. The promoter region was taken to be 500 base pairs up or downstream from the start codon of the gene. Each interaction from YEASTRACT was matched to the promoter regions from the SGD annotations (giving zero or more putative regulated genes per interaction), and the transcription factor from YEASTRACT was also matched to a gene in SGD (giving one regulator gene per interaction). For each regulated gene, a record of the regulator-regulated gene interaction was recorded, creating the topological model. The distribution of in-degree (number of regulator genes) and out-degree (number of regulating genes) are shown in Figures 5.3 and 5.4, respectively.

Edges were deleted from the wildtype model at random to produce the perturbed wildtype model, with a record of which edges were deleted taken. The original topological GRN model is likely to be missing some true regulatory interactions, and to have some false ones. However, if more than half of all edges are correct, and we delete an edge from the model at random, it is more likely that the edge that was deleted represents a true regulatory interaction than an incorrect one. It is therefore possible to test the method by testing if there are disproportionately many genes with deleted regulators high in the ranked list of genes with putative missing regulators.

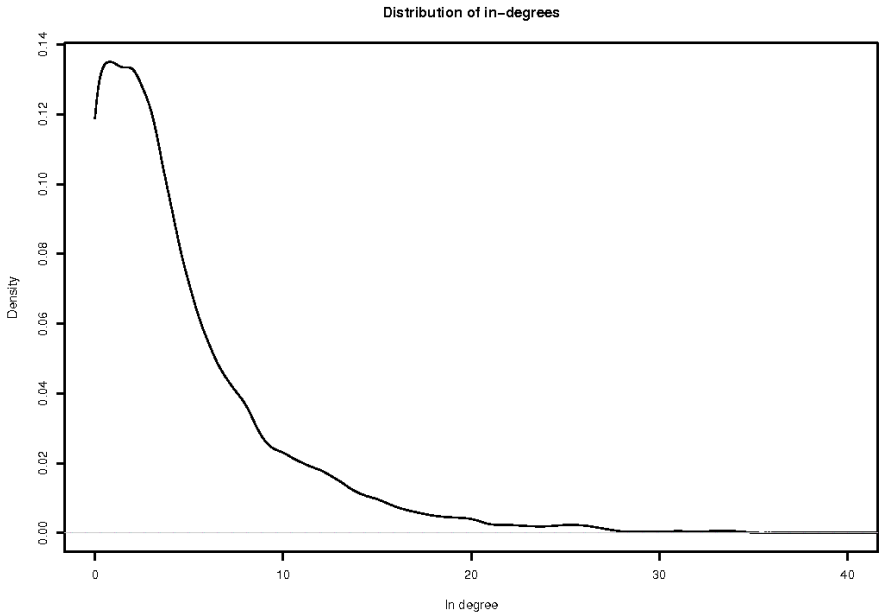

Figure 5.3: The distribution of in-degrees (number of regulators per gene).

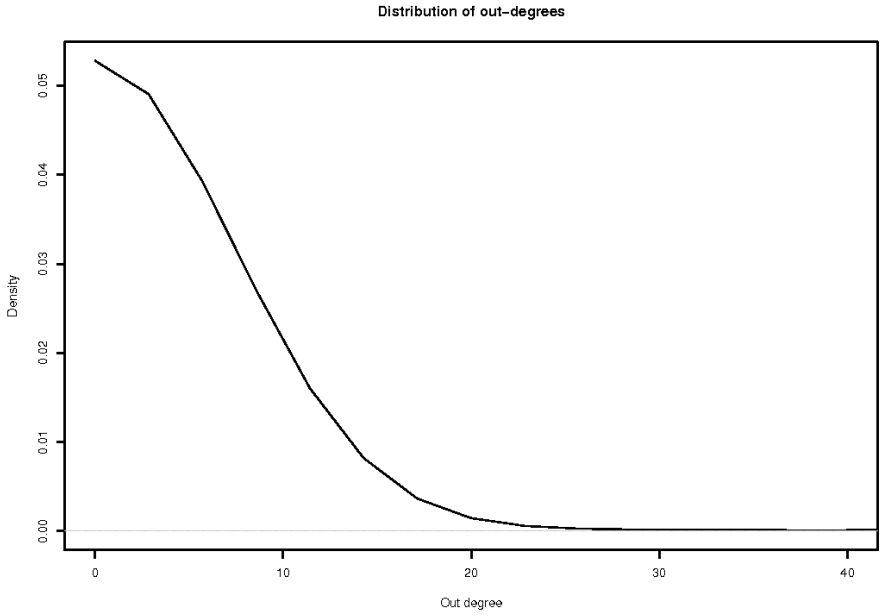

Figure 5.4: The distribution of out-degrees (number of regulated genes per regulator gene).

Interactions were selected for deletion independently and at random, so that each interaction had a 10% chance of being deleted. This was achieved by assigning an independent pseudo-random number from the **Uniform**$(0, 1)$ distribution to each edge in the model, and ignoring edges which had a number greater than the cut-off. The cut-off was set to 0.9 to treat 10% of the edges as deleted.

Original and perturbed wildtype quantitative models were built by converting the qualitative models to quantitative models using the ϵ-SVM Regression function, as described in Section 5.2[1].

Models for each knock-out mutant were created by deleting the regression function for the expression level of the knocked out gene, and instead inserting an equation setting the level to zero, as described above. The remaining regression functions in the knock-out models keep the same unique identifier, and so can share training data.

The perturbed models for each knock-out mutant were processed to yield a binary bytecode program for training the models and making predictions. These byte-code programs were used to carry out leave-one-out cross-validation; each array was successively left out, with all other arrays used for training the regression functions. The iterative procedure described above is then used to estimate the levels of expression of each gene on the left out array using no information from that array.

Once all predictions had been made over all left-out arrays, the residual sum of squares e_i was computed for each gene using equation 5.7.

The SVM parameters ϵ, γ and C were computed using the leave-one-out cross-validation procedure, except that only the first two microarrays in the data set were left out (so as to control the computational cost to a feasible level). The optimal parameters were selected using a coarse grid search with a grid of size $10 \times 10 \times 5$ (that is, 500 evaluations in total for the coarse grid), with grid values equally spaced on the logarithmic scale in the range suggested on the website for libsvm, namely C from 10^{-3} to 10, and ϵ and γ from 10^{-10} to 10^{10}.

[1]These models were represented using CellML, with an extension allowing external functions to be represented as a MathML csymbol element. The regression functions were given a unique identifier.

The grid cell on the coarse grid with the lowest total median residual sum of squared error across all genes and left-out microarrays is then selected for a fine grid search of size $5 \times 5 \times 5$, again on a logarithmic scale. The outcome from the minimisation on the fine grid is taken as the parameters to use.

To determine the statistical significance of a shift in prediction error caused by the deletion of edges, the Mann-Whitney two-sample rank sum test was performed between errors from data points with no edges deleted, and errors from data points with at least one edge deleted. The p-value for the test was significant (1.34×10^{-5}). However, genes with fewer regulators to start with are more likely to have an edge deleted, because there is a constant probability that each edge is deleted, and so this significant result could be due to differences in the original number of regulators. To rule out this possibility, the Mann-Whitney test was also performed only on genes with a fixed number of regulators before edges were deleted. This restriction decreases the amount of data available, and the significance, but a significant result is still obtained for suficiently frequent initial numbers of regulators (recall that Figure 5.4 shows the number of regulators per gene). For genes with a single regulator, there are only 71 points in the data set (for the 10% deletion rate) where there is one regulator that was selected for deletion. For genes with two regulators, the p-value is 3.38×10^{-2}. For genes with three regulators, the result is not significant, while for genes with four regulators, the result is significant (2.10×10^{-3}), even after correcting for multiple comparisons.

Counterintuitively, there is a correlation between increased number of edges in the model after the edges have been deleted and increased model error; this can be seen in Figure 5.5; the distribution can be seen in Figure 5.6. However, despite the general trend towards higher errors as the total number of regulators after deletion increases, there is a trend towards lower errors as the number of regulators deleted increases, as can be seen in figure 5.7 (the distribution can be seen in Figure 5.8).

When only 10% of edges are deleted, it is unlikely that all regulator edges will be deleted except for genes with very few regulators in the model to start with. So that data could be obtained for the full range of deleted edges, the methods

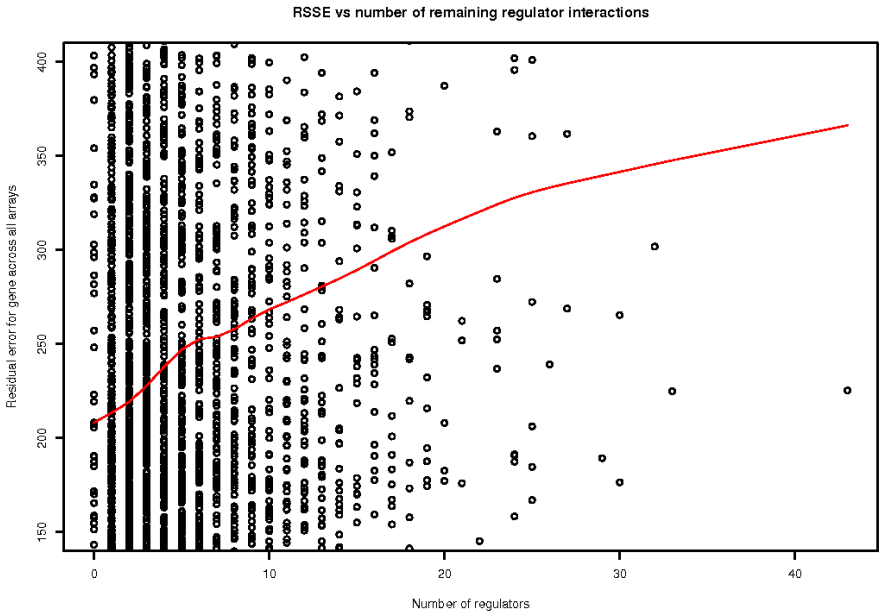

Figure 5.5: The relationship between total residual sum of squared error for a gene, across all arrays, and the number of remaining regulators affecting a gene after edges have been deleted.

described here were repeated deleting 50% and 90% of all edges (and with no deletions). The results from the 10%, 50% and 90% edge deletion data sets were combined to determine the relationship between the ratio of regulator edges deleted from a regulated gene and the total residual sum of squared error for that gene; this is visualised in Figure 5.9.

The results establish that removing regulators from the model increases the error. This suggests that it should be feasible to build a classifier to detect which genes are missing a regulator. Such a classifier was constructed, with a simple cut-off, c, such that if $e_i > c$, gene g_i is classified as missing. By adjusting c, it should be possible to adjust the true and false positive rates of the method.

A Receiver Operating Characteristic curve, showing the trade-off between true and false positive rates, was generated by varying c. This ROC curve is shown in Figure 5.10. The ROC curve shows that the method performs better than

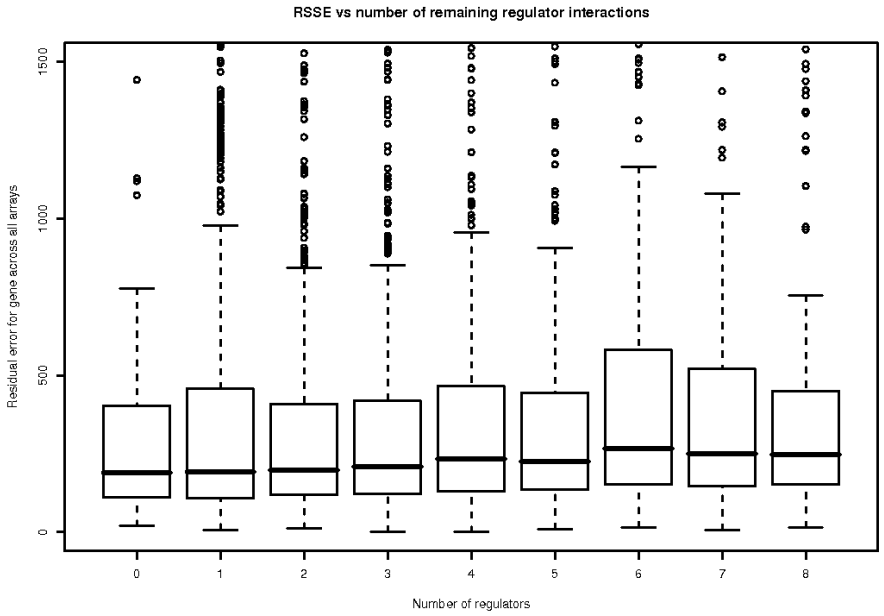

Figure 5.6: A box-plot showing the distribution of the total residual sum of squared error for a gene, across all arrays, for different numbers of remaining regulators affecting a gene after edges have been deleted.

chance (and so above the line of no discrimination), but is not very accurate in absolute terms.

One possible explanation for why the method is not very accurate is that the so called 'false positives' are actually true positives, because the method is detecting genes that are missing regulators that regulate expression by mechanisms other than transcriptional regulation, or by transcriptional binding that is not included in YEASTRACT, or by binding at a distance of greater than 500 base pairs from the start codon. The solution to this problem would be to try the method with a more accurate 'gold standard' initial model. However, until such a model is available, it is not possible to resolve whether the problem is that the method is working but finding unknown missing regulators, or whether the method is simply not very accurate.

Another possible explanation is that the regression method is not able to find a good fit to the data given the amount of training data available. This is only

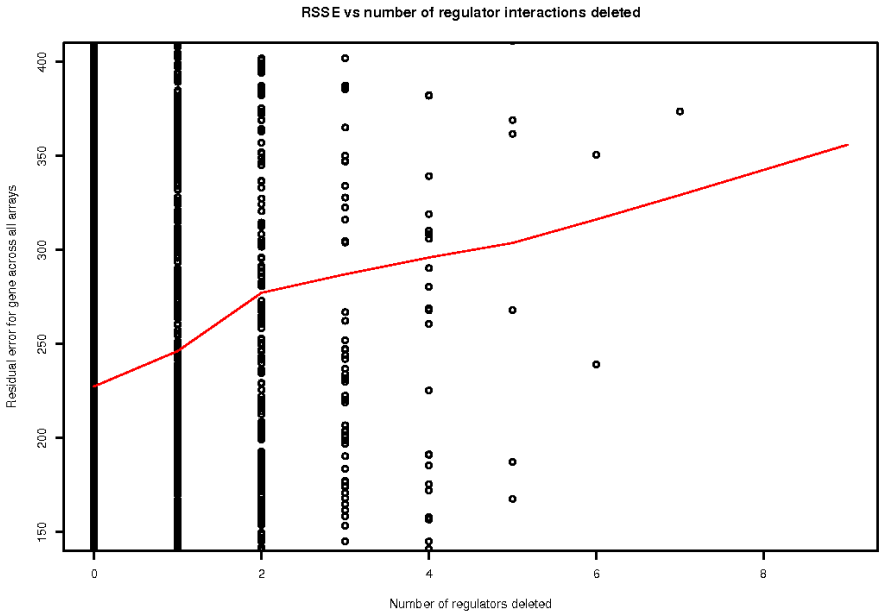

Figure 5.7: The relationship between total residual sum of squared error for a gene, across all arrays, and the number of regulators that were deleted.

likely to be the case if the interaction logic between the expression levels of regulating genes in controlling gene expression is very complicated.

To determine if any particular types of genes were over-represented amongst genes with the highest error, annotations for yeast genes against the Gene Ontology [Ashburner et al., 2000] were obtained from `http://www.yeastgenome.org/gene_list.shtml`. In addition, the full Gene Ontology was obtained. The full gene ontology was applied so that genes that are annotated against an ontology term are recursively also annotated against all terms higher up in the hierarchy. The Mann-Whitney rank-sum test was applied for each term in the ontology, to determine whether genes with that term were statistically overrepresented. The Holm-Bonferroni method [Holm, 1979] was used to control the familywise error rate to 0.05.

Numerous Gene Ontology terms were found to be associated with higher prediction error: association with an organelle, 'nucleobase, nucleoside, nucleotide and nucleic acid metabolic process', 'DNA metabolic process', and 'protein

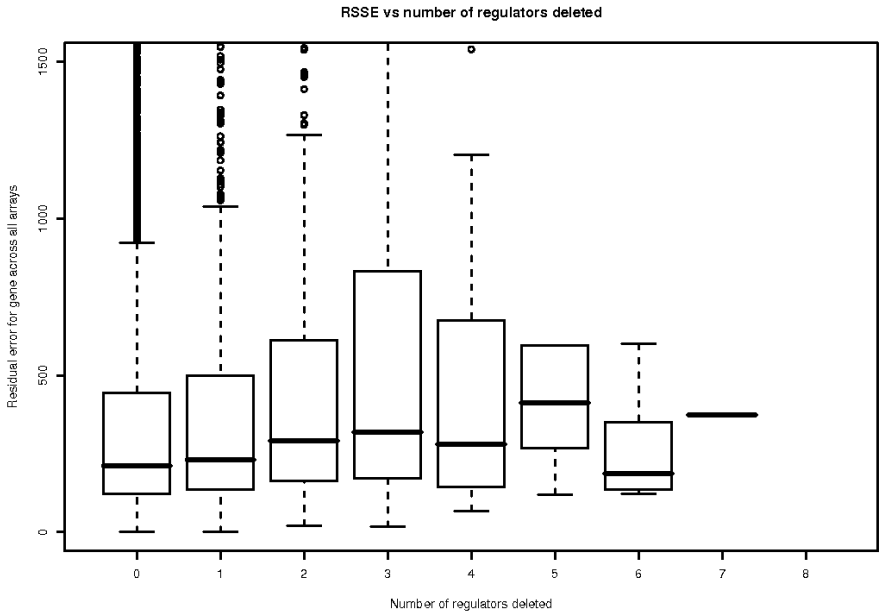

Figure 5.8: A box-plot showing the distribution of the total residual sum of squared error for a gene, across all arrays, for different numbers of regulators that were deleted.

complex' are all examples for which a very small p-value was obtained (well below the threshold to achieve a Family-Wise Error Rate of 0.05 under the Holm-Bonferroni method). These associations could be explained because they are all associated, to some extent, with the cell cycle, and so are the categories of genes for which the highest noise would be expected due to the steady state assumption.

5.4 Conclusion

The method presented in this chapter can detect when edges have been deleted, and has been shown to perform better than chance.

However, we conclude that the experimental data available to date is not sufficient to determine whether or not the method performs well in absolute terms, and so it cannot yet be recommended as a method for determining when

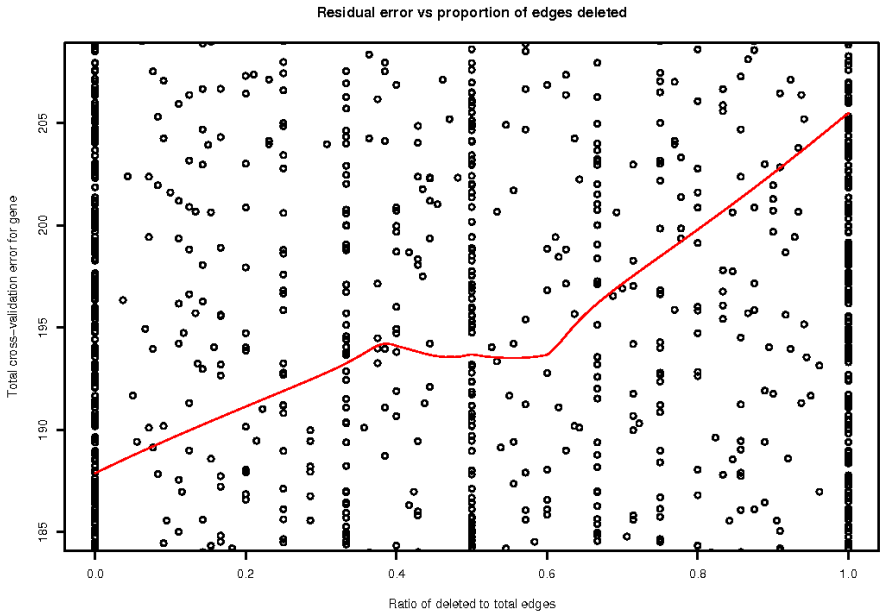

Figure 5.9: The relationship between total residual sum of squared error for a gene, across all arrays, and the number of regulators deleted from the gene at random. Data was pooled from three different fittings of the model, deleting 10%, 50%, and 90% of edges to generate this figure. The range on the y axis has been artificially reduced to show the shape of the curve of best fit.

regulating edges are missing from a gene in a GRN.

A slightly easier problem than detecting whether individual genes are missing regulators is to detect which of two sufficiently different Gene Regulatory Network models is better; this approach is explored, extending some of the methods discussed in this chapter, in Chapter 6. If the models are very similar, for example, only differing by one regulator edge into a gene, then comparing such models breaks down to the same problem discussed in this chapter. However, a method used for validating models does not need to be so sensitive as to detect a single missing edge.

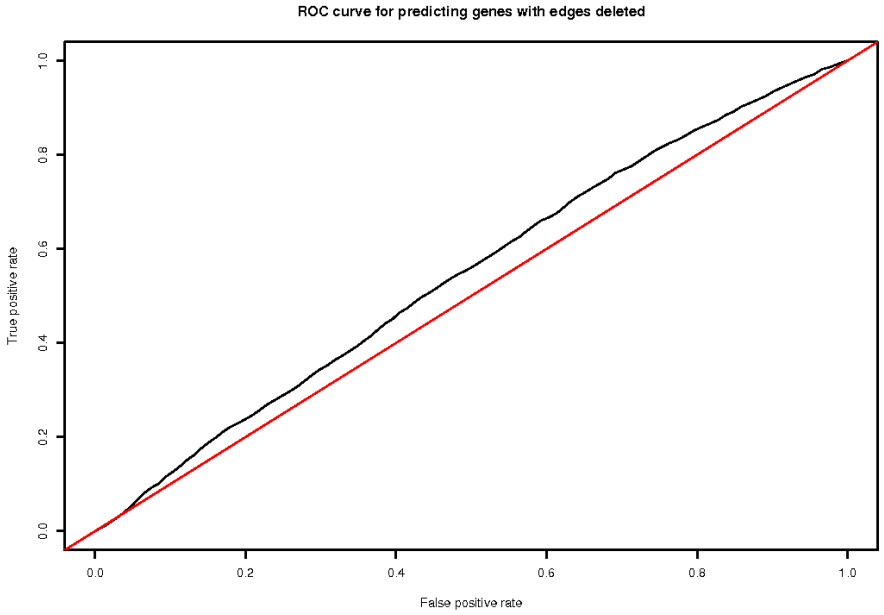

Figure 5.10: The Receiver Operating Characteristic curve, showing how the method to detect missing genes can trade off between true and false positive rate. The true positive rate is the proportion of all genes from which a regulator was deleted that were detected as missing a regulator. The false positive rate is the proportion of all genes that hadn't had an edge deleted that were detected as missing a regulator.

Chapter 6

Validating models of Gene Regulatory Networks with Support Vector Regression

6.1 Introduction

An important aspect of building models of gene regulatory networks is model validation; to gain confidence that a model is valid, a model can tested against experimental data.

Such validation is a crucial part of the scientific method; for a modelling discipline to be scientific (as defined in Popper et al. [1959]), models need to be falsifiable. Testing models against data such that some models will fit the data better than others is one way to achieve this. It is not necessary to rule out the existence of two models that fit the data equally well; in such a case, it is not possible to falsify a claim that one of the models is correct and the other is not, but it is still possible to falsify models that do not fit the data.

If a model is built with a technique which optimises the fit to certain data, that model is not falsifiable by fitting it against the same data; it is therefore important that the method for validating the model uses a distinct source of data compared to those to build the model.

In this chapter, I present a methodology that adapts the use of Support Vector Regression from Chapter 5 to support the validation of models describing the topology of gene regulatory networks. Validation is against expression microarray data.

It is reasonable to expect that a high quality topological model of a gene regulatory network will explain the readings on an expression microarray for a regulated gene in terms of the regulators relatively well compared to a scrambled model. This property is not unique to accurate models of gene regulatory networks; steady state expression information is purely correlative, so a model that explained the correlation by reversing the relationship between a regulator and a regulated gene could fit the data equally well. Similarly, if genes A and B were both regulated by R, a model that assigned A as the regulator of B might fit the data better than a model that omitted the incorrect relationship between B and A. However, despite these limitations, model validation techniques provide a way to compare significantly different models.

Experiments on the method have been carried out using a model of gene regulation in human, testing the method against all expression microarrays on a particular popular microarray platform.

6.2 Building a Genome-wide GRN with TFNetBuilder

In this section I discuss how a gene regulatory network can be built using the method for detecting gene interactions described in Chapter 3.

The BaSeTraM algorithm described in Chapter 3 can be used to search a genome and annotate all sites at which there is a high posterior probability that a transcription factor will bind. In Chapter 5, a model was built by searching for transcriptional regulators close to a putative binding site; in this section, a method for generalising that approach and applying it efficiently to larger genomes is discussed. This approach is implemented in a software package called TFNetBuilder.

The overall process for producing a Gene Regulatory Network is shown in Figure 6.1.

The inputs to the algorithm are the outputs from BaSeTraM (annotating the locations of matches to postion weight matrices in genomes), the genome as a series of GenBank Flat Format (GFF) files (containing a description of assembled

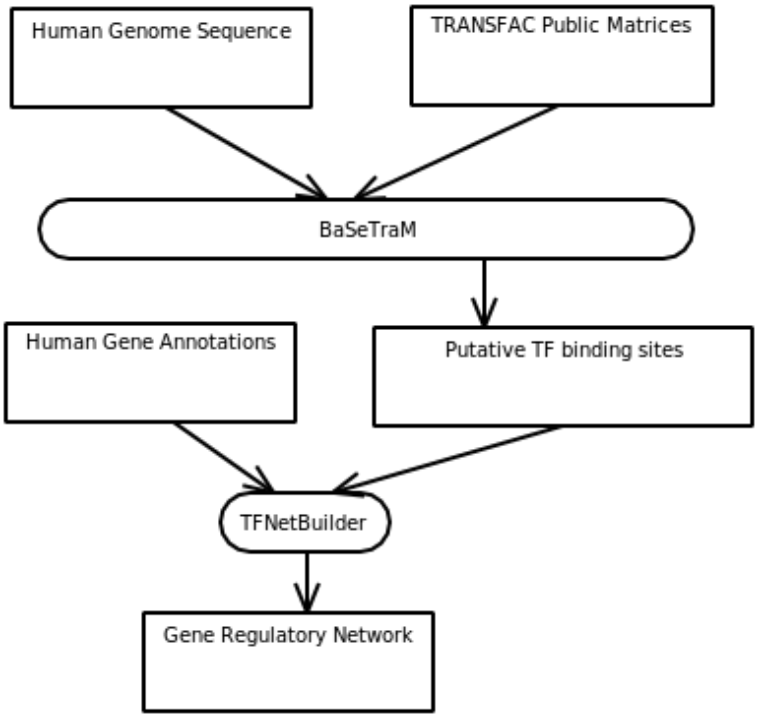

Figure 6.1: The process used to build a GRN with BaSeTraM and TFNetBuilder.

contiguous sequences of DNA from each chromosome, and importantly, annotations describing the putative locations of genes in those contigs), a database of names for genes, and the set of position weight matrices, with annotations describing the names of the genes corresponding to the transcription factors for each matrix.

Linking TF matrix names to genome features

An important part of building a gene regulatory network in this fashion is being able to link a transcriptional regulator to a human gene. This is complicated by the fact that TRANSFAC contains matrices for multiple species, and a gene for a conserved transcription factor may have different names in different species. In addition, when a paralogy event means that the same transcription factor can be produced by more than one gene, TRANSFAC may include or omit a suffix used when annotating the genome; if the paralogous genes have a numeric

suffix to distinguish them, TRANSFAC matrix annotations usually simply omit the suffix.

On the other hand, GFF formatted genome feature annotations for genes describe the location range and direction for a gene, along with cross-references to other databases to describe what the gene is. The most important cross-reference for our purpose is to a gene naming database; for the human genome, the primary naming database is the HGNC (HUGO Gene Nomenclature Committee) database [Bruford et al., 2008]. Names from this database are used as the primary key to identify a gene.

The transcription factor database is linked to a gene identifier by text matching on the gene name; this process is inexact and relies on human input to check for false positives. The following rules were developed by an iterative process of examining non-matching matrices and identifying reasons why the names weren't matching. Initially, the HGNC database is searched for an exact match to the transcription factor name. If that fails, a series of rules are attempted until a gene is found. Any numerical suffix on the TRANSFAC name is dropped, and the search repeated using the prefix. Next, any occurences of -alpha or -beta are replaced with A or B, respectively, to make the name consistent with HGNC naming conventions. Similarly, replacing -1 with I and -2 with II is tried. Next, the full name from TRANSFAC is searched for with the suffix of 1 added; the same is tried with a suffix of A. In addition, the entire process is repeated with all dashes removed from the name.

Identifying regulated genes

The previous subsection describes how to identify the gene identifier corresponding to the transcription factor. However, the next step is to predict which genes are regulated by that transcription factor.

Processing of the BaSeTraM output occurs, one contig at a time. There are generally a number of contigs per chromosome (because the entire chromosome cannot be assembled into a single contiguous sequence, due to repetitive sequences in regions such as the centromeres making complete assembly difficult).

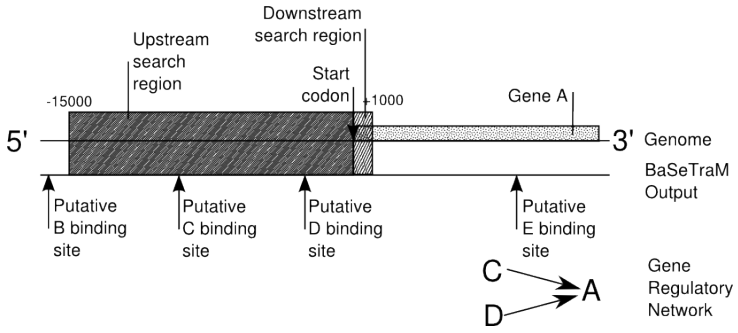

Figure 6.2: An overview of the process by which TFNetBuilder produces a Gene Regulatory Network from the output of BaSeTraM and the human genome sequence.

GenBank sequences are typically distributed with one GFF file per chromosome, with multiple contigs in each file. BaSeTraM output is produced with one directory per chromosome, and with one file in that directory per contig (also in GFF).

The GFF files are opened one chromosome at a time, and parsed using a library developed in the course of this research called ParseGenBank; ParseGenBank is a minimalistic, incremental, event driven GenBank Flat File Format parser written in C++. It aims to efficiently produce events for keywords, features, qualifiers, and coding data. It does not attempt to parse the contents of feature or qualifier data, but provides a framework on which such a system can be built.

The events generated by the ParseGenBank library as contigs and features are processed drive the construction of the Gene Regulatory Network. As gene features are read in, they are stored in one of two lists: forward genes, and reverse genes, depending on the strand on which the gene is present.

After the lists of genes have been stored, they are sorted by the offset of the gene in the contig, where offset is measured from the 5' end in the respective direction corresponding to the gene. The location for each putative transcription factor binding site is checked against the sorted list using a binary search algorithm to find all genes that are within the upstream and downstream zone of the regulator. This is shown in Figure 6.2. For the purposes of this chapter, the region used was 15000 base pairs upstream, and 1000 base pairs downstream.

These numbers were picked to be considerably larger than estimates in yeast from Harbison et al. [2004], to ensure that true edges are unlikely to be missed. When a smaller upstream region of 4000 base pairs was used, the number of edges found was only 1846 (a small network when contrasted to the 29381 gene annotations on the human genome; as discussed below, a total of 6111 edges were found when the upstream search distance was increased to 15000 base pairs).

For every transcription factor that intersects with a gene, an edge is added from the gene corresponding to the transcription factor (the putative regulator) to the intersecting gene (the putative regulated gene). Note that it is possible for a single transcription factor match to intersect with more than one gene.

6.3 Comparing models by fit to data

In Chapter 5, qualitative models of gene regulation were converted into quantitative models using support vector regression, and an iterative procedure was used to predict expression levels for all expression microarrays; the leave-one-out cross-validation error was used to predict which genes were missing from the model.

In this chapter, the intention is instead to validate models, rather than to predict the parts of the model contributing to poor fit. Because the intention is to validate models, it isn't necessary to predict entire arrays as was done in Chapter 5; instead, the strategy used is to compare the actual level of gene expression to the predicted level, where the prediction is made based on expression level measurements of regulators on the same microarray.

As a larger data set is used in this chapter, leave-one-out cross-validation over the full set of arrays is not feasible. Instead, in this chapter, microarrays were divided into three disjoint sets at random; the training set, S_1, used to train the SVMs, the parameter testing set, S_2, used to test the choice of SVMs and to optimise the SVM parameters over, and S_3, the final testing set, used to generate the total residual error for a given model.

Let M be the set of topological models, G be a set of genes, and A be the set of all microarrays. For all $m \in M, g \in G$, let $r_{m,g}$ be the vector of genes regulating gene g according to model m. Let $n_{m,g} = |r_{m,g}|$. For all $a \in A$ and $g \in G$, let $l_{a,g}$ be the expression level reading of gene g on microarray a (with $L_{a,g}$ as the random variable for the distribution it is sampled from). Let $\alpha_{m,g}$ be the support vectors for regulated gene $g \in G$ in model $m \in M$, controlling the position of the hypertube in feature space. Let ν be the parameter in ν-support vector regression describing the proportion of points in the hypertube (in feature space), let γ be the parameter to the kernel for the radial basis function between the input and feature spaces, and let C be the parameter describing the penalty multiplier for values outside of hypertube.

Let $f(\alpha_{m,g}, \nu_{m,g}, \gamma_{m,g}, C_{m,g}, l_{a,r_{m,g,1}}, l_{a,r_{m,g,2}}, ..., l_{a,r_{m,g,n_{m,g}}})$ describe the $\nu - SVR$ test function, and let:

$$\hat{L}_{m,a,g} = f(\alpha_{m,g}, \nu_{m,g}, \gamma_{m,g}, C_{m,g}, l_{a,r_{m,g,1}}, l_{a,r_{m,g,2}}, ..., l_{a,r_{m,g,n_{m,g}}}) \qquad (6.1)$$

be an estimator of $L_{a,g}$.

The ν-SVR functions are trained using the arrays from the training set S_1 to determine $\alpha_{m,g}$ for each model and regulated gene.

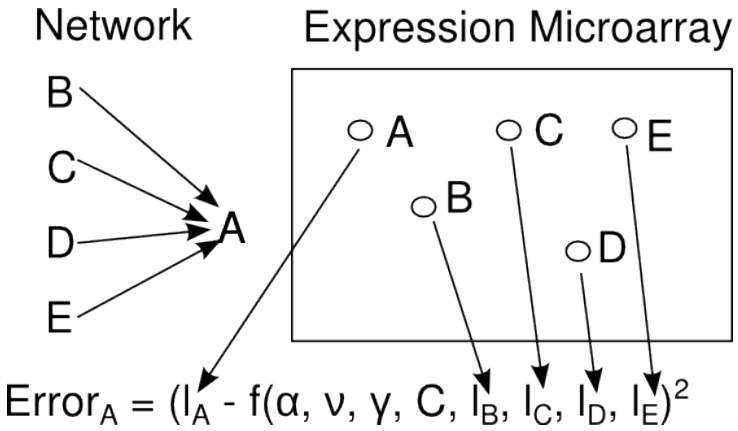

Figure 6.3: The error per microarray and per gene for a model. These errors are added up over all genes and testing microarrays to compute the total error.

The error for array a and gene g from model m is then $e_{m,a,g} = (\hat{L}_{m,a,g} - l_{a,g})^2$. The derivation of this error is shown in Figure 6.3. The total error for an array a in a given model m is $e_{m,a,.} = \sum_{g \in G} e_{m,a,g}$.

Different SVM parameters $\nu_{m,g}$, $\gamma_{m,g}$, and $C_{m,g}$ are used for each regulated gene in a given model. This allows the trade-off between reducing model complexity and improving model fit to be adjusted per-gene to maximise generalisation. This means the relatively expensive parameter finding process needs to be completed for every regulated gene, for every model; parallelisation of parameter finding can be employed to decrease the time required to find parameters for all genes and models. Preliminary work suggested that errors are much higher if one set of parameters is shared for multiple genes and models (as would be expected if explaining the regulation of some genes is harder than for others, so different parameters are required). Parameter finding is carried out using the Covariance Matrix Adaption Evolution Strategy (CMA-ES); this non-linear optimisation method is recommended for SVM parameter tuning by Friedrichs and Igel [2005].

The total error for a model is $e_{m,.,.} = \sum_{a \in S_2} e_{m,a,.}$; the CMA-ES algorithm will attempt to find the values of ν_m, γ_m, and C_m that yield the global minimum value of $e_{m,.,.}$. For each parameter value, the support vectors $\alpha_{m,g}$ are recomputed using the $\nu - SVR$ training function.

Once the parameters have been found, the support vectors are computed for the final parameter estimate, and the errors $e_{m,a,g}$ are computed using the parameters over the final testing set ($a \in S_3$), for all genes $g \in G$.

Two models can be compared using the sign test over the differences between the two models for each model.

Let $\text{pos}(\mathbf{x})$ be the number of positive values in the vector $\mathbf{x}$, and let $\text{neg}(\mathbf{x})$ be the number of negative values. Let ϵ_{m_1,m_2} be the vector over $a \in S_3, g \in G$ of $e_{m_1,a,g} - e_{m_2,a,g}$. Let $x_{m_1,m_2} = \text{pos}(\epsilon_{m_1,m_2})$ and $\lambda_{m_1,m_2,a} = \text{pos}(\epsilon_{m_1,m_2}) + \text{neg}(\epsilon_{m_1,m_2})$. Let p_{m_1,m_2} be the true proportion of tests for which m_1 yields a better prediction than m_2; $\hat{p}_{m_1,m_2} = \frac{x_{m_1,m_2}}{\lambda_{m_1,m_2}}$ is used as an estimator of that true proportion. If the two models are equally good, $p_{m_1,m_2} = 0.5$. This is taken to be the null hypothesis. Under the assumption of independence between each comparison, each comparison where the errors are not equal is a Bernoulli trial, and so x_{m_1,m_2} is a sample from the random variable

$X_{m_1,m_2} \sim \text{Binomial}(p = p_{m_1,m_2}, n = \lambda_{m_1,m_2})$. This allows a p-value to be computed to assess the two-sided probability of seeing a value further from 0.5 than $\hat{p}_{m_1,m_2}$, under the null hypothesis.

6.4 Making inferences about model generation methodologies

Overall, the approach presented in section 6.3 allows inferences to be drawn about how well one model is supported by microarray data compared to another model.

One of the most useful applications of this is that it can be used to draw inferences about the relative effectiveness of methods for generating models of transcriptional regulation.

In this case, model m_1 could be generated by one method, and model m_2 could be generated by another method (perhaps the same as that used to generate m_1, except with an additional source of information taken into account). If the second method is significantly better than the first, then with high probability, $\hat{p}_{m_1,m_2} > 0.5$ and the p-value discussed above will be small (close to zero).

6.5 Testing the method

Microarray data for testing the method was obtained from the GEO (Gene Expression Omnibus) database described in Edgar et al. [2002]. At the time of the download, the expression microarray platform for which the most data was available was the platform with GEO Platform identifier GPL570, the Affymetrix GeneChip Human Genome U133 Plus 2.0 Array. The platform has 54,675 probes (some of which are so-called mismatch probes), for a total of 47,000 human gene transcripts. The manufacturer provides instructions on how to perform the microarray hybridisation, and so the methodologies used should be fairly consistent even between different laboratories. Data for a total of 13,214 different microarrays were downloaded, across a wide variety of cell

types and conditions. The microarray platform includes multiple probes for each gene; the median of these values was used to estimate the level of the gene.

Building a gene regulatory network

To test the method, a gene regulatory network was built using the methodologies described in Chapter 3 and Section 6.2.

All transcription factor binding matrices from TRANSFAC Public were downloaded from `http://www.gene-regulation.com` using a script that requested each matrix by successive identifiers and constructed a FASTA formatted file containing all the matrices.

The human genome sequence [Venter et al., 2001] release 5, build 36 was downloaded from GenBank [Benson et al., 2000] in GFF format. These files include both sequence and feature data.

BaSeTraM, as described in Chapter 3, was used to search all chromosomes of the human genome for any occurrence of a transcription factor corresponding to any matrix in TRANSFAC Public. Due to the high computational cost of this search, a relatively low posterior probability cut-off (of 0.5) was used, as the cut-off can be increased by filtering the output, but can only be decreased by re-running BaSeTraM (however, a cut-off much lower than 0.5 would have produced too much data to easily store). A total of 360,071 transcription factor binding sites were identified with a posterior probability higher than this value.

The output from BaSeTraM was given to TFNetBuilder (see section 6.2) to build a GRN. This network contained 3632 gene vertices (only counting genes that were mentioned in at least one edge as either a regulator or regulated gene), with 6111 edges into 3568 regulated genes from 135 regulator genes.

The process taken to build a model and compute errors is shown in Figure 6.4.

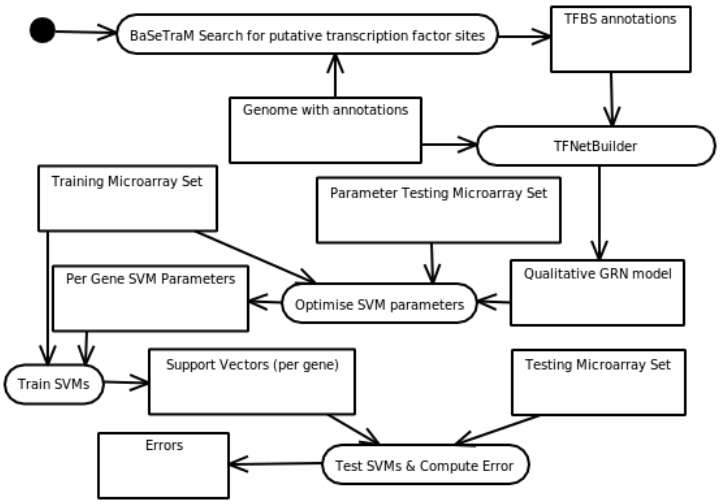

Figure 6.4: A UML Activity diagram showing the overall process used to build and validate a Gene Regulatory Network.

Perturbing the GRN

The method presented in this chapter allows two or more gene regulatory networks to be compared; it is therefore necessary to build a gene regulatory network to compare with the one generated by the process described above.

This was achieved by creating random perturbations of various kinds from the initial gene regulatory network. These perturbations are designed to decrease the quality of the gene regulatory network; however, as discussed later, it appears that paradoxically, these perturbations are more likely to increase the predictive ability of the gene regulatory network than to decrease it.

The following types of perturbations were implemented:

- Label switching: Vertexes are selected independently with probability p (so p is the expected proportion of vertices to be selected). The labels associating the selected vertices with a gene are scrambled. In this chapter, $p = 1$, so all vertex labels were scrambled.

- Edge deleting: Edges are selected independently with probability p and deleted from the model. In this chapter, $p = 0.5$ was used, corresponding

to 50% of edges deleted.

- Edge inserting: The number of edges to add is computed by multiplying p by the current number of edges and truncating to an integer. The regulator is selected by picking a vertex at random, so each vertex has equal probability of being selected. The regulated gene is selected in the same way, except with the constraint that the regulated gene cannot be the same as the regulator. Where the edge inserting perturber would have inserted an edge that is already in the model, that edge is not counted against the total number of edges, and another edge is instead selected. In this chapter, $p = 1$ was used, meaning that the number of edges to be added matched the number of edges already in the model.

Two models were generated for each perturber described above, giving a total of 7 models.

The errors for each of these models were evaluated using the procedure described in Section 6.3. Computation was carried out on a 24 core high performance computer running Xeon CPUs at 2.666 GHz in 64 bit mode; due to other users of the system, not all 24 cores could be used at all times. Code was developed in Haskell to make use of the Concurrent Haskell facilities [Jones et al., 1996] to implement parallelism safely.

6.6 Results and Discussion

Each perturbed model was compared to the original unperturbed model. Paradoxically, all of the perturbed models achieved a statistically significant improvement in fit (lower error) than the original unperturbed model, as shown in Table 6.1.

One explanation for these results is that the initial model is not very accurate. This can explain the results for edge deletion because if more than half of edges are incorrect, deleting an edge is more likely to increase the quality of the model than to decrease it. The decrease in error when edges are inserted or labels are scrambled can be explained if genes that are not regulators are more likely to

Table 6.1: Comparison between different perturbed models and the original model by fit to expression microarray data in human.

Perturbed model	Proportion of gene/array pairs where perturbed model has higher error	Two-sided p-value
50% deletion, Repeat 1	0.414	3.63×10^{-27349}
50% deletion, Repeat 2	0.411	1.45×10^{-28516}
Insertion (100%), Repeat 1	0.499	0.0152
Insertion (100%), Repeat 2	0.495	9.32×10^{-89}
Label Perturbing, Repeat 1	0.486	8.64×10^{-920}
Label Perturbing, Repeat 2	0.482	4.93×10^{-1605}

be predictive of other genes than regulators. This is likely if a few regulators explain most of transcriptional control; adding an edge from a gene regulated by a common regulator to a gene that doesn't already have an edge from that common regulator is likely to improve the fit.

If the above explanation is correct, then it would be expected that there would be two populations of regulated genes; those from which a true edge has been removed, which contribute to increasing the error in the model, and those from which an incorrect edge has been removed, which contribute to decreasing the error in the model. Likewise, the same pattern could be expected with inserting an edge. When scrambling labels, some edges would be expected to have a combination of regulators and regulated genes that fits the data well, while others would be expected to not fit well.

This predicted bimodal distribution of error values was observed in the data. Figures 6.5 and 6.6 show the distribution of the proportion of microarrrays that were predicted better by the model with deleted edges than the original model, over genes. Similarly, Figures 6.7 and 6.8 show the distribution when the number of edges in the model is instead doubled by inserting random edges, and Figures 6.9 and 6.10 show the distribution when the labels on vertices in the model are scrambled.

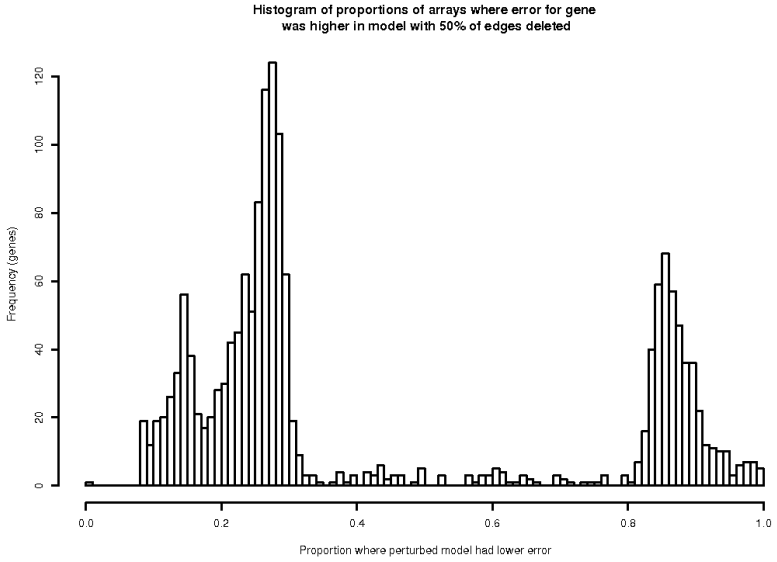

Figure 6.5: The distribution (over genes) of the proportion of microarrays for which the expression level of the gene was predicted more accurately in the model with 50% of all edges deleted, compared to the original model generated by TFNetBuilder (replicate 1).

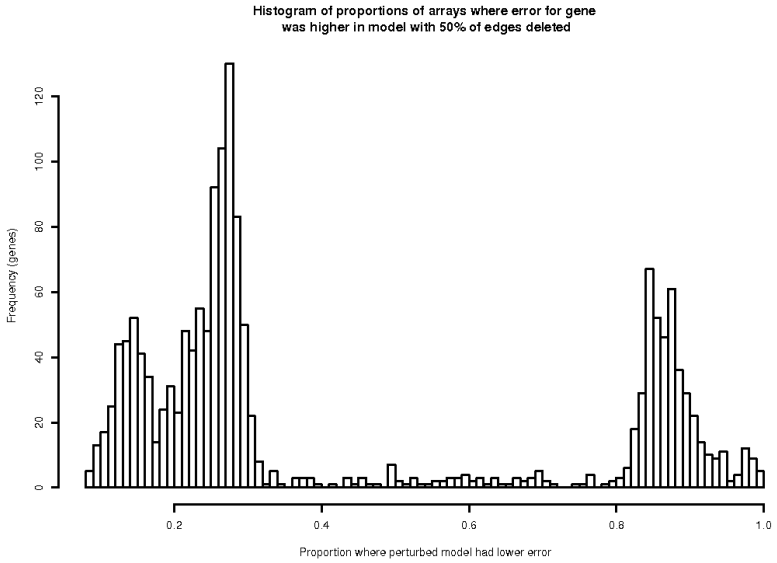

Figure 6.6: The distribution (over genes) of the proportion of microarrays for which the expression level of the gene was predicted more accurately in the model with 50% of all edges deleted, compared to the original model generated by TFNetBuilder (replicate 2).

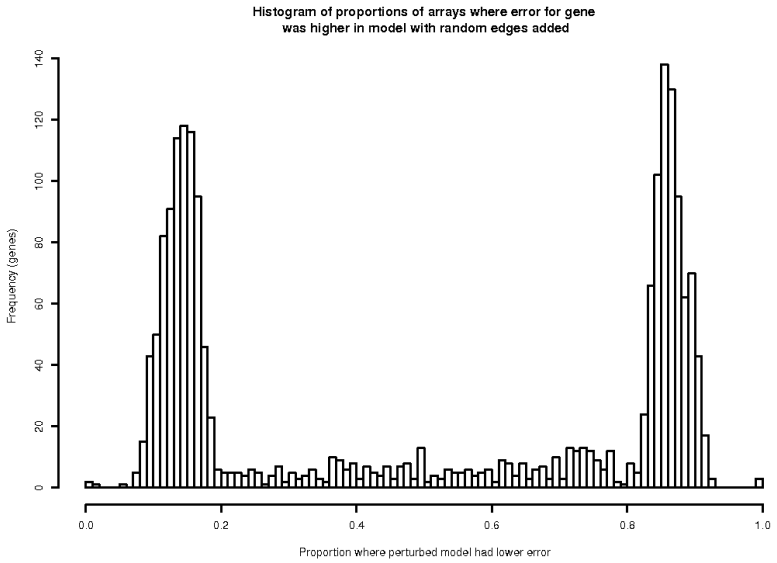

Figure 6.7: The distribution (over genes) of the proportion of microarrays for which the expression level of the gene was predicted more accurately in the model with the number of edges doubled due to random edges added, compared to the original model generated by TFNetBuilder (replicate 1).

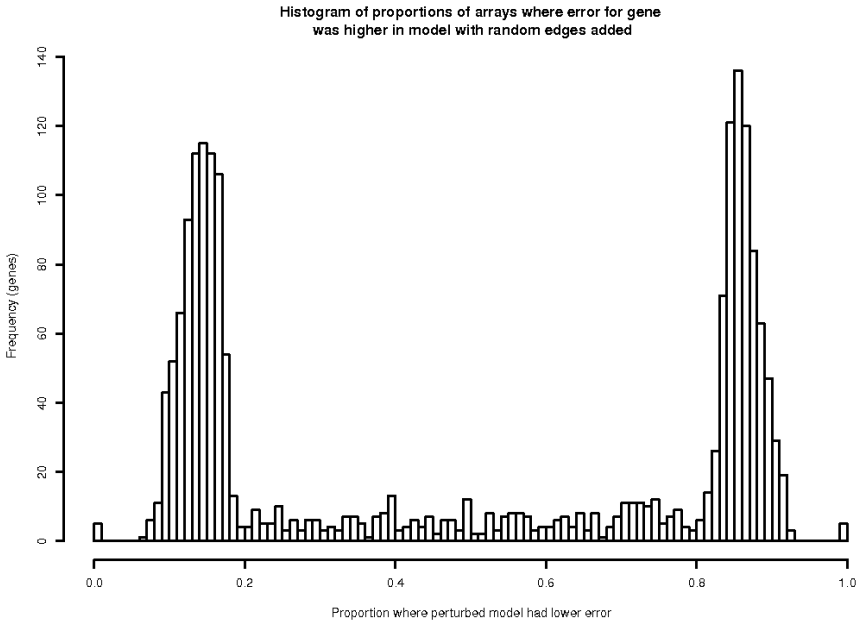

Figure 6.8: The distribution (over genes) of the proportion of microarrays for which the expression level of the gene was predicted more accurately in the model with the number of edges doubled due to random edges added, compared to the original model generated by TFNetBuilder (replicate 2).

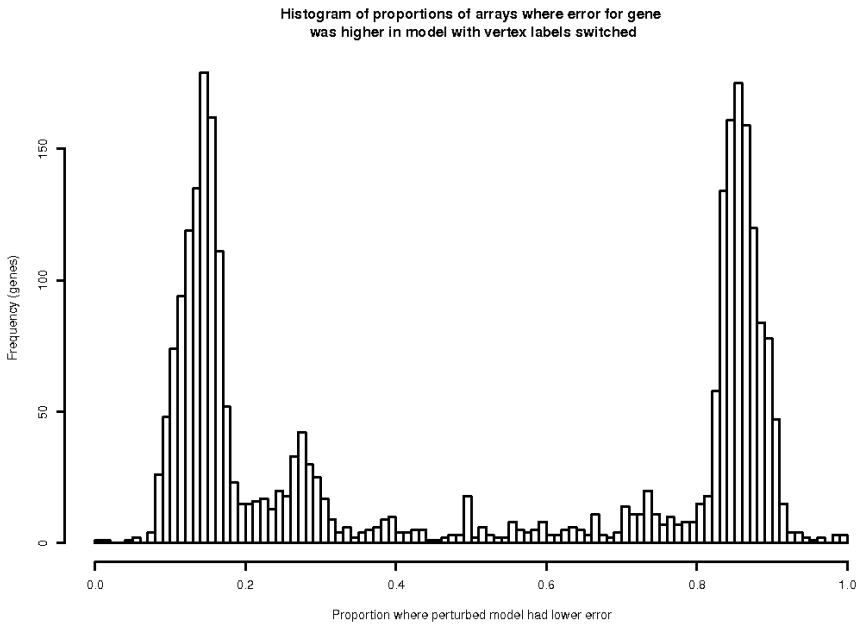

Figure 6.9: The distribution (over genes) of the proportion of microarrays for which the expression level of the gene was predicted more accurately in the model with all gene-vertex associations scrambled, compared to the original model generated by TFNetBuilder (replicate 1).

6.7 Conclusion

In this chapter, I presented a way to compare qualitative gene regulatory network models with data by converting them into quantitative models using support vector machines.

Validating models of gene regulatory networks in human using expression microarray data has been shown to be difficult because the correlations present in expression microarray data mean that there can be multiple incorrect models that fit the data better than a partially correct representation, and so even a model with vertex labels scrambled randomly can predict the data better than a model based on putative transcription factor binding.

There are two clearly separate populations of genes shown in the edge deletion experiments in this chapter; those for which prediction accuracy significantly increased when deleting the edge, and those for which prediction accuracy

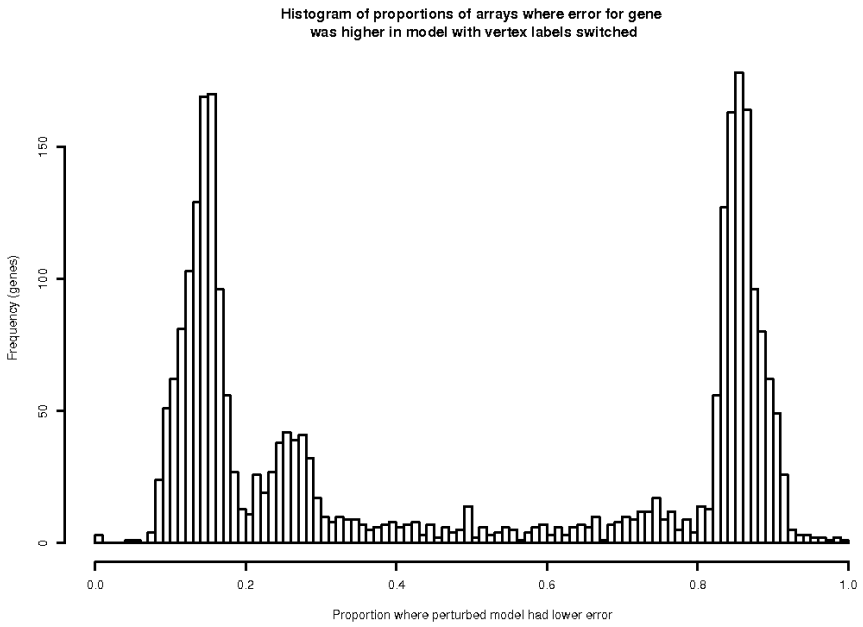

Figure 6.10: The distribution (over genes) of the proportion of microarrays for which the expression level of the gene was predicted more accurately in the model with all gene-vertex associations scrambled, compared to the original model generated by TFNetBuilder (replicate 2).

significantly decreased. This suggests that these techniques for model validation may be significantly more useful for comparing different, more accurate models (for example, models built using more accurate models of transcription factor binding).

Chapter 7

Literature Review: Mathematical Model Representation

7.1 Introduction

A mathematical model is a hypothesis that describes the properties of a system as mathematical relationships between system properties (and may include a description of how system properties vary over independent variables such as space and time).

Mathematical models are useful because they enable modellers to confirm that assumptions about the structure of a system are consistent with each other and with experimental data. Creating a mathematical model around a hypothesis to explain new data, in the context of an existing theoretical framework can therefore be a more reliable way to ensure consistency compared to purely qualitative reasoning. While mathematical models can make more efficient use of experimental data, they are not a substitute for experimental data, and the trade-off between model complexity and model fit discussed in section 4 needs to be taken into account.

A mathematical model that is believed to be an accurate representation of a system can be used to better understand the system and determine the impact that interventions will have on a system. Simulations from mathematical models allow access to variables that may be hidden and difficult to measure in the real system. In addition, experimenting on mathematical models can often bypass

financial, safety, and ethical limitations that may apply to experimentation on the system itself.

Computer simulation using mathematical models is now common practice. For a computer simulation to occur, the mathematical model must be represented in a computer readable form. This section provides background on such representations.

7.2 Representing physical and biological systems

Prior to the advent of widely available access to computer, mathematical models were solved by hand (either analytically or numerically), and communicated as a series of equations on paper.

Analogue computers were used in early simulations of biological systems [Fry, 1960]; the details of these simulations were also communicated primarily as human readable equations on paper.

In some cases, as for the well-known Hodgkin and Huxley [1952] model, both the equations and the simulation procedure were described in the text without reference to the exact technologies used to perform the simulation.

As digital computers became widely available, they were used for computer simulations. For example, Grodins et al. [1967] describes a biological model of the respiratory system. It is described both as a circuit diagram and in terms of equations. The article mentioned that Fortran was used, but aside from through publication on paper, widespread distribution computer code was difficult in 1967.

The use of imperative programming languages, such as Fortran (or more recently, MATLAB), to represent models continues today. Such general purpose languages allow any computable algorithm to be expressed; however, unprincipled use of imperative languages for model representation encourages mixing of the mathematical model with the solver algorithm, and prevents the model from easily being composed with another sub-model. For this reason, standardised

representations of mathematical models is important.

One of the first efforts to standardise mathematical model representation formats was Garfinkel [1968]. The format is a domain-specific language for representing chemical and biochemical reaction systems. The set of systems are represented as a series of lines (one line per punched card; several lines per reaction) of specially formatted text. The authors provided a simulator written in Fortran capable of processing the representation. This effort was further developed in Roman and Garfinkel [1978] and KINSIM [Barshop et al., 1983].

A major development in mathematical model representation was the introduction of object-oriented modelling. In object-oriented modelling, models are composed out of objects, each of which describes a number of mathematical equations, with interfaces that can be connected to describe how the parts of the model interact. Modelica [Mattsson et al., 1998] is a standard for exchanging object-oriented models that has seen widespread adoption in certain industries. In Modelica, models are expressed in terms of classes; classes define the structure of objects (that is, objects are specific instances, while classes describe a type of thing). Classes can inherit from other classes and add additional properties and equations. Classes can contain equations, and in addition, connectors can define links between different objects. Modelica additionally provides a facility (called a stream connector) specifically for the problem domain of dealing with processes that involve the movement of some quantity between objects, such as pressure dependent flow of fluids.

JSIM [Li et al., 2000] introduced a modelling format for mathematical equations, called MML. MML is an ASCII-text based direct description of the model, including the equations and units on the model.

CellML 1.0 [Hedley et al., 2001] is a language for representing mathematical models in an XML-based format. It divides the model up into components, each of which can contain equations (represented in content MathML [Carlisle et al., 2001]). Equations are written in terms of variables declared for the component; variables in different components can be connected, meaning that two variables are the same, even if they have different names in different

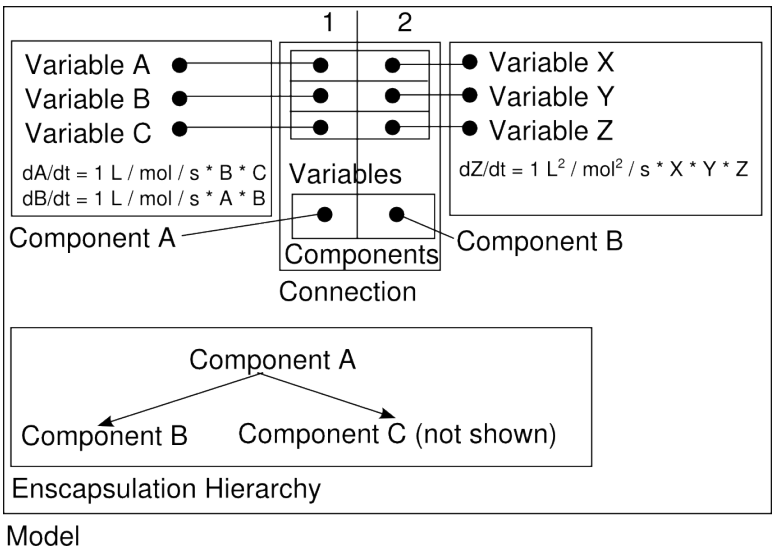

Figure 7.1: A diagram of a hypothetical CellML model, showing how an encapsulation hierarchy of components can be created, and variables can be connected.

components. In addition, CellML models have physical units specified on variables and constants; variables can only be connected if they are dimensionally consistent. If variables are dimensionally consistent but in different units, tools are expected to insert the appropriate conversion factor. Components can be grouped into an encapsulation hierarchy, with the external details that can be connected to other components at the top of the hierarchy. Figure 7.1 shows the relationships between variables, components, and encapsulation for a simple model.

CellML 1.1 [Lloyd et al., 2004] introduced imports, which allow components (including the components encapsulated underneath those components) from one model to be included into another; this allows modular files to describe parts of models for later composition.

Around the same time, a separate standardisation effort, SBML (Systems Biology Markup Language) Level 1 was also being developed [Hucka et al., 2001]. Like CellML, SBML is based on XML; however Level 1 requires that reaction rates be defined in a custom format rather than MathML. SBML Level 1 is similar in scope to KINSIM in that networks of reacting substances are

directly described. However, the way it is structured means more machine processable information about reactants and reactions can be provided.

SBML Level 2 Finney and Hucka [2003] allows rates and other mathematical expressions to be defined using MathML. Amongst other changes from level 1, it additionally allows equations, as well as just reaction rates, to be specified, along with delays and events. This means it can potentially be used for a wider range of models. SBML Level 3 further extends Level 2, and provides a base specification with extension modules for additional functionality.

The markup language used by Virtual Cell [Moraru et al., 2008], called Virtual Cell Markup Language, is another example of a language that describes models in biological terms (such as compounds, membranes, and reactions); the tool then solves the model from the domain-specific view.

The text and XML based model representations presented above fix a particular representation for mathematical models or reactions as part of the language, so that all tools need to understand the entire language to be able to represent such models. The disadvantage of this approach is that models either need to build a model in very generic terms (for example, as a system of differential-algebraic equations), or use a tool that is built to support a more specific representation for that domain. Modellers wanting to work on a new problem domain or to combine two different problem domains not already supported in the same tool are forced to either develop new tools or use the generic representation.

This problem is addressed by ModML, presented in Chapter 9. ModML represents mathematical models as functional transformation that computes a mathematical model; this transformation means that the model can include a standard translation from a domain specific language (DSL) to a generic description as a system of differential-algebraic equations. For example, Chapter 10 describes a module that can be included in a model to check physical units, and then another module built on top of that for describing systems of reactions.

7.3 Metadata

An important aspect of model representation is the ability to represent metadata about the model.

The line between the computational model representation and metadata about a model is somewhat blurry, and depends on the model representation technologies employed; generally, if a model cannot be simulated without reference to some information, and the information is believed to be an intrinsic part of the structure of the system, that information is considered part of the model. For example, in CellML, information about the reactions associated with a given equation would be considered metadata because the information is not needed to run a numerical simulation of the model, only to interpret the results of the simulation. In SBML, however, the reaction information is considered part of the model representation, because it is used to determine the underlying equations.

Metadata is useful because it provides machine-readable information about the physical and biological context of models, sub-models, equations and variables. Human readable variable names and comments on a model are a basic form of metadata; however, more rigid computer readable metadata can enable applications like accurate searches for models and automated model composition [Matos et al., 2010].

As discussed in Beard et al. [2009], CellML metadata is often represented using RDF/XML; the underlying data model is called the Resource Description Framework (RDF) [Klyne and Carroll, 2006]. RDF makes assertions as triples, each of which consists of a subject, a predicate, and an object. The subject and predicate are resources (discussed later), while the object is a node, and can be either a resource or a literal, which contains either human or machine readable information. There are two types of resource in RDF: Uniform Resource Identifiers (URIs) and blank nodes, anonymous resources which exist to form parts of data structures without a name. When annotating a model, each part of the model is given a URI. Technologies such as OWL [McGuinness et al., 2004] build on top of RDF to allow formal ontologies to be defined.

An important aspect of providing biological information about a model is a well-known terminology for a specific entity; otherwise, it will be difficult to determine that, say, two variables in two different models actually refer to the concentration of the same protein. The simplest approach to the problem is to use a controlled vocabulary; an agreed upon list of terms. A more general and useful approach is to annotate against an ontology, a set of terms and relationships between them. Ontologies are available for many different types of entities.

The SBO, Systems Biology Ontology [Le Novère, 2006] describes 'systems biology representations'. There are seven base terms for a systems biology representation in SBO: mathematical expression, metadata representation, modelling framework, occurring entity representation, participant role, physical entity representation, and systems description parameter. For example, there is a term for a concentration conservation law, which is classified as a mass conservation law, which is in turn a conservation law, which is in turn a mathematical expression. Another widely used ontology for model annotation is GO, the Gene Ontology [Ashburner et al., 2000].

Often, it is useful to refer to an entry in a database, such as the UniProt protein sequence database, and use the set of database identifiers as a controlled vocabulary. There are, however, often many possible URIs that would refer to the same entity. The MIRIAM Resources effort [Laibe and Le Novère, 2007] is a database and associated web service. The aim of the effort is for there to be a single canonical URI for each resource. MIRIAM URI schemes can be registered for new databases, and the web service can be used to resolve a MIRIAM URI to the database containing information about the URI.

ModML takes a slightly different approach to metadata, as discussed in more detail in Chapter 9. Rather than expressing a model in a general form and then annotating it with domain specific information, modellers using ModML are encouraged to write the model in a domain specific form, and import a function into the model to convert the domain specific model into a general model. This means that domain specific information is already being provided by expressing the model in a domain specific form. ModML allows metadata, in the form

of triples, to be computed alongside the model. The transform from domain specific form to general form is therefore able to generate the annotations at the same time, avoiding the repetition of information between metadata and model. As with SBML and CellML, these triples could be used to link part of a ModML model to an ontology term.

The proliferation of several different metadata standards describing how to annotate models has created a need for annotation format independent standards about what information should be provided, regardless of the format. The Minimum Information Requested in the Annotation of Biochemical Models (MIRIAM) standard [Le Novère et al., 2005] suggests what information should always be provided with published biochemical models.

Chapter 8

The CellML API and its implementation

Abstract

CellML is an XML based language for representing mathematical models, in a form which is suitable for the exchange of models between different authors, and for archiving models in a model repository.

Allowing for the exchange and archival of mathematical models of biological systems in a computer readable form is a key strategic goal in bioinformatics, because of the associated improvements in the accuracy of the scientific record, the faster iterative process of scientific development when any modeller can continue the work of another, and the ability to build integrative models which are larger than any individual group has the resources to develop on their own.

However, for CellML models to be useful, there need to be tools which can process CellML models correctly. Due to some of the more complex features present in CellML models, such as imports, developing code *ab initio* to correctly process CellML models can be an onerous task. For this reason, there is a clear and pressing need for an application programming interface, and a good implementation of that interface, upon which tools can base their support for CellML.

An application programming interface which allows the information in CellML models to be retrieved and / or modified has been developed. I have also developed a series of optional extension APIs, for tasks such as simplifying CellML connection handling, dealing with units, validating models, and translating models into different procedural languages. Other

modules have been developed by other people; this chapter focuses on the work that was done in the course of my doctoral studies. This chapter is based on the initial draft of a manuscript for a paper before other authors suggested changes, with descriptions of extension modules primarily developed by other people removed; for a more complete description of the API, readers are referred to the published paper [Miller et al., 2010a].

The extensions discussed are the annotation tools service, the CellML Variable Association Service, the CellML Units Simplification and Expansion Service, the Validation Against CellML Specification Service, the MathML Language Expression Service, the CellML Code Generation Service, and the CellML Integration Service.

The API also provides an implementation of this application programming interface, optimised to achieve good performance.

Tools have been developed using the API which are mature enough for production use. The API has the potential to accelerate the development of additional tools capable of processing CellML, and ultimately, lead to an increased level of sharing of mathematical model descriptions.

8.1 Background

One particularly common and useful form of mathematical model is the form of systems of differential algebraic equations (DAEs) [Gear, 1971]. These systems are of the general form

$$F(\mathbf{x}, t, \mathbf{x}') = 0$$

DAE systems are often broken up into individual equations, each of which hold true. Systems of differential algebraic equations are used to model a wide variety of different biological processes, across a huge diversity of scales. For example, at one extreme there are models describing the action of ion channels [Pandit et al., 2001], and at another extreme, models of predator-prey dynamics [Volterra, 1928].

Historically, models of differential algebraic equations have been exchanged and

archived by publishing each of the equations in a scientific paper. Someone wanting to take the published model then needs to convert from the equations in the scientific paper back into a computer program to compute results. This entire process is both time-consuming and error prone. Reviewers are unlikely to invest the time to check that the model published is accurate. Likewise, it becomes prohibitively expensive to do integrative biology [Hunter and Nielsen, 2005b], as building a large model out of several pieces then requires significant effort on each of the pieces already in the literature.

CellML [Garny et al., 2008] is an XML [Bray et al., 2000] based format for representing mathematical models, capable of representing DAE systems (as well as other mathematical relationships). As such, it provides an ideal format for the exchange and archival of models. There are public databases containing large numbers of CellML models, such as the CellML Model Repository [Lloyd et al., 2008]. The BioModels database [Le Novere et al., 2006] also provides CellML models (which are translated from SBML to CellML).

However, for the scientific advantages of using CellML for mathematical model exchange to be fully realised, it is important that software used by modellers is able to read and write CellML models. It is also important that the bioinformatics community has the ability to easily develop software which does new things with the existing databases of CellML models.

Using an application programming interface (API) simplifies the task of processing an XML language, making them important to the exchange of information. Systems Biology Markup Language (SBML) [Hucka et al., 2003] is another XML-based format, used for encoding computational models of biochemical reaction networks. CellML differs from SBML because CellML avoids domain-specific elements, while SBML requires that implementations support elements specific to biochemical reaction networks (although SBML also provides more general mechanisms for representing models). There is an API for processing SBML models, known as libSBML [Bornstein et al., 2008]. The CellML API serves an analogous purpose to libSBML, except for CellML rather than SBML models.

Supporting CellML correctly can be a difficult task, due to some of the more complex features in the CellML language. It is therefore important that software developers do not need to re-invent the same functionality every time they support a new tool. To avoid the need for this, I present both an API for working with CellML models, and an efficient implementation of that API.

8.2 Implementation

The CellML API is a platform and programming language agnostic description of interfaces, with attributes and operations on the interfaces. These attributes and operations are used to retrieve information about the model, or alternatively to manipulate the model in memory.

The API is specified using OMG IDL [Siegel, 1998], and is made available under an Open Source / Free license, at `http://www.cellml.org/tools/api/`. It is suitable for both CellML 1.0 and CellML 1.1 documents. All attributes and operations in the IDL files are documented in place using the Doxygen comment format [Doxygen developers]. The choice of this programming language agnostic format to specify interfaces makes it possible to define bindings to the API from many different programming languages. I have developed bindings for C++ [Koenig], Java [Gosling et al., 2005], and Javascript (via XPCOM) [Stearn, 2007].

In addition, I have developed a high-performance implementation of the API. This implementation is written in C++, and is based on the libxml2 XML parsing library [Veillard], and an implementation of the W3C DOM [Wood, 1999] and MathML DOM [Carlisle et al., 2001]. Due to the language bindings and bridges provided, this API can be accessed from C++, Java, Javascript, and from an even wider range of languages over CORBA. Work is underway to allow access from Python [Lutz, 2006] without going through CORBA.

The interfaces defined in the API all use the inheritance capabilities of OMG IDL to derive from a base interface, called IObject. IObject is modelled after the similarly named interface in the XPCOM and COM object models. The

IObject interface is used to provide interfaces for basic common facilities relating to the object underlying the interface, such as maintaining the reference count (as discussed later), and providing a unique identifier for each object. This unique identifier is useful for determining if two interface references describe the same object, and for building data structures which require that objects can be compared.

API implementations use reference counting [Bevan, 1987] to determine when there are no remaining references to a particular object. The IObject interface has operations for incrementing and decrementing the reference count. In order to ensure reference counting works correctly, a few simple rules are followed consistently throughout the API design (and the API design relies on the same rules being followed by code which deals with the API). All operations and attributes which provide an interface reference also increase the reference count of the underlying object. For example, in the case where the operation creates a new object, but no internal references to the new object are kept, the reference count of the returned interface should be one. Secondly, for every time code increments the reference count by invoking the add_ref operation (or by obtaining a returned interface), it must ensure that eventually, the reference count is decremented by invoking the release_ref operation. The API is designed to be accessed through wrappers, and so the actual storage of the object may even reside on a different machine to the wrapper providing the interface being used. For this reason, the third rule arises: implementations should always match an add_ref invocation with a release_ref invocation on the exact same interface pointer (as opposed to a different interface pointer for the same object, which may point to a different wrapper around the same object).

It is worth noting that while IObject provides facilities for reference counting, many programming languages perform automatic garbage collection using either implicit reference counting, or variations of the mark-and-sweep algorithm. When using a direct bridge to these languages, the wrapper code will automatically call add_ref and release_ref on behalf of the user, and so the need for explicit memory management is avoided. For example, the Java bridge makes use of the finalisation facilities in Java, combined with the memory management

facilities provided by the Java Native Interface, so that Java users do not need to explicitly modify reference counts.

In addition to the reference counting scheme, the IObject interface also provides a QueryInterface operation. This operation is used to ask an object if it supports a particular interface, and if it does, to provide an interface representation. As discussed earlier, the API is often accessed through wrapper code, and so users of the API should always perform QueryInterface operations on API interfaces, rather than directly using the language-specific casting mechanisms in their language of choice.

8.3 Results

The facilities for information retrieval in the API are closely aligned to the arrangement of XML elements in a CellML document. There is one object for each CellML element in the document. These objects implement an interface, which is specific to the type of the CellML element. These elements' interfaces all inherit (directly or indirectly) from the CellMLElement interface. This interface provides functionality which is useful on all elements. For example, it provides the ability to insert and name all child elements of the element concerned, and to set user data annotations, identified by a unique key, against the elements.

The interfaces for CellML elements which have a mandatory name attribute all inherit from the NamedCellMLElement interface. This interface provides a name attribute (which can be fetched or set), and inherits from the CellMLElement interface.

Each interface for a specific type of CellML element has, for each type of child CellML element allowed by the CellML specification, a read-only attribute for retrieving all the CellMLElements of that type. The returned list implements a list interface specific to the type of element expected.

These specific types of list follow an inheritance hierarchy parallel to those of the element objects in the list. Each list interface has a corresponding

iterator interface, which allows each object to be fetched in sequential order. Because the iterator interface is specific to the object being fetched, the required interface is returned, avoiding the need to call QueryInterface. However, it is also possible to use the less specific (ancestor) list interfaces to retrieve a less specific (ancestor) iterator object (for example, for use in generic code which works on more than one type of element).

Iterators derived from NamedCellMLElementIterator also provide interfaces for fetching elements by name. All descendant iterator interfaces provide more specific fetch by name operations.

List interfaces also provide facilities for modifying the relevant lists by inserting CellML elements. Because order is not important to the meaning of the model, the iteration and insertion facilities provide no control over the actual order of the elements in the model.

Dealing with model imports in CellML 1.1

Because CellML 1.1 provides for components and units to be imported into models from other models, it was necessary to provide facilities in the API so that such information can be accessed.

The result of supporting CellML 1.1 is that processing one mathematical model can require that more than one XML file be examined. To deal with this issue, the API introduces the concept of instantiated models. An imported model is instantiated once it has been loaded. When all imports required for a mathematical model have been loaded (including models which are imported by an imported model), the model is said to be fully instantiated.

In addition to providing facilities for accessing the list of imported models, their URI, and the components and units they import, the CellML API allows for particular imports to be instantiated, or for the model to be made fully instantiated. It also provides access to the model element of the imported model from the corresponding import (once imports have been instantiated).

To make it possible to more easily find the list of components or units in the

complete model, taking into account the presence of imports in the model, I have included three separate attributes for lists of components in the model, with three corresponding lists of units.

The local component list contains only components in the particular CellML file (but not including imported components). The model list contains all components which are in the local list, and also the import component elements in the same file. The full lists contain all components in the model, across all files making up the model. Where the model containing a component is uninstantiated, the import component is provided by iterators.

When a model is instantiated, the components / units in the imported model are returned by the iterators, and in addition, these models are examined to identify further imported models to search for components or units, as appropriate.

The three corresponding lists of units follow the exact same semantics as the lists of components, except over units rather than components.

Extension APIs

In addition to the core APIs, I have also produced APIs to provide services which are beyond the scope of the core API (which is the basic manipulation of, and access to, the content of CellML models). The relationships between these APIs is shown in Figure 8.1.

The core API does not depend upon the extensions, and so individual API implementations can choose not to support all extension APIs. However, all extensions depend upon the core API, and some extensions also depend on other extensions.

The Annotation Tools

The core CellML API provides basic facilities for in-memory annotation of elements in the CellML model with arbitrary user-supplied objects. However, the user data annotations are difficult to use for some applications, because the

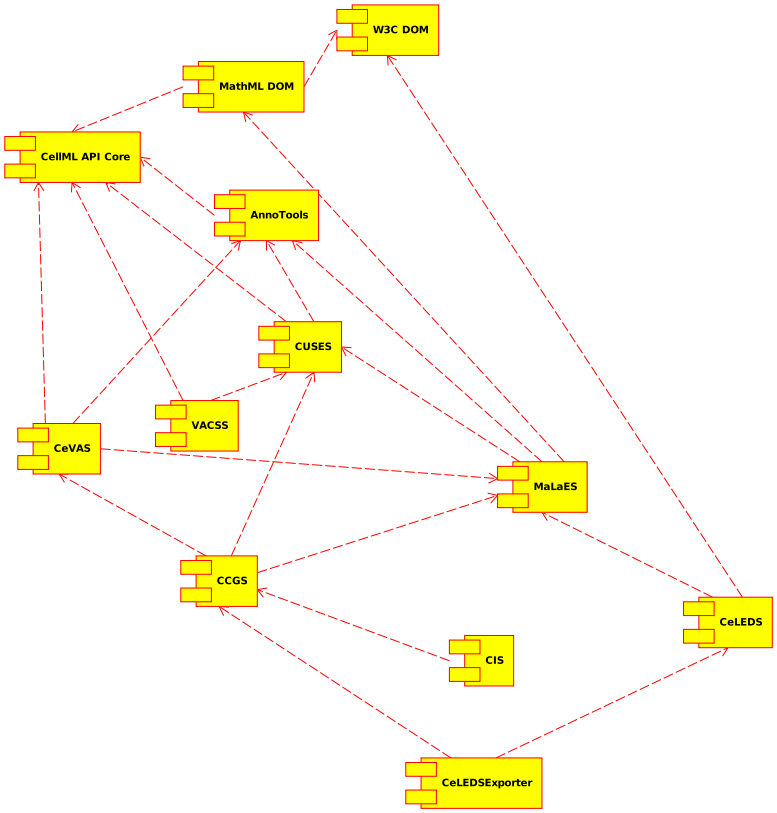

Figure 8.1: A UML package diagram showing the relationships between the CellML Extension APIs

core API requires that user data be associated with a key and then removed when the application has finished with it.

The AnnotationTools (AnnoTools) API provides the ability to allocate and release a set of annotations, without needing to worry about interfering with other annotations being placed by independent calls to the same code, or about needing to individually remove all annotations left on objects.

AnnoTools implementations generate a unique prefix for each AnnotationSet, and allow the user to set annotations with that prefix. They keep an internal list of all annotations which were added, and clear all annotations in the AnnotationSet when the AnnotationSet is destroyed.

The AnnoTools API also includes facilities for more easily setting and retrieving

string, integral, and floating point annotations.

The CellML Variable Association Service

The CellML Variable Association Service (CeVAS) provides facilities for simplifying the process of treating interconnected CellML variables, across many different components, some of which may be imported from different models, as the same mathematical variable (albeit possibly in different units).

CellML 1.0 and 1.1 require that variables which are connected to variables in other components have a public or private interface value of in or out. Whether the public or private interface applies depends on the encapsulation relationship between the components. In CellML, all 'in' interfaces must be connected to an out interface, encapsulation is always acyclic, and valid CellML models have a finite number of variable elements. This means that there is always a variable in each connected network of variables that has no in interfaces. This variable is called the source variable, and is used by CeVAS as a representative of all variables connected (directly or indirectly) to it.

The interface allows users to supply a CellML Model interface, and pre-compute which variables are connected. All variables connected to a particular variable can be iterated, and the source variable can be retrieved.

This is implemented using an efficient disjoint sets algorithm, which allows for inverse Ackerman amortised time merges of sets [Tarjan, 1975]. Initially, every variable in the model is treated as a set of size 1. The algorithm iteratively processes all connections in the model, merging the disjoint sets associated with each of the two connected variables. Therefore, the amortised time complexity of processing a model with n components and m connections is in $O(n\alpha^{-1}(m, n))$.

The CellML Units Simplification and Expansion Service

The CellML Units Simplification and Expansion Service (CUSES) interface provides facilities for processing units in a CellML model.

CellML has a set of built-in units (those defined in the SI specification [Taylor and Thompson, 2001]). These units are defined in terms of the SI base units; metre, second, kilogram, ampere, and mole. Other pre-defined units are defined in terms of these. For example, the Joule is defined as $kg.m^2.s^{-1}$. In addition, the modeller can define their own derived units, for example mmol/L for concentrations, or a new base unit. However, when processing models, it is important to know what the relationship between connected variables is, so the appropriate conversions can be performed. For example, when a variable in metres is connected to a variable in millimetres, tools are expected to insert an implicit conversion factor, so the same variable is compatible across the two components. CUSES allows tools to implement this more simply.

All units are firstly expanded out to be expressions in terms of the base units. Prefixes are connected to multipliers, and multipliers are taken to the power of the exponent, and combined into a single multiplier (the product of multipliers to their exponent). Exponents on the same base unit are added up, so there is only on instance of each base unit, and base units with a final exponent of zero are eliminated. These are exposed to users of the API in an enumerable list of base unit instances.

Facilities are provided to enquire whether two units are compatible. This is useful for determining if a connection is valid. The necessary offset and multiplier needed to perform a conversion can also be obtained.

The Validation Against CellML Specification Service

The Validation Against CellML Specification Service (VACSS) takes as input files which are putatively CellML files, and identifies whether or not the CellML is valid, and where the file is not valid, it attempts to build a list of the problems with the file. Errors which can be detected fall into two types, representational and semantic errors. Representational errors are errors relating to the encoding of CellML in XML, such as essential elements or attributes which are missing, or illegal extraneous elements.

Semantic errors are higher-level errors, where the basic elements of the CellML

are in the correct form, but there are inconsistencies, such as references to names which are required to exist but don't, or violations of any of the numerous rules specified in the CellML Specification. Semantic warnings, such as about potential units problems in mathematical equations, due to dimensional inconsistency, are also made available.

MathML Language Expression Service

One task which is common to many applications is to convert the fragments of MathML which are embedded in CellML documents into fragments of text in some other linear text-based representation.

The MathML Language to Expression Service (MaLaES) provides functionality to assist with this task. MaLaES makes use of CeVAS in order to identify the source variable corresponding to each MathML ci element (*i.e.* reference to a variable by identifier). It then makes use of AnnoTools to retrieve an annotation (which the user can set) containing the symbol to be used for that variable in the output.

It is often the case that these transformations need to take units into account, to ensure that all variables in the MathML contain any necessary conversion factors. To allow for this, MaLaES allows variables (referenced by ci) to be converted into the units of the source variable, and also for the result of an expression to be converted.

In order to allow for conversion into many different languages to occur, MaLaES uses a specification in a custom format called MAL. The MAL description describes the mapping between MathML elements and their forms in the output text-based representation (allowing for pre-order, in-order, or post-order style outputs, with arbitrary separators between arguments), as well as describing the precedence of each operation, what strings are used to begin and end groupings of low precedence operators inside a higher precedence operator, and the format of conversions. The MAL is precompiled into an efficient in-memory representation, which can then be used to generate output.

The CellML Code Generation Service

Another common task is to convert an entire CellML model into code in a procedural programming language, capable of solving the model. The CellML Code Generation Service (CCGS) simplifies this task. Users of the API obtain the CodeGeneratorBootstrap interface pointer through the language specific bootstrap process, and then use the createCodeGenerator operation to obtain a CodeGenerator interface.

On this CodeGenerator interface, it is possible to specify a wide range of different attributes about the language to be generated. This means that code can be generated for nearly every procedural programming language in existence (in some cases, with a requirement for some post-processing to fold long lines or perform similar transformations).

Because CCGS relies upon MaLaES to translate individual mathematical expressions into the correct text-based form, the user also needs to supply a MAL description for the language of interest.

CCGS uses the terminology computation target (represented by a ComputationTarget interface pointer) to represent anything which is required to be computed to evaluate the equations in a CellML model (including those with a constant value, in which case computation is merely assignment to that value). There is not a one-to-one relationship between variables in the CellML model and computation targets. For example, there may be a variable called x, with an initial value of 0, and then an equation such as $\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{x}{t}$. In this case, x and $\frac{\mathrm{d}x}{\mathrm{d}t}$ are both computation targets (t, the independent variable, is also treated as a computation target for consistency). Note that when a variable is used in several components, but the variables are connected together (making them the same mathematical variable), there will only be one computation target for all the variable elements.

CCGS gives every computation target a degree (degree zero means that it is the original variable, degree one means it is the first derivative of the variable, degree 2 means it is the second derivative, and so on). All computation targets which

have a corresponding computation target of higher degree are treated as being state variables, while computation targets with a lower degree computation target are treated as being rates. Computation targets which have both a higher and lower degree computation target are in the unique position of being both a rate and state variable (and the CCGS will generate code especially to handle this case).

Variables which are not state variables, but have an initial value are treated as constants. Anything which is computable using only constants are in turn classified as constants (with this process continuing until completion). It is possible that a system of simultaneous equations may need to be solved in order to determine n otherwise unknown constants from n equations (but when it is possible to split this into small subsystems, it is more efficient to do so). This is done in the implementation using a heuristic algorithm which guarantees that the smallest possible systems are found when the largest indivisible system needed to be solved has at most three equations with three unknowns, and has given good results in testing.

After identifying the order in which equations or systems need to be solved, code is generated for them. This is done using one of three different patterns supplied to CCGS by the application. Where CCGS needs to compute a computation target y using an equation like $y = f(x_1, x_2, ..., x_n)$, CCGS will use the assignment pattern to directly assign into the symbol for y. In other cases, the equation might be in the form $x_1 = f(y, x_2, x_3, ..., x_n)$, in which case the univariate solve pattern is used to compute y. Finally, for systems of equations, the multivariate solve pattern is used.

Any computation targets which are not constants, states, nor are rates, are classified as being 'algebraic computation targets'. In the same fashion as is done for constants, CCGS works out a directed acyclic graph for the order in which systems or equations are used to work out the rate and algebraic computation targets, using the constants, states, and the independent variables. However, these computations are split into two code fragments. The first code fragment contains all computations necessary to compute the rates from the states, constants, and independent variables, while the second code fragment

computes any remaining algebraic computation targets not computed in the first code fragment. This separation allows for more efficient processing of models, because at many time steps, the integrator may not want to report back any results, and so there is no need to evaluate computation targets that aren't required to compute the next time step.

CCGS has the capability to automatically assign indices into four different arrays, referred to as the constants, states, algebraic, and rates arrays. The constants array stores the values of any computation targets which do not depend on the independent variable, or upon any of the rate or state computation targets. The states array is used to store the values of each state computation target. The rates array is used to store the values of the rate of change corresponding to each state computation target. The CodeGenerator object allows the first index to be assigned in each array to be set (for example, to be 0 in languages like C where array indices start at 0, and 1 in other languages like MATLAB). In addition, the user can supply a pattern, for example STATES[%], to describe how the arrays are dereferenced in the output programming language. The caller can also supply their own AnnoSet object, and explicitly provide a name for each computation target if this is required.

Overall, four different code fragments are available. Firstly, the fragment to initialise constants, as discussed above. Secondly, the fragment to compute the rates (and all algebraic computation targets needed to compute these rates). Thirdly, the fragment for the remaining variables. The final code fragment contains any functions which needed to be generated (using a pattern supplied to the CCGS) in order to evaluate the code. These functions can be requested in univariate and multivariate solver patterns, and also in MAL specifications, such as those for evaluating definite integrals.

As CCGS processes models, it will also check for and report back on certain error conditions, such as models which have extraneous equations (reported as being overconstrained), or models which have too few equations to compute all computation targets (i.e. underconstrained models).

The CellML Integration Service

The CIS (CellML Integration Service) provides an interface for performing simulations of models, and receiving asynchronous notifications as results become available. CellML Model interface pointers are given to CIS, which then creates a CellMLCompiledModel object.

The application then specifies the algorithm to be used, and the parameters of the simulation (such as error tolerances, maximum step sizes, as well as parameters controlling which points are reported back). The application may also choose to override an initial value without recompiling the model.

The IntegrationProgressObserver interface is implemented by the application, and given to the CellMLIntegrationRun interface prior to starting the simulation. This interface receives information about the values of constants which were computed, as well as the results from each time-step, and an indication of whether the integration has succeeded or failed (with an error message in the latter case).

The implementation of the CellML API internally makes use of the CellML Code Generation Service to generate C code. The C code is then compiled using a compiler. For example, in one application based on the CellML API, a stripped down version of the C compiler from the popular Free / Open Source GNU Compiler Collection (gcc) is bundled with the application. The code is then linked into a shared object and dynamically loaded into the CIS implementation, which then uses a separate program thread to simulate the model (using either a solver from the SUNDIALS CVODE project, or a solver from the GNU Scientific Library, depending on the algorithm requested).

The CellML API implementation is at the point at which it is stable enough for widespread use. It is already used extensively by the OpenCell environment (formerly known as PCEnv), which provides support for viewing, editing, and running simulations from CellML models.

In addition, third-party users have applied the API to process CellML models and carry out simulations and post-processing [Nickerson and Buist, 2008].

I have also developed an extensive test-suite for testing API implementations. The API implementation is automatically tested against this test-suite after every commit, on Linux, Mac OS X, and Windows XP, with *ad hoc* testing on a range of other platforms. The API implementation currently passes all of the above tests.

Test-suite

We have also developed an extensive test-suite for validating API implementations. For the core API (including DOM and MathML DOM), and some extension APIs, a program included with the test-suite makes use of every attribute and operation in the API, and checks that invariants which are expected to be true if the implementation behaves correctly, are in fact true. In addition, the test-suite also includes a series of small programs, as well as a series of inputs to those programs, and expected outputs. For example, the program CellML2C is a small, command-line driven test program, that takes a CellML model as input, and uses the CCGS API to generate C-code from it. The test-suite calls CellML2C with 17 different models (each of which are crafted to contain peculiarities to test different features). Our API implementation is automatically tested against this test-suite after every commit, on Linux, Mac OS X, and Windows XP, with *ad hoc* testing on a range of other platforms. The API implementation currently passes all of the above tests.

In the future, we plan to add tests that the numerical results provided by implementations of the CellML Integration Services are correct, in a similar vein to the SBML test suite (`http://sourceforge.net/projects/sbml/files/test-suite/2.0.0%20alpha/`).

Comparison with libSBML

The CellML API is, to my knowledge, the first publicly available API that supports the processing of CellML models. However, there are other similar projects designed to process mathematical models in different encodings.

LibSBML [Bornstein et al., 2008] is, in many ways, analogous to the CellML API, except that it processes SBML models. As CellML provides a higher level of domain independence than SBML, it is expected that tools used across many different domains of expertise will need to exchange CellML models. In addition, some tools, such as generic modelling environments, may need to import and export both SBML and CellML models, in which case, both a CellML API implementation, and libSBML can be used together in the same program. Aside from the difference in language support, there are some additional major differences between the CellML API and libSBML.

The CellML API is designed as an API, with care taken to clearly define the interface so it could be implemented by multiple implementations (one of which we provide). This means that, if desired, someone could implement the CellML API natively in a different language, and this implementation would be source and binary compatible with other implementations. It also means that, under the CellML API, adding a new language binding involves developing code to automatically produce a wrapper from the IDL description, rather than using Swig and then manually creating wrappers to tidy up the details. The CellML API approach can therefore be expected to be more robust to changes to the API, and to the addition of completely new modules.

In addition, the CellML API takes a fairly different approach to the manipulation of MathML. The CellML API requires implementations to support an existing API, the MathML DOM [Carlisle et al., 2001], while libSBML provides a more limited Abstract Syntax Tree (AST) based approach. The libSBML approach, however, does allow for translations to and from plain text; this is a feature which we have implemented using the API in OpenCell, and plan to move into the API.

8.4 Conclusions

The CellML API and implementation is available, and is ready for widespread adoption by the community. Developers of tools which process mathematical models are strongly encouraged to support CellML, so that users of the tool can

participate in model sharing, with all the associated benefits to the scientific community. The CellML API and implementation provide facilities which should make this task substantially easier.

8.5 Availability and requirements

The CellML API is Free / Open Source software, and can be used and redistributed under any one of three different popular Free / Open Source licenses, allowing the API and implementation to be used in a wide range of public and private research and applied settings.

The latest information on how to retrieve the API and implementation is always available at http://www.cellml.org/tools/api/. The source code and change history is currently stored in a repository at SourceForge.

The API implementation can be built on any POSIX like system, including Linux, Mac OS X, and Cygwin. It can also be built using Microsoft Visual C++ 2008. The build requires the omniidl tool, which is part of omniORB, as well as libxml2, and optionally the GNU Scientific Library (GSL). If the Java bindings are desired, the Java Development Kit is required. If Javascript (XPCOM) bindings are desired, xulrunner is required.

8.6 List of abbreviations used

Abbreviation	Definition
AnnoTools	The Annotation Tools
API	Application Programming Interface
CCGS	The CellML Code Generation Service
CeLEDS	The CellML Language Export Definition Service
CellML	An XML-based language for describing mathematical models
CeVAS	The CellML Variable Annotation Service
CIS	The CellML Integration Service
CORBA	Common Object Request Broker Architecture
CUSES	The CellML Units Simplification and Expansion Service
DAE	Differential Algebraic Equations
IDL	Interface Definition Language
MaLaES	The MathML Language Expression Service
VACSS	The Validation Against CellML Specification Service
XML	The Extensible Markup Language
XPCOM	Cross-platform Common Object Model

Chapter 9

ModML Core - building differential-algebraic systems in terms of functional transformation

9.1 Introduction

Differential-algebraic equations

Systems of differential-algebraic variables have the following general form:

$$0 = F(\mathbf{x}, \mathbf{x}', t) \tag{9.1}$$

where F is some arbitrary function, t is termed the independent variable, $\mathbf{x}$ is termed the variable vector, and $\mathbf{x}'$ is the derivative of $\mathbf{x}$ with respect to t. Figure 9.1 visually demonstrates a differential-algebraic equation and its solution.

Systems of Differential Algebraic Equations (DAEs, and their less general variants, such as systems of Ordinary Differential Equations, ODEs) have a long history of application to modelling biological systems.

They are widely used to model reaction systems in well-stirred reactor vessels, as well as various other physical systems.

Two systems of differential algebraic equations involving no common variables can be trivially composed by simply including both systems in any description of the model. Composing models involving a common variable can similarly

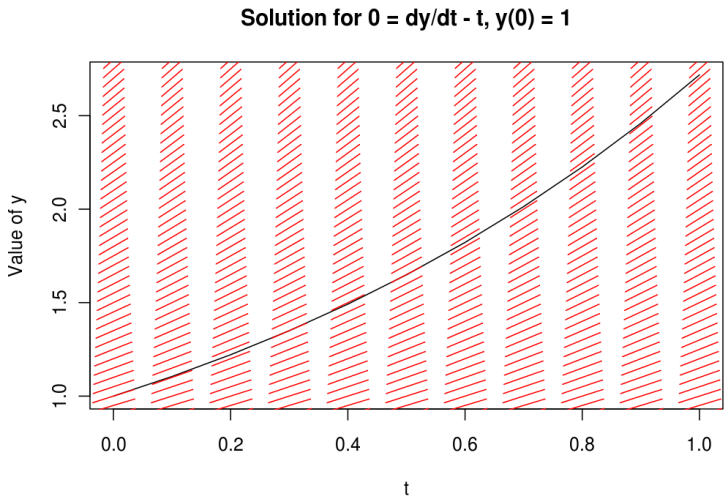

Figure 9.1: A demonstration of the nature of an initial value DAE solving problem for the simple DAE $F(y,t) = 0 = y(t) - \frac{dy(t)}{dt}$, $y(0) = 1$. The slope of the red lines shows the value of $\frac{dy}{dt}$ at different values of y and t. The black curve shows the analytic solution to the problem, in this case $y(t) = e^t$. Intuitively, the solution is the curve where the slope of the curve matches the derivatives at each point.

be done by combining the equations and ensuring that variables that have a different meaning are assigned a different symbol in the composed model.

Numerous different algorithms for solving systems of ODEs or DAEs have been developed. While no entirely numerical method for solving DAEs will ever support every pathological case, due to the possibility of steep peaks in the derivatives of variables at points not anticipated by the interpolation method used by the algorithm, there are algorithms that are usable for a wide range of real-world models, especially when the numerical integration can be supported with some analytically derived information about the model.

Imperative, Declarative, and, Functional Model Representations

There are numerous possible ways to enter systems of DAEs for computer processing. One early approach was to simply enter models as procedures in an imperative language such as C, FORTRAN, or MATLAB. An imperative language describes the sequence of steps to be taken to perform a computation. There are, however, several disadvantages to this approach.

Firstly, it encourages the mixing of the numerical integration algorithm with the formulation of the model. This problem can be reduced, to some extent, by using a numerical integrator library that provides clearer segregation between the algorithm and the user supplied functions. However, unwanted interactions can still occur due to the use of the global state.

Secondly, imperative language representations are difficult to compose efficiently, for example, to build a model that merges two existing ion channel models and adds some further equations, to link the two. In this case, a modeller would typically need to manually code up a model containing equations for each sub-model.

For this reason, non-imperative computational representations for systems of DAEs have been proposed. CellML is an example of a representation format for mathematical equations, including systems of DAEs. It does not express how to solve the systems of DAEs, but instead, declares the equations present. As such, it can be classified as a declarative representation. A declarative language describes how things are, rather than how to compute something.

Modelica has taken an object-oriented declarative approach; in this approach, models are represented as equations, but these equations are attached to classes, and each class can be instantiated multiple times to produce objects. Models are constructed by connecting multiple objects together, so the details of the parts are abstracted away.

Other representations, such as SBML, have also taken a declarative approach, but have included built in facilities for representing biochemical reactions.

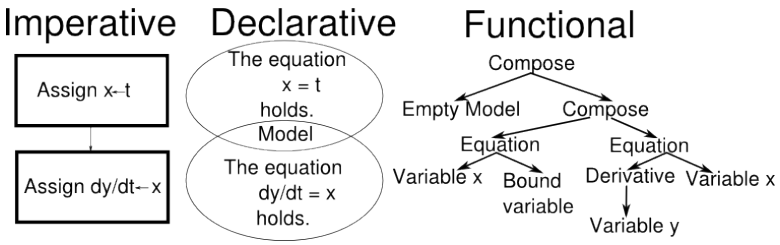

Figure 9.2: A comparison of the main differences in approach between imperative, declarative, and functional descriptions of the model $x = t$, $\frac{dy}{dt} = x$.

These facilities allow models containing biochemical reaction networks to be represented more succinctly, at the cost of requiring tools to understand this information specific to the domain of biochemical reaction networks. The benefits of this approach are therefore not scalable to models touching many different domains, and therefore potentially benefiting from many different Domain Specific Languages (DSLs).

A natural solution to the problem of allowing the use of DSLs in models in a scalable manner is to make the definition of DSLs a reusable part of the model, rather than a part of tools carrying out the solving. However, to allow a sub-module of a model to define a new DSL, significantly more powerful language features are required than are available in CellML, SBML, or Modelica.

Figure 9.2 shows the difference between imperative, declarative, and functional models.

Attributes of functional programming languages

A pure functional programming language is one where computations are carried out entirely by applying functions to immutable data structures or functions to produce another data structure or function. Such languages are therefore a natural fit to the problem of transforming a domain specific representation of a model (for example, as a description of the chemical reactions involved in gene expression and its regulation); the transformations could, for example, produce a system of differential-algebraic equations.

There are several types of functional programming language. One dividing attribute is whether the language is strict or lazy. In lazy programming languages, function applications are only evaluated when the value is required, while in strict programming languages, function applications are evaluated as soon as they are constructed. There are advantages to both approaches; laziness can result in some unnecessary work being avoided entirely, but introduces the overhead of storing the required function applications rather than a potentially simpler value. It is worth noting that lazy functional languages tend to provide facilities for strict evaluation, and vice versa, so the difference is in the default mode of evaluation.

Another factor in the classification of functional languages is whether the language is strictly or weakly typed. In a strictly typed language, the type of an expression is known at compile time, and expressions can be checked to ensure that the types are consistent. In a weakly typed language, there is no compile time guarantee of the type of a value; a function of a real value could have a different type depending on the real value. Historically, limitations in type systems have meant that it was only possible to do certain kinds of type-generic programming in weakly typed languages. However, the advent of languages with more powerful type systems based on typed lambda calculus, and type inference algorithms like Hindley-Milner [Milner, 1978, Damas and Milner, 1982], have largely alleviated this concern, and the guarantees that static typing offer now arguably outweigh the limitations imposed by the type system, especially when the type system can be applied to check many of the correctness constraints specific to a particular application.

Mallavarapu et al. [2009] described a method for representing biochemical reaction networks in a functional programming language, namely Lisp. However, the framework presented in that paper, little b, builds the concept of reactions into the framework, rather than providing a functional transformation from reactions to mathematical equations as part of the model. The variant of Lisp used in little b, is, by default, both weakly typed and strict.

When building a reaction model, it would be useful to define a large, reusable library of reactions (or other domain specific information relevant to a model),

but to only use information on a particular reaction when the species essential for that reaction to take place are present. This is likely to be the most practical if the language in which the models are represented is lazy; that way, complex and potentially computationally expensive transformations can be applied to build the sub-models in the library without incurring a performance penalty.

For these reasons, there is a clear need for a model representation language that allows models to be lazily transformed from a domain specific representation into a representation as a symbolic mathematical model.

I have developed such a language, called ModML, on top of Haskell, an existing lazy, strictly typed pure functional programming language.

Appendix A contains a summary of the basic facilities in Haskell that are will be utilised in the remainder of this chapter.

9.2 Introducing ModML Core

A ModML model is a pure functional computation that produces a particular data structure. The computed data structure is called the ModML Core data structure. It is designed to be as minimalistic as possible, while still allowing all systems of differential algebraic equations that can be expressed using a reasonably rich set of common mathematical operators.

Expressions, equations, and variables

In ModML Core, mathematical expressions of real numbers are expressed using a recursive data structure called RealExpression. RealExpression has constructors for constant values, the values of variables, the addition, subtraction, division and multiplication and taking powers and logs of expressions, trigonometric and hyperbolic functions and their inverses, the floor and ceiling functions of expressions, and a conditional function. In addition, because RealExpressions are used in models of differential-algebraic equations, constructors for the value of the bound variable (such as time), and for taking a derivative of an expression

with respect to the bound variable are also included. Finally, two special purpose constructors, RealExpressionTag and RealCommonSubexpression are present, as discussed below. The full structure of RealExpression is given in Listing 9.1, and graphically in Figure 9.3.

The conditional function is an expression on a boolean condition, and two real expressions, one that applies if the condition is true, and the other that applies if the condition is false; this creates a need to introduce a boolean data structure. Boolean expressions are represented using a different data structure, called BoolExpression. BoolExpressions have constructors for constants, the logical operators And, Or, and Not on boolean expressions, and the operators less than and equal on real expressions, and a special purpose BoolCommonSubexpression constructor (discussed below). Boolean expressions can include real expressions, and real expressions can include boolean expressions (i.e. the structures are co-recursive), and so complicated expressions can be built up with arbitrarily deep nesting of each type inside the other.

Because ModML models are typically based on a series of transformations, it can be difficult to determine the source of a particular expression in the final model. To simplify the debugging of such models, ModML also includes an expression constructor that represents the identity function of an expression, but additionally includes a user-specified string as a tag to facilitate debugging.

RealEquation is a simple data structure with a single constructor of two Real-Expressions. It is used in the list of equations, where the full generality of BoolExpression wouldn't be appropriate.

Variables in ModML Core are represented by a data type for a variable. This type has a single constructor, which takes an integer value. Because Haskell is pure functional, data structures must be distinguished solely by the data they contain (and not, for example, by a pointer address). Two different variables must therefore have a different integer value. The mechanisms that ModML provides to compose models while ensuring that different variables are given different identifiers are discussed below.

Listing 9.1: The RealExpression structure

```
data RealExpression =
    -- Any real value, as a constant.
    RealConstant Double |
    -- A free variable...
    RealVariableE RealVariable |
    -- The bound variable of the integration...
    BoundVariableE |
    -- The derivative of an expression with respect to
       the bound variable...
    Derivative RealExpression |
    -- A common subexpression tagged expression.
    RealCommonSubexpressionE RealCommonSubexpression |
    -- If x {- then -} b {- else -} b
    If BoolExpression RealExpression RealExpression |
    -- A sum of two expressions.
    Plus RealExpression RealExpression |
    -- First expression minus second expression.
    Minus RealExpression RealExpression |
    -- The product of two expressions.
    Times RealExpression RealExpression |
    -- a `Divided` {- by -} b
    Divided RealExpression RealExpression |
    -- a `Power` b - a to the power of b.
    Power RealExpression RealExpression |
    -- The floor function...
    Floor RealExpression |
    -- The ceiling function...
    Ceiling RealExpression |
    -- LogBase a b = log_a b
    LogBase RealExpression RealExpression |
    -- Trigonometric functions...
    Sin RealExpression |
    Tan RealExpression |
    Cos RealExpression |
    ASin RealExpression |
    ATan RealExpression |
    ACos RealExpression |
    Sinh RealExpression |
    Tanh RealExpression |
    Cosh RealExpression |
    ASinh RealExpression |
    ATanh RealExpression |
    ACosh RealExpression |
    RealExpressionTag String RealExpression
          deriving (Eq, Ord, D.Typeable, D.Data, Show)
```

Building complicated expressions from simple parts

ModML only provides a minimalistic set of primitive operations. More complex operations are instead built by combining these simple primitives. This approach makes it simpler to implement good quality solvers for ModML models (in a similar vein to the justification behind the development of RISC architectures [Patterson and Ditzel, 1980]). Functions included as part of a model (including in a reusable library) can then build more sophisticated operations on top of the simple built-in ones.

One issue that arises from this approach is that the same sub-expression will appear multiple times in the same model. For example, "x is greater than y" can be implemented as "(not (x is less than y)) and (not (x is equal to y))", where x and y could be arbitrarily complex expressions, and it would be wasteful to evaluate them twice. In the overall expression, each instance of x (or y) can be considered a common sub-expression. These sub-expressions could in turn include more common sub-expressions. The computational complexity of re-evaluating each common sub-expression every time it appears grows exponentially with the depth of nested common sub-expressions. It is therefore necessary for ModML to provide an efficient way to handle models including complex common sub-expressions.

One approach that could be taken would be automated common sub-expression elimination. However, automated common sub-expression elimination can potentially slow down evaluation when the cost of identifying and removing sub-expressions is too high. It is outside the scope of ModML Core, as a representation format, to rule out the creation of a solver that uses automatic common sub-expression elimination, but ModML Core has been designed to provide an alternative solution so automatic common sub-expression elimination is not required. Instead, to simplify the implementation of solvers, and also to encourage modellers to clearly structure their model code to avoid repetition, ModML Core provides common sub-expression constructors for both RealExpression and BoolExpression, used to manually identify a common sub-expression. These common sub-expressions are identified by an integer, in much

Listing 9.2: The BasicDAEModel structure

```
data BasicDAEModel = BasicDAEModel {
        -- The equations which apply for this model at
            all times...
        equations :: [RealEquation],
        -- The boundary conditions (equations which
            apply at the starting
        -- point, and/or after interventions). Note that
             other equations also
        -- apply at these times as well.
        boundaryEquations :: [(BoolExpression,
            RealEquation)],
        -- Expressions which controls when the solver
            needs to be restarted.
        -- The values should cross zero at the point
            when the solver needs to
        -- be restarted.
        interventionRoots :: [RealExpression],
        -- An expression which the solver should try to
            keep positive.
        forcedInequalities :: [RealExpression],
        checkedConditions :: [(String, BoolExpression)],
        variables :: [RealVariable],
        commonSubexpressions :: [AnyCommonSubexpression
            ],
        annotations :: M.Map (String, String) String,
        contextTaggedIDs :: M.Map (D.TypeCode, D.
            TypeCode) Int,
        nextID :: Int
    } deriving (Eq, Ord, D.Typeable, D.Data, Show)
```

the same way that variables are identified.

The top-level ModML Core data structure

The top-level ModML Core data structure is called BasicDAEModel, and has a single constructor of the same name. The constructor includes multiple named arguments, each of which is discussed in turn below. The full BasicDAEModel data structure is given in Listing 9.2.

Equations and boundary equations

The equations parameter is a list of equations. Each member of this list is interpreted as an assertion that a particular mathematical equation is always true.

However, because variables (which change with respect to the main bound variable) are expressed as variables, and not functions of time, this formulation alone would make it impossible to specify boundary conditions (that is, the value of an equation at a particular value of the main bound variable, for example, at time 0). For this reason, an additional parameter called boundaryEquations is included in the constructor. This parameter is a list of pairs of a BoolExpression and a RealEquation. The semantics are that each entry in a list is an assertion that whenever the BoolExpression has value true, the RealEquation is asserted to be true.

Intervention roots

Models may include discontinuities (for example, due to the use of conditional expressions). These discontinuities are often poorly handled by numerical solvers (and often entirely skipped over by adaptive solvers). These problems can generally be avoided by restarting the solver at discontinuities. To allow modellers to indicate the location of such discontinuities, the constructor includes a parameter interventionRoots, a list of RealExpressions. The semantics are that the solver should expect a discontinuity at the point at which the RealExpression changes sign.

Forced and checked inequalities

In addition, many solvers support the ability to hold a particular inequality to be true by rejecting steps that cause it to be false, and therefore ensure the correct solution out of several choices is found. To support this, the constructor includes a parameter called forcedInequalities, which is a list of RealExpressions.

The semantics are that in the desired solution, each of the RealExpressions always has a positive value.

As well as forcing inequalities to be true, sometimes modellers will have conditions that are not strictly necessary to constrain the model, but which check that the model and its parameters are producing results in the expected range. For this reason, the BasicDAEModel constructor includes a checkedConditions parameter, which is a list of string and BoolExpression pairs. The semantics are that the model is not valid if the BoolExpression is False, and the string provides a description of the problem.

Identifier allocation

As presented earlier, variables and common expressions are identified by a unique identifier. When expanding on a model, it is often necessary to add a new variable or common sub-expression. To allow this, the BasicDAEModel constructor includes a parameter called nextID, which is an integer. NextID is a value one more than the highest identifier used in the model; to make a new model with an additional variable, in addition to adding the variable to the model with the old value of nextID, the nextID is increased by one. Facilities for simplifying the allocation of identifiers are discussed in Section 9.3.

Annotations

While the model itself is all that is necessary to produce the numerical results, the interpretation of the model requires additional information. Other representation languages like CellML and SBML provide facilities for associating metadata, in the form of arbitrary XML or RDF/XML fragments, with parts of the model, to carry this additional information. This approach solves the basic problem of model annotation, but when composing complicated models, sometimes it is useful to also compose metadata.

For example, a modeller might build a generalised model of an ion transporter, but then compose that transporter into a model as a sodium transporter. They

could rewrite the metadata manually to state that the transporter is a sodium transporter, but it would be preferable if the facilities that allowed the model to be composed and specific variables slotted in also allowed the metadata to be composed in the same way.

ModML Core therefore includes annotations on the BasicDAEModel constructor, as a map from a pair of strings to a string. The intention is that each entry in the map represents an assertion in a similar way to in RDF; the first string is the subject and describes what (for example, which part of the model) the assertion is about, the second is the predicate, and describes which property is being described, and the third is the object and describes the value of the predicate for the subject. All of the key data types in ModML Core derive a Show interface, meaning that they can be converted to a standard string representation, allowing them to be annotated.

Consider the following example of annotations: ModML variables have a number, not a name, associated with them, but it is useful to modellers to have a descriptive name rather than something like 'Variable 132'. To provide this, an annotation with subject Variable 132, predicate "nameIs", and subject "Ca2+ Concentration in micromol/L" could be added. The "nameIs" predicate can be used to assign a human readable name to any structure in ModML Core that has an instance of Show. ModML Core provides facilities to simplify both annotations in general, and name annotations; these are discussed in Section 9.3.

Type-tagging support

In ModML Core, each variable needs to have an identifier allocated. This means that naively importing a module and importing that extension twice will result in separate variables. Sometimes this is the desired semantics; for example, if an ion channel model is used twice (for example, once as a sodium channel, and once as a potassium channel), it would be correct to have a different variable for the flux of ions across the channel for each import. Other times, however, it would be desirable to be able to import the same variable into a model multiple

times.

For example, many parts of a model might make reference to the temperature. While a modeller could allocate the temperature variable once and pass it as a parameter to all functions, this manual approach would not be scalable to large numbers of variables. Instead, ModML Core allows type context tagging of identifiers. A type context tag makes use of the fact that a type defined in Haskell has the same unique identity every time the module containing that type is imported. This type-level tag can be converted to the value level and used as a unique identifier for identifying an entity.

Because identifiers can be used to identify a number of things in models - variables, real and boolean common sub-expressions, and potentially user-defined entities, a second type tag is used to identify the way the identifier is used (for example, RealCSEContextTag is used to identify an identifier for a real-valued common sub-expression).

The BasicDAEModel constructor includes a parameter that maps from the two type tags to an Integer; when adding a new variable to a model, if a type-tagged identifier already exists for the variable, the Variable constructor can be used to obtain the corresponding data structure. If it doesn't exist, the nextID mechanism can be used to determine the next identifier, and the composed model has nextID increased and contextTaggedIDs updated so future references to the same variable can build the correct Variable data-structure. Section 9.3 describes the facilities to simplify the use of this approach.

Lists of variables and sub-expressions

Finally, the BasicDAEModel constructor includes two arguments, for this list of all variables, and the list of all common sub-expressions (via a type called AnyCommonSubexpression, with two constructors, wrapping real and boolean common sub-expressions, respectively).

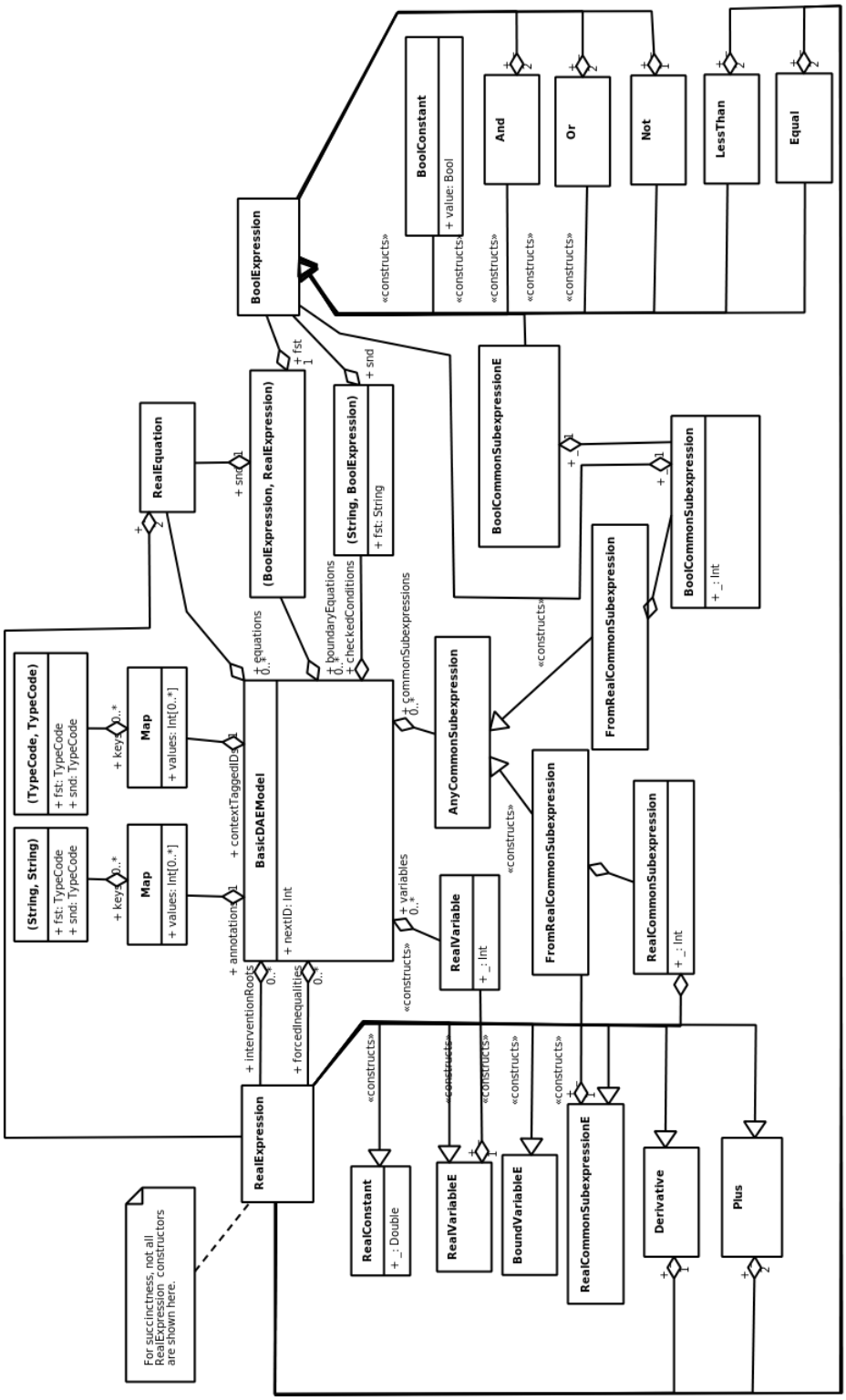

Figure 9.3: A UML class diagram showing the ModML Core data structures.

9.3 Building a domain-specific language around ModML Core

The core data structures described in the previous section contain all the information needed to represent a wide range of models of systems of differential-algebraic equations. As discussed in section 9.4, this representation as a data structure simplifies the development of software for solving models (and performing symbolic mathematical manipulations to improve solver performance). However, it would be difficult to directly input models using the constructors presented so far; for example, referring to variables only by sequential numbers and then giving them a name in a specific section of the model would make models very hard to create, read and maintain.

Fortunately, because ModML models are a computation that produces a data structure, and not just the data structure, additional functions can be provided to abstract away commonly repeated patterns, and to structure model input in a more natural way. In this section, I present functions that are widely applicable to creating models, and so are included in the ModML Core package. Because they are composed from the primitive constructors presented in the previous section, solver software does not need to do anything specifically to support them.

Model building through sequential transformation

The previous section presented arguments of the BasicDAEModel constructor, such as contextTaggedIDs and nextID, that make it possible to build up a model by starting with an empty model, that is, a model with no equations, and performing a series of transformations, each building a new model as the previous model with something added (such as a variable or equation). ModML Core provides a value, nullModel, that can be used as the starting point. nullModel is a BasicDAEModel with all the lists and maps discussed in the previous section empty, and a nextID of zero.

Managing the composition of models using the above approach, however, would

be rather tedious, and would presumably involve keeping track of values like model, model', model" and so on to denote each successive transformation of a model. Instead, ModML Core makes use of monads.

Monads and state

Monads are a concept from category theory that have been introduced to functional programming languages such as Haskell[Jones and Jones, 2003]. A monad allows the native type system to be embedded into a different type system. The specific definition of monad (the Monad type class) used in Haskell is known as the Kleisli construction. It provides a way to construct the monadic type from the native type, and has two essential class methods on it: one to embed a value from the native type system into the monadic type (called the unit function in general literature on the Kleisli construction, and return in Haskell), and the binding operation, which, given a monadic type value and a function from a native type value to a monadic type value, produces a monadic type value (this operator is called (>>=) in Haskell).

Monads are naturally suited to describing how to update state. For example, Control.Monad.State is an existing Haskell module that provides the State monad. The monadic type of the State monad can be thought of as a description of how to update some state information as a side-effect of performing a computation. The unit function (return) produces a state monad that returns the specified value but doesn't change the state. The binding operation models the sequential nature of state updates, allowing such updates to be composed in a specified order.

In addition, Haskell provides syntactic sugar (i.e. syntax which is a short-hand for some other construct, designed to be easier to read and understand) to make code that deals with Monads clearer. The do statement describes a series of composed monadic operations; in the below listing, f and g are equivalent.

```
f x = do
        y <- doSomethingWith x
        z <- doSomethingWith y
```

```
            return (z + 1)
g x = (doSomethingWith x) >>=
      (\y -> doSomethingWith y) >>=
      (\z -> return (z + 1))
```

It would be possible to embed a monadic type inside a monadic type, but this wouldn't be particularly useful for interspersing access to the functionality of the inner and outer monad. Haskell provides a type class for monad transformers, which are monads that additionally provide an operation, lift, which maps from the underlying monadic type to the monadic transformer type, allowing interspersed use of both monads. Multiple layers of monad transformers can be used, creating a monad transformer stack.

Model builders

ModML Core provides a data type, ModelBuilderT, which is a monad transformer, a wrapper around the state monad transformer Control.Monad.State.StateT, where the state is a BasicDAEModel. In addition, for the most common case where there is no monad underneath ModelBuilderT, ModML Core defines ModelBuilder as a ModelBuilderT with the Identity monad underneath (a monad that does nothing except wrap the native type in a monadic type). For simplicity, for the remainder of this chapter I shall refer to ModelBuilder even when the functions described are usable with any monad underlying ModelBuilderT.

A ModelBuilder represents a series of sequential transformations of a model, with no pre-defined starting model. ModML Core provides an instance of the existing type class Control.Monad.State.Class.MonadState. MonadState has two class functions, get and put, to retrieve and set the state, respectively, and includes an ordinary function modify that builds on get and put. The type of modify for ModelBuilder is (BasicDAEModel → BasicDAEModel) → ModelBuilder (), i.e. a function that takes a function between BasicDAEModels, and produces a ModelBuilder with an inner type of () (a tuple of size zero, used in Haskell as a placeholder type and value for when no information is required). The modify

function is used to build up a number of key operations that can be sequenced into a ModelBuilder, as presented in the remainder of this section.

In Haskell, the monadic type wrapping a type t is distinct from type t, and so in ModML, a value of type ModelBuilder t (a computation that transforms a model and produces a value of type t) cannot be used in place of a value of type t, even if the value of monadic type doesn't actually represent a transformation. For example, if x and y were of type ModelBuilder RealExpression, the expression Plus x y would not pass type checking (recall that Plus is the constructor for a RealExpression that adds two RealExpressions). However, using do notation, the following expression is valid, and can have type ModelBuilder RealExpression:

```
do
  x' <- x
  y' <- y
  return (Plus x' y')
```

This notation is syntactic sugar for:

```
(x >>= (\x' -> y >>= (\y' ->
  return (Plus x' y'))))
```

Lifted functions for building expressions

The standard Haskell library provides functions liftM, liftM2, and so on, which can be used to apply a function of one argument, two arguments, and so on, respectively, over the base type to the monadic type, so liftM2 Plus x y has identical semantics to the two above code samples. None of these three forms is a very intuitive way to build a complicated model. The first problem is that repeated use of higher order functions like liftM2 in an expression impede readability. The second is that expressing complicated mathematical expressions in pre-order (e.g. Times 3 4) is less familiar than an in-order representation like (3 * 4).

The first problem can be solved by defining a new function from the lifted form of the function on RealExpression. Haskell allows any function to be used in-order, so this also allows the second problem to be solved. For example:

```
plusX :: ModelBuilder RealExpression ->
         ModelBuilder RealExpression ->
         ModelBuilder RealExpression
plusX = S.liftM2 Plus
minusX :: ModelBuilder RealExpression ->
          ModelBuilder RealExpression ->
          ModelBuilder RealExpression
minusX = S.liftM2 Minus
myExpression :: ModelBuilder RealExpression ->
                ModelBuilder RealExpression ->
                ModelBuilder RealExpression ->
                ModelBuilder RealExpression
myExpression x y z = (x `minusX` y) `timesX` z
```

In this example, plusX and minusX are defined in terms of Plus and Minus, and then myExpression is defined to be a function from three model builders (x, y, z) to a model builder. Note that the lines with :: in them define the type of the expression; they are optional because Haskell can perform type inference. The type signatures are more specialised than would be ideal in practice, because plusX could not, for example, be used with a different monad transformer stack despite there being no good reason for that restriction in this example; this could be remedied by either omitting the types, in which case the most general type signature possible will be inferred, or by writing a more general type signature.

ModML Core provides three types of functions for building expressions. Expression-only functions take RealExpression (or BoolExpression) inputs, and produce RealExpressions (or BoolExpressions). Using expression data structure constructors directly, but where such functions are composed in ModML Core, the suffix E is appended to the name. For example, the square root function is defined in terms of the primitive operation of raising to a

power, and the expression-only form called sqrtE. Functions that require the allocation of a new identifier (for example, for a common sub-expression) do not have an expression-only form.

The next form takes RealExpression arguments, but returns ModelBuilder RealExpressions. Functions in these form are denoted by the suffix M. Finally, functions that take ModelBuilder RealExpression arguments, and return a ModelBuilder RealExpression are denoted by a suffix of X. Consistently using functions in this form is the simplest from a modeller perspective. Where a name would not overlap with an existing name (for example, in the Haskell Prelude package), the name with no suffix is used as an alias for the most commonly useful form.

For example, consider producing a real-valued constant from a double precision floating point value. The constructor RealConstant is the expression-only form, and realConstantM is a function from Double to a ModelBuilder RealExpression. There is no realConstantX as the parameter is not an expression, and realConstant is an alias for realConstantM.

Allocating variables

The Variable type can be wrapped to create an expression using realVariableX, which takes a ModelBuilder RealVariable and produces a ModelBuilder RealExpression. A ModelBuilder RealVariable can be obtained using mkNewRealVariable (or mkNewNamedRealVariable if an annotation giving the variable a name should also be added). However, care needs to be taken when using this approach; it would be a user error to pass a ModelBuilder RealVariable that allocates a new RealVariable to realVariableX multiple times, when the intention is that they will all refer to the same variable.

Several convenience functions are provided that reduce the risk of user errors like the one described above. The function mkNewRealVariableM has type ModelBuilder (ModelBuilder RealVariable). The outer ModelBuilder allocates the RealVariable identifier, and so the inner one can be evaluated multiple times using the X-suffixed functions, each of which will refer to the same

variable. However, in most cases, another function, newRealVariable, with type ModelBuilder (ModelBuilder RealExpression) is more convenient. A variant that adds a name annotation, newNamedRealVariable, is also provided, and this form is usually the most intuitive to use when building models.

Adding equations

Facilities similar to those for RealExpressions are provided for the higher level aspects of the model. A new equation can be added with newEqM or newEqX (newEq is an alias for newEqX). Similar functions exist for adding boundary equations: newBoundaryEq, with type ModelBuilder BoolExpression → ModelBuilder RealExpression → ModelBuilder RealExpression → ModelBuilder (), can be used to add an equation that is true under the conditions specified by the first argument. It is often more convenient to use initialValue (an alias for initialValueX), of type Double → ModelBuilder RealExpression → Double → ModelBuilder (), which adds a boundary equation valid when the bound variable equals the first argument, at which time the RealExpression in the second argument is equated to a constant made from the third argument.

A simple example

Using the facilities described so far, the following example model could be built:

```
myModel =
  do
    let t = boundVariable
    x <- newNamedRealVariable "x"
    y <- newNamedRealVariable "y"
    z <- newNamedRealVariable "z"
    x `newEq` (y `plusX` (z `timesX` (realConstant 3)))
    y `newEq` ((realConstant 2) `timesX` t)
    z `newEq` ((realConstant 3) `timesX` t)
```

This model defines three variables, x, y, and z, and three equations in terms of the bound variable t, specifically $x = y + 3z$, $y = 2t$, $z = 3t$.

Symbols for use in equations

The use of infix notation like x ‘plusX‘ y makes the model more readable compared to prefix notation plusX x y. However, many users will be more familiar with using non-alphanumeric characters to more succinctly represent equations (such as the asterisk, *, to represent multiplication), due to the ubiquity of such representations in programming languages such as MATLAB.

ModML is built on Haskell, and the Haskell Prelude already defines operators such as * and + as class functions on the Num type class, with others on various other type classes (for example, the equality operator, == is defined on the Eq type class). However, making RealExpression an instance of Num, Eq[1], Ord, Floating, and the other various type classes from Prelude would not produce coherent semantics.

For example, the == operator must produce a boolean result (i.e. of type Bool), but ModML Core expressions cannot be evaluated, in general, outside of the solver, without reference to the rest of the model and a value of the bound variable. To avoid this problem, new operators have been defined for use in ModML Core models.

ModML Core defines a set of operators, each of which is an alias for an X-suffixed function. These operators are identified as being ModML operators by the present of a period before and after a symbol. The operators defined in this way are .&&. for logical And, .||. for logical Or, . < . for less than comparisons, . <= . for less than or equal to, . > . for greater than, . >= . for greater than or equal to, . == . for equality, . + . for addition, . − . for subtraction, . ∗ . for multiplication, ./. for division, and . ∗ ∗. for raising to a power.

Using these operators, the above example becomes:

[1] ModML Core does, in fact, implement Eq and Ord on RealExpressions, but the instance functions compare the RealExpression data structures and not the values they evaluate to.

```
myModel =
  do
    let t = boundVariable
    x <- newNamedRealVariable "x"
    y <- newNamedRealVariable "y"
    z <- newNamedRealVariable "z"
    x `newEq` (y .+. (z .*. (realConstant 3)))
    y `newEq` ((realConstant 2) .*. t)
    z `newEq` ((realConstant 3) .*. t)
```

These facilities, taken together, allow for the succinct basic definition of the underlying mathematics in models. In the following chapters, I discuss how building richer, more domain specific libraries that can be included in ModML models allows for more descriptive modelling.

Now that a basic framework for model representation has been laid down, it is important to consider how such models can be solved. The next section addresses this issue.

9.4 Building a solver for ModML Core

I have implemented a software package (ModML Solver) that takes a ModML model and solves for the state variables, with respect to the bound variable, using an existing numerical integrator program.

Background

The current state of the art Differential Algebraic Equation solvers extend the techniques used in Ordinary Differential Equation solvers. ODE Initial Value problems are written in the form:

$$y'(t) = f(t, y(t)) \tag{9.2}$$

$$y(0) = \mathbf{y}_0 \tag{9.3}$$

where t is the bound variable (referred to as time for simplicity) for the problem, $y(t)$ is the function from time to the vector of state variables, $y'(t)$ is the function from time to the vector of rates of change of the state variables with respect to time, and $\mathbf{y}_0$ is the vector of initial value constants.

Ordinary Differential Equation solvers have a long history, commencing with the Euler methods, developed in the 18th century by Leonhard Euler. The Forward Euler method computes the values of the state variables for discrete time steps from the previous time steps as follows:

$$y(t_{n+1}) = y(t_n) + (t_{n+1} - t_n)y'(t_n) = y(t_n) + (t_{n+1} - t_n)f(t_n, y(t_n)) \quad (9.4)$$

where t_i is the ith time step, and $t_0 = 0$; the algorithm commences with the known values of $y(0) = y(t_0)$ and proceeds to compute updates at fixed steps using Equation 9.4. Because the step size is finite, using this method results in error (other than for trivial models linear in t). With fixed step sizes, for many problems, this error can cause numerical instability and produce inaccurate results. Problems that produce such numerical instability are known as stiff problems.

The forward Euler method is known as an explicit method, because it assigns the next values for the state variables from a formula in terms of the state variables at the previous time step and the rates at the previous time-step. The method can be adapted to the implicit form using the rates at the new time. The implicit form of the Euler method is called Backwards Euler, and takes the form:

$$y(t_{n+1}) = y(t_n) + (t_{n+1} - t_n)f(t_{n+1}, y(t_{n+1})) \quad (9.5)$$

As $y(t_{n+1})$ is used on both sides of the equation, a numerical solver is needed to find the root of the function:

$$g(\mathbf{x}) = y(t_n) + (t_{n+1} - t_n)f(t_{n+1}, \mathbf{x}) \quad (9.6)$$

This implicit form can be more stable for certain types of stiff problem.

The Euler method only evaluates the derivatives of y at a single point per time step, and so is said to be first order. The Runge-Kutta method, introduced by Runge [1895] and Kutta [1901], generalises this to compute a weighted average of a finite number of rate function evaluations per time step; the method can be applied to different orders. An implicit form of the Runge-Kutta method was introduced by Butcher [1964].

Solvers using the Runge-Kutta method introduce a number of innovations that improve the performance. For example, solvers will often use an adaptive step size based on error approximations, and will also often adapt the order of the Runge-Kutta method.

Gear [1971] showed that implicit form solvers can be adapted to solve some classes of the more general Differential-Algebraic Equation type of problem. These problems are given in the form:

$$F(t, y(t), y'(t)) = 0 \tag{9.7}$$

$$y(0) = \mathbf{y}_0 \tag{9.8}$$

A software package, DASSL [Petzold, 1982] adjusted the algorithm to extend the solving capabilities. A more recent solver, DASPK, described in Brown et al. [1994], allows Krylov subspace solvers such as the Generalised Minimum Residual algorithm (GMRES; Saad and Schultz [1986]) to be used to more efficiently solve the implicit problem. ModML Solver makes use of IDA [Hindmarsh et al., 2005], a port of DASPK from FORTRAN into the C programming language.

ModML Solver consists of a library, ModML.Solver, along with a program that uses the library to solve models and generate CSV output.

The ModML.Solver library

Given a BasicDAEModel data structure, the overall approach of the ModML Solver is to generate a C program for the model that uses IDA (along with a small solver main routine written in C), compile it using a C compiler, run the program, and communicate results back to the ModML Solver API. Results are available

by lazy evaluation of a list of IntegrationResult values. The IntegrationResult data type has five constructors, each representing a different possible outcome at each step of solving: FatalError, indicating an error in the model or solver parameters that could not be recovered from; Warning, indicating a warning from the numerical solver that could indicate problems with the results, but that does not prevent results from being obtained; CheckedConditionFail, which indicates that one of the checked conditions specified in the model was false; Result, with a single argument, a triple of a real number and two lists of real numbers, representing the bound variable value and the values of the state variables and derivatives of those variables, respectively, and Success, indicating that the numerical integration succeeded.

The solver allows the user to specify the range of bound variable values over which to solve, along with a maximum step size for the solver to take, and between reporting back a solution for a particular bound variable value. The step size used by the solver is dynamic, but setting a maximum step size can be useful for functions where the adaptive step size algorithm allows the step size to become too large to ensure the stability of the evaluation (however, reducing the step size is not the best way to deal with discontinuities; it is more accurate to instead add it to the model as an intervention). The reporting size controls which values are reported back from the solver; setting a smaller reporting step size than what the solver would have taken otherwise reduces the step size, but setting a larger one will not prevent the solver from taking intermediate steps, which are not reported back to the user. A user who wants to see all internal steps without influencing the step size choice can set showEveryStep, a boolean parameter, to True.

The solver additionally allows for the relative and absolute error tolerances (used by IDA to decide whether or not to reject a step) as scalars (i.e. values that are tested against the L_2 norm of the error vector).

In addition the solver allows specific real parameters to be overridden; doing this through the API, rather than by changing the model, has the benefit that much of the analytic processing done by ModML Solver can be carried out once for multiple sets of parameters, rather than once per set of parameters.

IDA requires that the caller provide the initial values and a function to compute the residuals by subtracting one side of Equation 9.7 from the other:

$$r(t) = F(t, y(t), y'(t)) - 0 \tag{9.9}$$

In addition, the caller can optionally provide a function to compute the Jacobian matrix used by the solver:

$$J(t) = \frac{\partial F(t, y(t), y'(t))}{\partial y(t)} + \alpha \times \frac{\partial F(t, y(t), y'(t))}{\partial y'(t)} \tag{9.10}$$

If the Jacobian function isn't provided, IDA will compute a numerical Jacobian using a finite differences approach; the benefit of providing the Jacobian is that direct computation of the Jacobian can be more efficient, and also that for some models, computing the Jacobian numerically can create stability problems when applying the finite difference causes evaluation at a point where the residual function is not defined.

ModML Solver initially processes a model to simplify it symbolically to allow it to be more readily processed. The first phase is simplification of the mathematics. During this phase, expressions made up entirely of constants are recursively simplified to constants. Additions or subtractions of zero, and multiplications or divisions by one are removed, and multiplication of anything by zero is simplified to a constant zero. For example, if x and y are variables, the expression $x \times \frac{\sqrt{4+5}}{1+2} - y \times (3 - 2 \times 1 - 1)$ would be simplified to x. Division of zero by anything, powers of zero and one, and conditionals where the conditional is constant or the two outcomes are symbolically identical are also simplified. Boolean expressions are also simplified; double logical Nots are removed, logical And/Or operations where one of the arguments is constant are simplified.

Next, derivatives are simplified, so that the model only contains first order derivatives of a variable. This is done by replacing the expression inside a derivative with a new variable, if it isn't already a variable, and adding a new equation equating the new variable with the expression. For example, if t is the bound variable, and x and y are variables, then $\frac{d}{dt}(x + \frac{dy}{dt})$ is simplified by assigning a new variable (call it z), changing the expression to $\frac{dz}{dt}$, and adding an equation $z = x + \frac{dy}{dt}$.

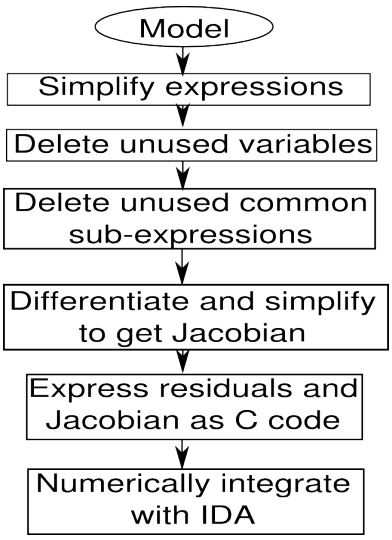

Figure 9.4: The steps taken by ModML Solver to obtain results numerical results for a model.

The simplification process (or processing carried out as part of the model description itself) may have resulted in common sub-expressions or variables that are not referenced from any equations (or, in the case of common sub-expressions, in any boundary equations, intervention roots, forced inequalities, or checked conditions). ModML Solver therefore removes these variables and sub-expressions, leaving only those that are used.

Although ModML Core identifies variables by a unique identifying number, there is no requirement that these numbers be consecutive, and even if they were, the previous variable deletion step may have created gaps in the sequence. As the solver algorithm requires contiguous array of all variable values, ModML Solver allocates each variable an index.

ModML Solver finds the initial conditions by using symbolic manipulation, followed by numerical solution. Recall that in ModML Core, there are two types sets of equations, one that is true at all times, and one that is true when a particular boolean expression is true (the boundaryEquations). ModML Solver initially groups the boundaryEquations by symbolic comparison of the boolean expressions. This symbolic comparison is limited at present to checking that the

expressions are equal; for this reason, ModML Solver performs better if users ensure they always represent conditions (for example, $t = 0$) in a consistent form. Once boundaryEquations have been grouped by condition, for each condition, equations in the form $v_i = h(v_1, v_2, ..., v_{i-1})$ are added to a map from v_i to $v_1...v_{i-1}$, where $v_1...v_i$ are arbitrary variables, and h is an arbitrary function. This map is then processed using a depth first search to build an ordering of variable assignments to make, starting with the entries in the map where no variables are required, and adding new entries as the prerequisite assignments have been completed.

Not all variables can be assigned initial conditions using the above algorithm (because the system can contain equations or systems of equations that the analytic solver cannot handle), but providing conditions for as many as possible can help the next phase of the initial value solution process. Variables are given an initial value based on the ordered assignments generated from the previous procedure, and for those for which no assignment could be made, an arbitrary value of 0.1 is instead assigned. ModML Solver additionally generates a residual function that computes the residuals for all equations and boundaryEquations for which the condition holds. This residual is minimised using the Levenberg-Marquardt algorithm[Levenberg, 1944, Marquardt, 1963] to give the initial conditions for the model.

Analytic Jacobian generation

The code to evaluate the Jacobian is generated by recursive rule-based symbolic differentiation interspersed with simplification using the rules described above. Given the limited number of simplification rules available, the choice of symbolic differentiation rules to apply is more important than it might first appear, because it is important that the rules do not generate terms that could theoretically be simplified out, but not with the simplification rules available. Otherwise, two problems arise; firstly, the derivatives tend to become extremely complicated, and secondly, problems like numerical overflow can occur unnecessarily in parts of the expression that could otherwise be simplified out.

The following numerical differentiation rules are implemented using Haskell pattern matching:

- Constant real values become constant 0.
- Variables become constant 1 if the differentiation is with respect to that variable, and constant 0 otherwise.
- The bound variable becomes constant 0 (symbolic differentiation with respect to the bound variable, which might be time, is not required to compute the Jacobian, only differentiation with respect to other variables).
- The derivative of a variable (with respect to the bound variable) becomes constant 1 if differentiation is with respect to the derivative of that same variable, and constant 0 otherwise. In other words $\frac{\partial}{\partial dv/dt}(\frac{dv}{dt}) = 1$, but for $u \neq dv/dt$, $\frac{\partial}{\partial u}(\frac{dv}{dt}) = 0$.
- Conditional expressions become conditional expressions with both of the possible outcomes differentiated.
- Plus and Minus constructors retain the same structure, but the expressions inside them are differentiated.
- Times constructors are differentiated according to the standard product rule:

$$\frac{d}{dy}(f_1(y) \times f_2(y)) = f_1(y) \times \frac{df_2(y)}{dy} + f_2(y) \times \frac{df_1(y)}{dy} \tag{9.11}$$

- Divided constructors are differentiated according to the following rule:

$$\frac{d}{dy}\left(\frac{f_1(y)}{f_2(y)}\right) = \frac{\frac{df_1(y)}{dy} \times f_2(y) - \frac{df_2(y)}{dy} \times f_1(y)}{f_2(y)^2} \tag{9.12}$$

- Power constructors are handled by two separate cases; the first case is a specialised version of the second, but as discussed in more general terms above, the more general case is not simplified well in the case of constant exponents. An expression $f_1(y)$ raised to a constant power n, is simplified by the following rule:

$$\frac{d}{dy}(f_1(y)^n) = n \times f_1(y)^{n-1} \times \frac{d}{dy}(f_1(y)) \tag{9.13}$$

- The more general exponentiation rule used is:

$$\frac{d}{dy}(f_1(y)^{f_2(y)}) = f_1(y)^{f_2(y)} \times (\frac{f_2(y)}{f_1(y)} \times \frac{d}{dy}(f_1(y)) + \frac{d}{dy}(f_2(y)) \times \ln f_1(y)) \tag{9.14}$$

This rule is derived as follows:

$$f_1(y) = e^{\ln f_1(y)} \tag{9.15}$$

$$f_1(y)^{f_2(y)} = e^{\ln(f_1(y)) \times f_2(y)} \tag{9.16}$$

$$\frac{d}{dy} f_1(y)^{f_2(y)} = \frac{d}{dy} e^{\ln(f_1(y)) \times f_2(y)} \tag{9.17}$$

$$= e^{\ln(f_1(y)) \times f_2(y)} \times \frac{d}{dy}(\ln(f_1(y)) \times f_2(y)) \tag{9.18}$$

$$= f_1(y)^{f_2(y)} \times (\frac{d}{dy}(\ln(f_1(y))) \times f_2(y) + \ln(f_1(y)) \times \frac{d}{dy}(f_2(y))) \tag{9.19}$$

$$\frac{d}{dy}(\ln(f_1(y))) = \frac{1}{f_1(y)} \times \frac{d}{dy}(f_1(y)) \tag{9.20}$$

$$\frac{d}{dy}(f_1(y)^{f_2(y)}) = f_1(y)^{f_2(y)} \times (\frac{f_2(y)}{f_1(y)} \times \frac{d}{dy}(f_1(y)) + \frac{d}{dy}(f_2(y)) \times \ln f_1(y)) \tag{9.21}$$

- The Floor and Ceiling constructors become a constant zero. The fact that the derivative is undefined at certain points is not relevant to the use to which the Jacobian is put.

- The LogBase constructor is differentiated according to the following rule:

$$\log_{f_1(y)} f_2(y) = \frac{\frac{df_2(y)}{dy} / f_2(y) \times \ln(f_1(y)) - \frac{df_1(y)}{dy} / f_1(y) \times \ln f_2(y)}{(\ln f_1(y))^2} \tag{9.22}$$

This rule is derived as follows:

$$\log_{f_1(y)} f_2(y) = \frac{\ln(f_2(y))}{\ln(f_1(y))} \tag{9.23}$$

$$\frac{d}{dy}\log_{f_1(y)} f_2(y) = \frac{d}{dy}\frac{\ln(f_2(y))}{\ln(f_1(y))} \tag{9.24}$$

$$= \frac{\frac{d}{dy}(\ln(f_2(y))) \times \ln(f_1(y)) - \frac{d}{dy}(\ln(f_1(y))) \times \ln(f_2(y))}{(\ln f_1(y))^2} \tag{9.25}$$

$$= \frac{\frac{df_2(y)}{dy}/f_2(y) \times \ln(f_1(y)) - \frac{df_1(y)}{dy}/f_1(y) \times \ln f_2(y)}{(\ln f_1(y))^2} \tag{9.26}$$

- The trigonometric and hyperbolic functions, and their inverses, become the standard derivatives (after application of the chain rule).

Common sub-expressions receive special treatment during symbolic differentiation: common sub-expression derivatives that are required to compute a variable derivative (either directly by a variable derivative, or indirectly by a required common sub-expression derivative) are evaluated in the order computed by a depth first search of the dependency graph. The simplifier greatly reduces the number of such derivatives required, because many common sub-expression derivatives that would otherwise be required can often be simplified out.

The CSV-AutoSolver

While ModML.Solver allows ModML tool builders to work with ModML Models, for ModML models to be immediately useful, tools that can process the models directly are needed. The program CSV-AutoSolver fills this need.

CSV-AutoSolver is a command line application that processes command line arguments indicating which ModML model to process, what kind of simulation to perform (and optionally, what non-default parameters to use, and where to store the output), and stores the results in comma-separated value (CSV) format.

ModML models make use of the Haskell language, and so it is necessary to process the model to generate the final data structure for use by ModML.Solver. This is achieved by compiling the model with GHC, the Glasgow Haskell Compiler [Jones et al., 1993, Peyton-Jones et al.], a popular Haskell compiler, together with a line of code to make it use a library that solves the model using ModML.Solver and output the required results.

ModML models might refer to an external package (for example, the ModML Units or ModML Reactions modules, or an existing complete model of a system that the user wishes to extend). To facilitate such model re-use, several mechanisms are provided to find the location of the package. Firstly, the include path can be set on the command line, causing the compiler to search the locations specified for imported modules. Secondly, GHC includes a built-in mechanism for registering and finding packages. ModML Solver hides all packages by default (to avoid name conflicts) and so a mechanism is needed to specify which packages are required. CSV-AutoSolver pre-processes model files looking for specially formatted comments at the beginning of the file. These comments must be formatted as follows (with package-name substituted as appropriate):

```
-- + Require package-name
```

ModML Core itself does not give variables a name, only a number, so using only ModML Core, variables would be called things like 'Variable 132'. Producing a CSV file with variables given names like this would make it hard to interpret a model. Instead, CSV-AutoSolver looks at the annotations on the model to find the human-readable name annotation on a variable, and only falls back on numbered variables if no annotation is found.

Two types of simulation are possible; basic time-course[2] simulations, and sampling based sensitivity analyses.

In addition to those options mentioned so far, all types of simulation allow the user to select the start time, end time, maximum solver step, and relative and

[2]In this context, time refers to the bound variable for the simulation, without loss of generality

absolute error tolerance from the command line. Additionally, sometimes it is useful to override a parameter without needing to change the model. I have implemented a re-usable module, ModelTransformations, for making global changes to models. CSV-AutoSolver uses ModelTransformations, to provide a 'remove constraint' command line option. The option takes a variable name (either as a number or a variable name annotation), and searches the model for an equation with that variable by itself on one side, removing the first matching equation found. The 'add constraint' option takes a variable name and a real number; it adds in a new equation equating the variable to that constant value. The 'add constraint' and 'remove constraint' options can be used multiple times.

Debugging support is also provided; an option is provided to display the generated intermediate C code, or to run the model with profiling enabled to diagnose performance problems.

The basic time-course simulation allows users to request that every step be returned, and to select a minimum reporting step size. When the simulation is run, the output includes a header naming all variables, followed by a row for each reported time step, with the values of all state variables listed, along with the bound variable.

To perform a sensitivity analysis, the model must either be missing a constraint on the variable over which the sensitivity analysis is to be performed, or alternatively, the constraint must be removed with the 'remove constraint' command line option. When performing a sensitivity analysis, a command line option is used to specify the variable over which to perform the sensitivity analysis. Additionally, command line arguments must be used to specify the lower and upper bounds for the variable in the sensitivity analysis, as well as the step size. The sensitivity analysis runs the simulation at every value between the lower and upper bounds, at increments of the step size. The arguments can be repeated to perform a multiple variable sensitivity analysis; in this case, every combination of the variable values in range are tried (causing an exponential increase in the number of points as the number of variables increase). The output is a CSV file with a header giving variable names, and rows giving the value of each state variable at the end time point for each sensitivity variable

Listing 9.3: The CoreOnly example model

```
-- + Require ModML-Core
module CoreOnly
where
import ModML.Core.BasicDAEModel

model = buildModel $ do
  x <- newNamedRealVariable "x"
  y <- newNamedRealVariable "y"
  x `newEq` boundVariable
  initialValue 0 y 1
  (derivative y) `newEq` x
```

step.

9.5 Demonstrating ModML on a real model

A simple valid ModML model is given in Listing 9.3.

This model is the ModML Core equivalent of:

$$x(t) = t \tag{9.27}$$

$$y(0) = 1 \tag{9.28}$$

$$\frac{dy}{dt} = x \tag{9.29}$$

This model has an analytic solution:

$$x(t) = t \tag{9.30}$$

$$y(t) = 0.5t^2 + 1 \tag{9.31}$$

Numerical results for this simple model were obtained using the solver described in subsection 9.4.

```
csv-autosolver --starttime=0 --endtime=10 -c=CoreOnly.
    csv ./CoreOnly.hs
```

The resulting CSV file contains results that closely match the analytic solution, as shown in Figure 9.5.

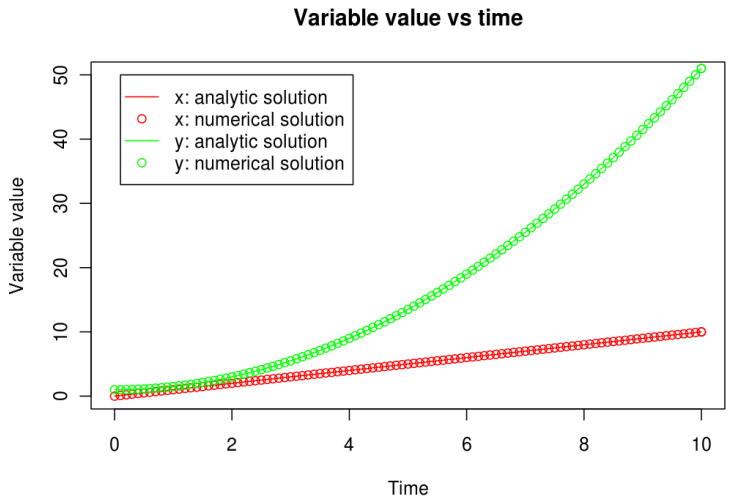

Figure 9.5: A comparison of the analytic and numerical solutions to the simple 2-parameter model discussed in the text.

9.6 Conclusions and future directions

The ModML Core provides basic facilities for describing systems of differential-algebraic equations, and allows modules that can extend ModML Core. ModML Solver provides both a library and an application for solving ModML models.

While ModML Core separates the solving algorithm for systems of DAEs from the equations, it does not, by itself, add very much in the way of predefined functions. This minimalistic design means that modellers will greatly benefit from extension libraries that build useful facilities on top of the core. Some such libraries have been developed in the course of this research and are discussed in Chapter 10. However, the utility of ModML will be greatly enhanced when more library support, specific to many different domains of application, is developed. The great advantage of the ModML formalism is that because these libraries would form part of the model, and not part of the ModML Core, they can be implemented without changing existing solvers.

Further future directions could, however, also include improvements to the solvers available for ModML. The existing ModML Solver library makes use of

IDA to perform numerical solving. However, it uses IDA in sequential mode, and so can only make use of a single processor core. The parallelism that can be achieved while solving a single DAE initial value problem using current algorithms is limited because computing the results after each time-step depend on the results from the previous time-step, forcing the sequential evaluation of each time-step. However, some limited parallelism is possible within time-steps; for example, some of the matrix operations needed to solve models can be divided up into parallel tasks; this level of parallelism is supported by IDA. Much greater parallelism can be obtained when versions of a model with slightly different parameters need to be evaluated (as in the sampling based sensitivity analyses described above); this problem is embarrassingly parallel because each simulation is performed independently in a different thread of execution, and the results are collated at the end.

Other future work could include support for more sophisticated symbolic manipulation, similar to that done by existing computer algebra software, such as Macsyma [Rand, 1984], Maxima [de Souza et al., 2003] and Mathematica [Wolfram, 1988]. This symbolic manipulation could be used both for more advanced simplification, and in some cases, to solve models analytically and avoid the need for a numerical solution entirely.

The approach used in ModML Core is limited to a specific data model, for systems of differential-algebraic equations. Future directions could include using a similar approach to that taken in ModML Core to represent more general problems. For example, a similar approach could be used to represent partial differential equation problems, for use with a PDE solver. The data structures would need to be extended to describe partial differential equations, as well as more complicated boundary conditions.

Chapter 10

Building domain specific languages on top of ModML

10.1 Introduction

ModML Core (described in Chapter 9) provides the basic infrastructure for describing systems of differential algebraic equations. However, simply writing ModML Core models as in Listing 9.3 means models cannot be described in more domain-specific terms, as in languages like CellML and SBML. For example, both CellML and SBML can check that the units in a model are consistent, but ModML Core lacks units altogether.

In this chapter, I introduce two modules that ModML models can import to allow modelling in a way natural to particular problem domains. The first is useful for any domain of application where physical units apply. The second allows for descriptions of systems as compartments, entities, and processes, and so can be applied to, for example, systems of chemical or biochemical reactions.

10.2 ModML Units

Physical units can be attached to many of the variables and constants in mathematical models of physical systems (including biological systems). There are basic rules and constraints on how units interact with comparison and arithmetic operators, creating the possibility of units-inconsistent models.

For example, suppose that x and z are in metres, and y is in seconds. Then the equation $z = x \times y$ is units (and dimensionally) inconsistent. This inconsistency

would be avoided if the units on z were metre.second.

However, information on models is useful for two major reasons; firstly, it allows the values of variables in the model to be interpreted correctly. Secondly, it allows the model to be checked against the units rules, a process that will often detect coding errors in a model.

Many existing model representation languages (for example, CellML and SBML) therefore include a mechanism for representing units and associating them with variables and constants.

For example, in CellML, the units millimolar/s can be declared as:

```
<units name="millimolar_per_second">
  <unit units="mole" prefix="milli"/>
  <unit units="litre" exponent="-1"/>
  <unit units="second" exponent="-1"/>
</units>
```

It is important to note, however, that some functions are useful if expressed in a way that is units consistent without needing to specify the exact units. For example, a function that takes a variable number of parameters and returns the geometric mean of the parameters is usefully generic provided all the parameters have the same units, and so it would be inconvenient if the function needed to be rewritten for every units it was required to work with. The approach taken in ModML Units provides a solution to this problem, as discussed below.

As with expressions, it is useful to build complicated units through the repeated combination of simpler units, using multiplication and exponentiation of constants. For example, one might define the joule according to the standard SI definition as $\text{kilogram} \times \text{metre}^2 \times \text{second}^{-2}$, and then the watt as $\text{joule} \times \text{second}^{-1}$. This functionality is supported by CellML and SBML.

Functions of units

In addition, it is useful to be able to perform functional transformations on constants or units to produce units. For example, nth order rate constants can be defined in terms of n as $s^{-1}(mol.L^{-1})^{-n}$. This functionality is not provided by CellML or SBML but is provided by ModML.

Another possibility is to allow units to depend on state variables or the bound variable. The cost of this level of generality, however, is that it makes it no longer possible to determine a single set of units that a particular variable will always have, and it is only possible to determine if the units are consistent, in general, at the time when a simulation is run. These drawbacks can be considered analogous to the problems associated with dynamic typing. Dynamic units would also require changes to ModML Core. Due to the drawbacks of dynamic units, the remainder of this chapter focuses on statically determinable units.

Overview of ModML Units

In the ModML module discussed in this chapter, a transformation from declarations with units to ModML Core is defined. As this transformation occurs before any simulation results are generated, the successful termination of the transformation guarantees that the units on each variable can be computed and are independent of the bound-variable and all state variables that vary with respect to the bound variable.

ModML Units defines a series of data structures, analogous to ModML Core, but with minor differences to allow variables and constants to be annotated with units, as well as additional data structures to describe the units. The top-level data structure is called UnitsDAEModel, and is the analogue to BasicDAEModel.

Unit representation

Base units are represented by the BaseUnit type, which has a single constructor of the same name, and carries an integer to uniquely identify the base unit. The semantics of a base unit are that they are the most fundamental description of a unit of measurement; for example, the SI units module discussed below defines base units for the seven SI base units.

Units are represented by the Units type, which again has one constructor, Units, and takes a Double, representing the multiplier, and a map from a BaseUnit to a Double, representing the BaseUnits and their respective exponents making up the units. The multiplier represents a quantity by which a value should be multiplied to get the value in terms of the base units with the exponents applied. For example, if the base units for metre had identifier 1, then the following could represent millimetre:

```
Units 0.001 (Data.Map.fromList [(BaseUnit 1, 1.0)])
```

Expressions with units

The ModML Units RealExpression data type also includes two special purpose constructors not shared with ModML Core. The first is called a units assertion. Functionally, it is equivalent to the identity function over a RealExpression argument in the final model. However, it also takes a Units argument. During the conversion from ModML Units to ModML Core, if the units on the RealExpression argument don't match the specified units, an error is raised. This functionality is useful because a model may contain complicated expressions generated over many lines of code, and knowledge that there is a units error somewhere in an expression of such complexity does not necessarily allow the problem to be rapidly isolated. Using units assertions, assumptions about the units of an expression can rapidly be checked.

Another special facility provided is a constructor of two RealExpression arguments, x and u. The constructor is treated as having the value of x, but the units of u. While the widespread use of this operator should be discouraged, as

it can be used to circumvent the tests for units and dimensional consistency, it is provided for special cases where the standard units inference rules are inadequate. For example, they could be used together with a new base unit for pH and special functions to convert between $[H^+]$ and pH.

ModML Units ModelBuilder monad

As with ModML Core, ModML Units provides a ModelBuilder monad to facilitate the construction of models with units, and the more general ModelBuilderT monad. As in the previous chapter, I shall present functions in terms of ModelBuilder even though the actual implementation is described in more general terms using ModelBuilderT. Note that the ModelBuilder here, ModML.Units.ModelBuilder, is a distinct type to the previously described ModML.Core.BasicDAEModel.ModelBuilder; if both need to be referred to, qualified types must be used. In this section, ModelBuilder is used to refer to ModML.Units.UnitsDAEModel.ModelBuilder, and B.ModelBuilder is used to refer to ModML.Core.BasicDAEModel.ModelBuilder.

When building a model with units, the units on the independent (bound) variable are often important. For example, suppose a model includes an equation with the derivative of a variable with respect to the bound variable on one side, and a constant on the other. To units check the equation, it is necessary to determine the units of the derivative. The UnitsDAEModel structure carries the units on the variable, but the units on the derivative are the units on the variable divided by the units on the bound variable. The ModML Core B.ModelBuilder is a wrapper around a state monad storing B.BasicDAEModel; one option would be to make UnitsDAEModel type carry the units for the bound variable. However, treating the units on the bound variable as state would be less than ideal, because it would allow the units to be updated multiple times throughout the state monad sequencing for the model. Instead, I have made use of a different standard monad, the Reader monad. The Reader monad allows a value to be fetched, but not updated[1]. The ModelBuilderT monad transformer stack includes both the State and Reader monads, and the ModelBuilderT monad

[1] other than by using a function called local to isolate a temporary change

Listing 10.1: Declaring a named and tagged base unit

```
data MyUnitBaseTag = MyUnitBase deriving (D.Typeable, D.Data)
myUnitBaseTag = D.typeCode MyUnitBaseTag
myUnitBase = newNamedTaggedBaseUnit myUnitBaseTag "myUnit"
```

implements the standard MonadReader and MonadState type classes. However, to access the bound variable units in ModML Units, it is more idiomatic to use the more self-evidently named function boundUnits.

Monad operations are provided to create new base units and variables, along with type-tagged operators to create base types or variables if they don't already exist, and otherwise to use the existing registered one. As with variables in ModML Core, this additionally allows libraries of units and expressions with units to be built up.

Allocating units

The newBaseUnit function works in a similar way to the newRealVariable function described in chapter 9; it has type:

```
ModelBuilder BaseUnit
```

ModML Units provides a function that also records name meta-data, newNamedBaseUnit, a tagged one, newTaggedBaseUnit, and one that combines naming and tagging, newNamedTaggedBaseUnit. This can be used to define a BaseUnit that can be reused from multiple places as shown in Listing 10.1.

The first time myBaseUnit is sequenced in the ModelBuilder monad, it will be allocated; subsequent appearances will cause the same base unit to be retrieved by the tag.

The syntax required to create a type-tagged base unit is both verbose and unintuitive, so it would clearly be desirable to provide syntactic sugar around it. Template Haskell is an extension to Haskell that is supported by GHC. It allows Haskell code to be written that is run to generate data structures representing Haskell code, which are in turn compiled. ModML Units includes Template

Haskell functions for generating named, type-tagged base units and variables. The function declareBaseType can be used as a short-hand for Listing 10.1.

```
U.declareBaseType "myUnit" "myUnitBase"
```

Operations on units

ModML Units provides a number of functions that can be used to build up Units structures from other Units. The function unitsMultiplier builds a dimensionless Units from a double as the multiplier. The unitsTimes function computes the Units that would apply if the two Units arguments were multiplied. It has type:

```
Units -> Units -> Units
```

The final fundamental Units combining function is unitsPow; it computes the Units that would be found on a value with the Units specified in the argument raised to a constant power. It has type:

```
Units -> Double -> Units
```

The monadic versions of these functions are more useful. Functions from monadic type to monadic type are named, as in ModML Core, with an X suffix. The monadic form of unitTimes is unitTimesX, and the monadic form of unitsPow is unitsPowX. In addition, operator forms are provided; ($*$) is an alias for unitTimesX, and ($**$) is an alias for unitsPowX. Operators surrounded by $ are used to combine units, while operators surrounded by . are used to combine expressions.

The type BaseUnit is distinct from the type Units; the value myUnitBase of type ModelBuilder BaseUnit defined in Listing 10.1 cannot immediately be used as a ModelBuilder Units. The function singletonUnits takes a BaseUnit, and builds a unit made up of only that BaseUnit, with a multiplier of one and an exponent of one. The monadic form singletonUnitX is also provided. The following listing demonstrates such usage:

```
myUnit = singletonUnitX myUnitBase
```

Translating ModML Units to ModML Core

For ModML Units to be useful, it needs to be translated into ModML Core, and additionally, there needs to be a mechanism to check that the units are correct.

As discussed earlier, the translation from ModML Units to ModML Core is only well-defined if the units on the independent (bound) variable are specified; for example, these units might be second when the bound variable is time. However, the bound units are likely to be drawn from a library that builds the units up from named, tagged, base units. This creates a chicken-and-egg problem; to translate from type ModelBuilder a to B.ModelBuilder a, it is necessary to translate from ModelBuilder Units to B.ModelBuilder Units. However, the problem can be circumvented if we assume that there is no need to know the true bound variable units to compute these units. This transformation is shown in Figure 10.1.

The runInCore function has type:

```
ModelBuilder Units -> ModelBuilder a -> B.ModelBuilder a
```

The function firstly builds an initial UnitsDAEModel from the current B.BasicDAEModel as follows: annotations are copied directly, as is the nextID and contextTaggedIDs. All of the remaining constructor arguments are initialised to empty lists. The first parameter is converted from ModelBuilder Units to B.ModelBuilder Units, by running the Reader monad, using a dimensionless units type, and the State monad beginning with the initial UnitsDAEModel discussed above. This means that any attempt to access boundUnits in the part of the code that builds the bound variable units will yield dimensionless units.

The initial UnitsDAEModel may have been updated in the State monad. The second argument is run through the Reader monad, with the newly determined bound variable units, and the updated UnitsDAEModel, yielding the result from runInCore.

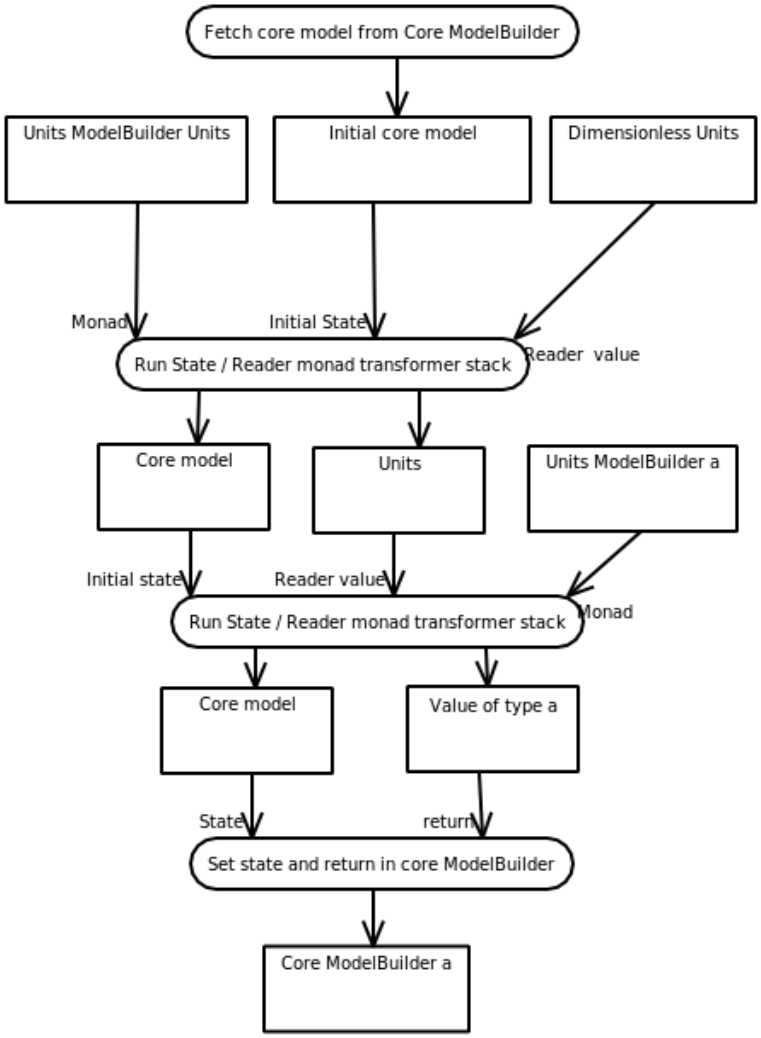

Figure 10.1: The transformation from a ModML Reactions ModelBuilder to a ModML Core ModelBuilder

The runInCore function is then used as a primitive in the translation to ModML Core. The unitsToCore function uses runInCore, but additionally translates the resulting data structure at the end of the ModML Units evaluation back into the ModML Core data type, and finally stores it using the ModML Core builder. Modellers can either use this function, if they are building a model that combines parts using ModML Units and parts using ModML Core, or if this isn't necessary, they can use the convenience function buildModel, a function that composes unitsToCore with B.buildModel, giving an overall type of:

```
buildModel :: ModelBuilder Units -> ModelBuilder a -> B.
    BasicDAEModel
```

The key part of the translation occurs in mergeIntoCoreModelWithValidation, called from unitsToCore. This function converts to ModML Core, raising exceptions if the units are incorrect. The nextID and contextTaggedIDs are simply copied unchanged constructor arguments with the same names in the BasicDAEModel. The remainder of the model is recursively translated using translation and checking rules encoded in a series of co-recursive functions. The function translateRealExpression has type:

```
RealExpression -> ModelBuilder (Units, B.RealExpression)
```

It computes the Units of an expression as it converts the expression and checks the units. These Units are then used at the next level of recursion. For example, two steps are taken to translate:

```
a `Plus` b
```

Firstly, a and b are recursively translated. If the units on each of the arguments don't match, an exception is raised. Otherwise, a B.RealExpression with the translated arguments is produced.

ModML Units copies all annotations with a subject referencing the ModML Units variable to form a similar annotation that instead references the core variable. The original annotations are also retained.

The SI Units module

The ModML Units package includes a module called ModML.Units.SIUnits. The module provides ModelBuilder BaseUnit for all seven base units defined in the International System of Units (SI; Taylor and Thompson [2001]). For example, metreBase provides the base unit in metres. SIUnits additionally provides ModelBuilder Unit definitions for all SI base and derived units, as well as for litre (litre is not an official SI derived quantity, but it is mentioned in Taylor and Thompson [2001]).

The SIUnits module can be imported into any number of models that also import ModML Units. Because type-tagged base units are used, if sub-models that each individually import the SIUnits module are imported into a top-level model and composed, each base unit will have the same meaning in all of the sub-models, allowing the sub-models to be composed with units checking treating, for example, each separately imported instance of metreBase the same.

Physical constants in ModML Units

Physical constants are directly or indirectly relevant to numerous processes in many different areas of application.

I have created ModML.PhysicalConstants.Common, a ModML library of common physical constant definitions and their associated units, using ModML Units. The list of supported constants was taken from the list of physical constants collated on the Wikipedia page Physical Constants, as of the 15th of February, 2011, and checked according to Mohr et al. [2008], the source of most of the constant values.

Where possible, physical constants are written as an expression of other physical constants, so only a few fundamental constants are given directly as a value. In addition, a number of agreed exact standard values (such as standard gravity in $m.s^{-2}$, and the standard atmospheric pressure in Pascals) are also provided. A total of 32 direct and derived constants are provided.

Physical constants are given short descriptive plain-text names rather than their commonly-used symbols.

The text of the physical constants model is given in Listing 10.2, and demonstrates how expressions can be built up using ModML Units. Notice that in the listing, derived constants can be defined in terms of other constants, and units are computed rather than needing to be specified on derived constants.

10.3 ModML Reactions

CellML, SBML, and little b are all used to represent biochemical reaction networks. While it was a deliberate design choice to not incorporate domain-specific data such as reaction networks into ModML Core, it is useful to describe models as reaction networks, using a module that can translate this description into ModML Units, and so ultimately ModML Core. In this chapter, I describe the design of ModML Reactions, a module designed to achieve these goals.

Definitions of key concepts

ModML Reactions introduces a few basic definitions. An entity is the basic participant, and could represent a chemical species, or could represent something like heat energy. A compartment is a place where entities can be found; there could, for example, be a single reactor vessel compartment, or several compartments, representing the physical compartments of cells. A compartment-entity is the simple pairing of a compartment and an entity to describe a particular entity in a particular compartment.

The amount, in ModML Reactions, is defined as a measure of how much of an entity is present in a compartment at a point in time. Each entity always has the same units on all amounts, but different entities may have different units on amounts (if a chemical species needs to be measured in different units in different contexts, for example, between the free and membrane bound states, a different entity can be used for each context).

Listing 10.2: Defining physical constants with ModML Units

```
{-# LANGUAGE NoMonomorphismRestriction #-}
module ModML.PhysicalConstants.Common
where
import qualified ModML.Units.UnitsDAEModel as U
import ModML.Units.UnitsDAEOpAliases
import ModML.Units.SIUnits

speedOfLightInVacuum = U.realConstant (uMetre $*$
    uSecond $**$(-1)) 2.99792458E8
gravitationalConstant = U.realConstant (uMetre$**$3 $*$
    uKilogram$**$(-1) $*$ uSecond$**$(-2) ) 6.6742867E-11
planckConstant = U.realConstant (uJoule $*$ uSecond)
    6.6260689633E-34
reducedPlanckConstant = planckConstant ./. (U.piX .*. U.
    realConstant U.dimensionless 2)
magneticConstant = U.piX .*. (U.realConstant (uNewton $*
    $ uAmpere$**$(-2)) 4E-7)
electricConstant = (U.realConstant U.dimensionless 1)
    ./. (magneticConstant .*. speedOfLightInVacuum .*.
    speedOfLightInVacuum)
impedanceOfFreeSpace = magneticConstant .*.
    speedOfLightInVacuum
coulombsConstant = (U.realConstant U.dimensionless 0.25)
     ./. (U.piX .*. electricConstant)
elementaryCharge = U.realConstant uCoulomb 1.60217648740
    E-19
bohrMagneton = elementaryCharge .*. planckConstant ./. (
    U.realConstant U.dimensionless 2 .*. electronMass)
conductanceQuantum = U.realConstant U.dimensionless 2
    .*. (elementaryCharge .*. elementaryCharge) ./.
    planckConstant
inverseConductanceQuantum = U.realConstant U.
    dimensionless 0.5 .*. planckConstant ./. (
    elementaryCharge .*. elementaryCharge)
josephsonConstant = U.realConstant U.dimensionless 2 .*.
     elementaryCharge ./. planckConstant
magneticFluxQuantum = U.realConstant U.dimensionless 0.5
     .*. planckConstant ./. elementaryCharge
nuclearMagneton = elementaryCharge .*. planckConstant
    ./. (U.realConstant U.dimensionless 2 .*. protonMass)
vonKlitzingConstant = planckConstant ./. (
    elementaryCharge .*. elementaryCharge)
bohrRadius = fineStructureConstant ./. (U.realConstant U
    .dimensionless 4 .*. U.piX .*. rydbergConstant)
```

Listing 10.3: Defining physical constants with ModML Units (ctd)

```
classicalElectronRadius = elementaryCharge .*.
    elementaryCharge ./. (U.realConstant U.dimensionless
    4 .*. U.piX .*. electricConstant .*. electronMass .*.
     speedOfLightInVacuum .*. speedOfLightInVacuum)
electronMass = U.realConstant uKilogram 9.1093821545E-31
fineStructureConstant = magneticConstant .*.
    elementaryCharge .*. elementaryCharge .*.
    speedOfLightInVacuum ./. (U.realConstant U.
    dimensionless 2 .*. planckConstant)
hartreeEnergy = U.realConstant U.dimensionless 2 .*.
    rydbergConstant .*. planckConstant .*.
    speedOfLightInVacuum
protonMass = U.realConstant uKilogram 1.67262163783E-27
quantumOfCirculation = planckConstant ./. (U.
    realConstant U.dimensionless 2 .*. electronMass)
rydbergConstant = fineStructureConstant.**.(U.
    realConstant U.dimensionless 2) .*. electronMass .*.
    speedOfLightInVacuum ./. (U.realConstant U.
    dimensionless 2 .*. planckConstant)
thomsonCrossSection = (U.realConstant U.dimensionless
    (8.0/3)) .*. U.piX .*. classicalElectronRadius.**.(U.
    realConstant U.dimensionless 2)
avogadrosNumber = U.realConstant (uMole $**$ (-1))
    6.0221417930E23
boltzmanConstant = gasConstant ./. avogadrosNumber
faradayConstant = avogadrosNumber .*. elementaryCharge
gasConstant = U.realConstant (uJoule $*$ (uMole $*$
    uKelvin) $**$ (-1)) 8.31447215
molarPlanckConstant = avogadrosNumber .*. planckConstant
standardGravity = U.realConstant (uMetre $*$ uSecond$**$
    (-2)) 9.80665
standardAtmosphere = U.realConstant (uPascal) 101325
```

A process describes an influence on the amounts of one or more entities in compartments. ModML Reactions has been designed so that processes can be given in a library along with large numbers of other processes, and only the processes that actually occur in a given compartment should have an impact on the marginal real time taken for each unit of simulation time. This is achieved by using an iterative algorithm, called the Process Activation Algorithm, described later in this section. The algorithm converges on a set of compartment-entities that the algorithm failed to find were provably zero at all time points. The algorithm used is conservative; a value that is not provably zero may in fact

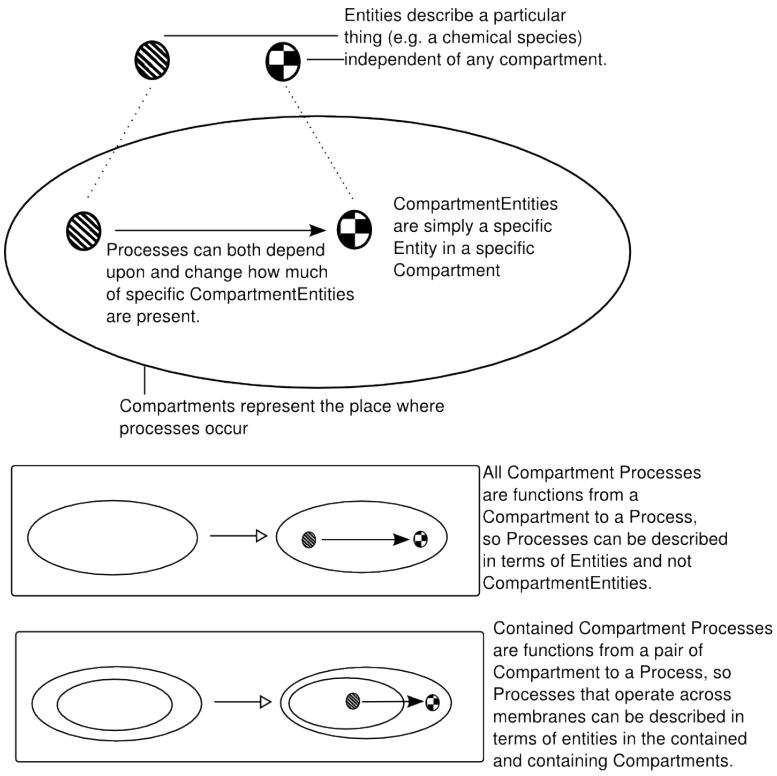

Figure 10.2: A visual overview of the key concepts in ModML Reactions.

be zero, but a value that is provably zero cannot be non-zero at any point in time (provided that the modeller has ensured that the assumptions ModML Reactions makes about the process hold true).

These concepts are shown in Figure 10.2.

Compartment and Entity structures

Compartments are represented by a data type, Compartment, with a single constructor of the same name. This constructor takes an Int argument, representing the unique identifier for the Compartment. An entity is represented by the Entity type, which also has a single constructor, Entity, with two arguments; a U.Units (where U is ModML.Units.UnitsDAEModel) and an Int as unique identifier. The Units represents the units of measurement used on the amounts

for that entity. Compartment-entities are represented by the simple pairing of an Entity with a Compartment; the type alias CompartmentEntity can be used to refer to this paired type.

Process structures

The process data structure has a single constructor, Process, that takes multiple named arguments. The Process constructor has an argument, processId, that uniquely identifies the process. The constructor has an argument, activationCriterion, that is a function from a set of CompartmentEntities to a Boolean, with the semantics that when all CompartmentEntities necessary for a process to occur are in the Set, the function should evaluate to True. The following restriction applies: if the function is true for set S_1, it must also be true for S_2, if $S_1 \subseteq S_2$; this restriction is relied upon by the ModML Reactions to ModML Units translation. In addition, there is an underlying assumption that processes can only be activated if the required CompartmentEntities for activation exist strictly before the process becomes active. This assumption means, for example, that if there are two processes, A and B, where A produces CompartmentEntity C from D, and B produces CompartmentEntity D from C, where C and D have initial amounts of zero, and no other processes produce C or D, that processes A and B will not be active.

In addition, the Process constructor includes two set arguments, modifiableCompartmentEntities and creatableCompartmentEntities. The semantics are that modifiableCompartmentEntities are entities that can be modified by the process, and creatableCompartmentEntities are CompartmentEntities that can be created (that is, given that the Process has been activated, all of the CompartmentEntities listed in the creatableCompartmentEntities set are not provably zero at all times).

The Process data structure describes the rate at which the process occurs. To achieve this, it is necessary to refer to the amounts of relevant CompartmentEntities present. However, ModML Reactions handles the allocation of variables, and so when building the data structure, the modeller does not

have access to these variables. Instead, the Process constructor includes an argument, entityVariables, which is a Map from a place-holder U.RealVariable to a CompartmentEntity. To refer to the amount of a CompartmentEntity, a modeller allocates a place-holder variable and includes it in the map. Another argument of the Process constructor, rateTemplate, of type U.RealExpression, describes the rate of the process in terms of the place-holder variables (and possibly other variables).

To allow the actual fluxes on each CompartmentEntity to be determined, the Process constructor also includes a Map from CompartmentEntities to a pair of a Double and a U.Units. This pair describes a constant stoichiometry multiplier, with units, to multiply the process rate by to get the individual flux, due to the process, for the CompartmentEntity. The product of a reaction would normally have positive stoichiometry, and a reactant would normally have negative stoichiometry.

The top-level data type in ModML Reactions is called ReactionModel. The data structure has a single constructor of the same name. Like the corresponding top-level structures in ModML Core and ModML Units, the structure contains a nextID field and maps for annotations and contextTaggedIDs. These fields play an analogous role in ModML Reactions to in ModML Core.

Some processes may genuinely only apply to specific entities in specific compartments. The ReactionModel constructor includes an argument explicitCompartmentProcesses, of type [Process] (i.e. a list of Processes).

However, more commonly, a process will occur wherever the necessary ingredients are present; if the right substrates, enzymes, and activators are in a compartment, the process can be expected to occur. To support this case, the BasicReactionModel constructor includes an argument, allCompartmentProcesses, of type:

```
[Compartment -> Process]
```

This type is a list of functions from compartments to processes. During translation to ModML Units, ModML Reactions will evaluate the function for all

Compartments, and the resulting process is put through the standard process activation algorithm. To allow this to happen, the ReactionModel includes an argument, compartments, a set of all Compartments used in the model.

Contained compartment processes

In addition, some processes (for example, those mediated by a trans-membrane spanning protein) may apply between any two compartments, provided the correct entities are present in the two compartments, and one compartment is directly physically contained in the other. To support this, the BasicReaction-Model constructor includes an argument called containedCompartmentProcesses, of type:

```
[(Compartment, Compartment) -> Process]
```

Each containedCompartmentProcess member generates a process for each containment relationship. To allow this to occur, ModML Reactions need to know which compartments are contained in which other compartments. To support this, the ReactionModel constructor includes an argument, compartments, with the type of a Set of (Compartment,Compartment) pairs. The semantics are that each entry in the Set describes a containment relationship stating that the second Compartment is contained within the first.

Entity instances

Models using ModML Reactions do not exist in isolation from the rest of the model. A reaction network may only form a small part of a model, describing the influence of the reactions on the system. The ModML Reactions part could potentially both contribute to part of a larger model, and be influenced by variables not controlled by processes and entities. There are several mechanisms in ModML Reactions to support this. Approaches for using output from the ModML Reactions part of the model in other parts of the model are discussed below as part of the discussion on translating from ModML Reactions to ModML Units, while approaches for taking inputs to the ModML Reactions part are

discussed here. The simplest case is where an external part of the model (for example, the temperature of the system) affects the rates of processes. In this case, because process rate templates are described using a RealExpression from ModML Units, ModML Units variables can simply be used directly in these expressions.

However, there may also be external influences on particular CompartmentEntities. For this reason, the ReactionModel constructor includes an argument, entityInstances, which is a map from a CompartmentEntity to an EntityInstance. EntityInstance is a data type with two constructors that describes how a CompartmentEntity interacts with the rest of the model. There are two constructors. The first, EntityClamped, takes a U.RealExpression argument, and indicates that the amount of a CompartmentEntity present is never updated by any processes, but instead has a value determined by that U.RealExpression. The second constructor, EntityFromProcess, tells ModML Reactions that it should allocate a variable for the amount of the CompartmentEntity (unless it is proved that the value is always zero), and includes two U.RealExpression arguments. The first argument specifies the initial value for the variable at time zero; it can include arbitrary computations. The second argument specifies a value to add to the fluxes. When no EntityInstance is present for a given CompartmentEntity, a default equivalent to EntityFromProcess (U.RealConstant0) (U.RealConstant0) is used.

Translating from ModML Reactions to ModML Units

For ModML Reactions to be used in ModML models, there needs to be a way to incorporate the reaction networks into models. This is achieved by a translation mechanism that can build a ModML Units expression from a ModML Reactions sub-model. The function reactionModelToUnits is a function from a ReactionModel to a ModML Units builder. In this section, the qualified name U refers to ModML.Units.UnitsDAEModel, B refers to ModML.Core.BasicDAEModel, and M refers to Data.Map. Using these conventions, and continuing the convention of describing the specialised U.ModelBuilder type and not the more general U.ModelBuilderT transformer type, the type of reactionModelToUnits is:

```
reactionModelToUnits :: ReactionModel -> U.ModelBuilder
    (M.Map CompartmentEntity U.RealExpression, M.Map
    Process U.RealExpression)
```

The semantics are that the function translates the ReactionModel into a ModML Units ModelBuilder, returning some supplementary information: a map from all active CompartmentEntities to the corresponding RealExpression for the amount of that CompartmentEntity, and a map from each active Process to the expression for the rate of that Process. The returned supplementary information can either be ignored, or used to connect the variables from the ReactionModel to other parts of the model.

The first step of this translation is to perform the Process Activation Algorithm, which was described in more detail previously. This computes two sets, of active processes and active CompartmentEntities.

Next, each active CompartmentEntity is given a corresponding variable in the model, representing the amount of that CompartmentEntity present at each time. It is important that these variables are annotated, because otherwise the modeller would not know which variables correspond to which CompartmentEntities. The name of the entity and compartment is fetched from the annotations by checking the nameIs annotations. If the names are not found, "Unknown Entity" or "Unknown Compartment", respectively, are used instead. These names are used to build an annotation for the variable in the form of "Amount of Entity Name in Compartment Name".

Each active Process is given a corresponding variable, for the rate of the Process. The rateTemplates in each Process are substituted to replace the placeholder variables with the newly allocated variables for each CompartmentEntity, and an equation is added to equate the new variable for the Process rate to the rate expression.

In addition, each active CompartmentEntity is given an equation equating the initial value of the CompartmentEntity amount variable to the initial value on the EntityInstance (or the default of zero if there is no EntityInstance), and

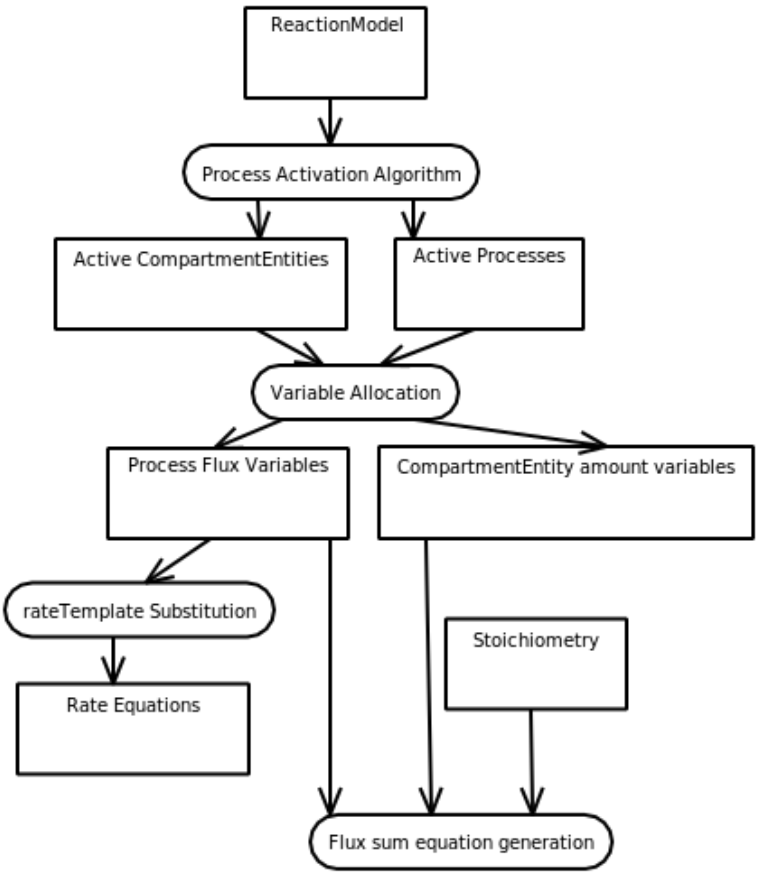

Figure 10.3: A flow diagram showing the overall process taken to convert a ModML Reactions ReactionModel a ModML Units ModelBuilder.

equating the derivative of the variable to the sum of the fluxes (Process rate variables times stoichiometry for the CompartmentEntity in the process), plus flux contribution from the EntityInstance (if any). The overall process is shown in Figure 10.3.

The process activation algorithm

As discussed earlier, one of the benefits of the ModML Reactions DSL over direct representation with ModML Units is that unnecessary processes can sometimes be eliminated from the model automatically; this allows the modeller to describe processes that could happen in any compartment if the necessary

entities are present (i.e. have non-zero amount at some point in time). However, it is important that processes or entities are not eliminated if there is any possibility that they could have an impact on the results of the model.

Let S be the set of all CompartmentEntities in the model M. This set could theoretically be partitioned into two complementary subsets, S_0, the set of all CompartmentEntities that have amount zero at all times $t \in \mathbb{R}, t \geq 0$, and $S_1 = S \backslash S_0$, the set of all CompartmentEntities for which the amount is non-zero at any point in time $t \in \mathbb{R}, t \geq 0$.

Theorem 10.3.1. *Splitting S into S_0 and S_1 is not computable in general.*

Proof. Recall from the introduction to chapter 9 that systems of DAEs generalise systems of ODEs. Branicky [2002] showed that systems of ODEs can simulate Turing machines. Suppose, for the purposes of contradiction, that there was an algorithm that could split S into S_0 and S_1. By building a DAE that simulates a Turing machine, with a variable for each cell on the tape, it would be possible to determine which cells in the computation were zero at all non-negative times and which were non-zero at least some non-negative point in time. To prove the theorem, it is therefore sufficient to prove that it is not possible, in general, to compute which cells of a Turing machine are zero at all non-zero times.

Suppose, for the purposes of contradiction, that there was an algorithm that could compute, in finite time, which cells of a Turing machine are zero at all times. Then this algorithm could be used to determine whether or not a Turing program, P_0 would halt as follows: the program to determine the halting problem, P_1, would act as a virtual machine to run P_0. The cell numbers on the machine P_1 runs on would be remapped so that real cell 0 would not be addressable by P_0 (instead, virtual cell $i \in \mathbb{R}, i \geq 0$ is mapped to real cell $i + 1$). The initial value of real cell 0 would be zero, and the halting of P_0 would cause P_1 to set cell 0 to a non-zero value. Therefore, the value of real cell zero will be non-zero at some point in time if and only if P_0 halts, and so the algorithm introduced in this paragraph could be used to compute whether or not a program P_0 halts. However, this contradicts the proof [Turing, 1937] that it is not possible, in general, to compute whether or not a program halts.

It has therefore been shown, by contradiction, that the theorem holds. □

However, even though it is not possible to compute S_0 or S_1 from M exactly in all cases, being able to determine, for some cases, that a variable belongs to S_0 is still a useful optimisation, as entire processes can be determined to be inapplicable.

I present the process activation algorithm, and prove that at the termination of the algorithm, it selects a set of variables S_2 such that $S_2 \subseteq S_0$, and a set of processes P such that every process that influences a member of S_1 is included.

For convenience, let $S_3 = S \setminus S_2$. As S is known, S_2 can be trivially computed from S_3.

The algorithm proceeds iteratively through a series of steps. The total number of steps, n, is not known *a priori*. At each step i, the following state is kept: $P_{1,i}$, called the set of active processes, is a set of processes that evolves so $P_{1,i} \subseteq P_{1,i+1}$ and $P_{1,n} = P$. $P_{2,i}$ is called the new active process list, and is related to P_1 by $P_{2,i} = P_{1,i+1} \setminus P_{1,i}$. $P_{3,i}$ is called the set of candidate processes; processes are only ever removed from the set ($P_{3,i} \subseteq P_{3,i-1}$). The set $S_{4,i}$ is called the active CompartmentEntities set; it is only ever added to ($S_{4,i} \subseteq S_{4,i+1}$), and $S_{4,n} = S_3$. $S_{5,i}$, called the new active CompartmentEntity set, is related to S_4 as $S_{5,i} = S_{4,i} \setminus S_{4,i-1}$. S_6 is called the definite inactive CompartmentEntity set; below, it is proved that $S_6 \subseteq S_0$ holds.

Initially, all CompartmentEntity to EntityInstance pairings are analysed to place each CompartmentEntity into one of three bins; definitely in S_0 (to be stored in S_6), possibly in S_0 but definitely zero at time zero (not explicitly stored), and either indeterminate or definitely in S_1 (to be stored in $S_{4,0}$). ModML Units provides the facility, as part of the simplifier, to try to simplify an expression to a constant value. For example, for the RealExpressions corresponding to $4+5*3$, it will return Just 19, while for v_1+3 it will return Nothing because the expression cannot be simplified to a constant without information on variable v_1.

Where a CompartmentEntity has an EntityInstance from the EntityClamped

constructor, if the RealExpression the CompartmentEntity amount is clamped to simplifies to a constant zero, then the CompartmentEntity is definitely in S_0, by definition. If it is clamped to some constant value other than 0, then it is definitely in S_1, and so is placed in the third set ($S_{4,0}$). If it is clamped to a value that doesn't simplify to a constant, it is also placed in the third set ($S_{4,0}$); whether or not it is in S_0 or S_1 depends on parts of the model outside of ModML Reactions, and so the conservative nature of the algorithm means these CompartmentEntities will be carried through $S_{4,i}$ and end up in S_3.

Where a CompartmentEntity has an EntityInstance from the EntityFromProcesses constructor, if either the initial amount RealExpression or the external flux contribution doesn't simplify to a constant 0, the CompartmentEntity is placed in the third set ($S_{4,0}$), otherwise they are placed in the second (definitely zero at time zero) set.

The following additional starting conditions are set: $S_{5,0} = S_{4,0}$, $P_{0,0} = P_{1,0} = P_{2,0} = \emptyset$, and $P_{3,0}$ is set to the list of all Processes, formed as the union of the explicitCompartmentProcesses, the containedCompartmentProcesses evaluated over the set of containment relations, and the allCompartmentProcesses evaluated over the set of all Compartments.

The algorithm then proceeds through the iterative updates. The new active process list, $P_{2,i}$ is built by enumerating through all candidate processes in $P_{3,i-1}$ and testing whether the activation criterion for any of the processes have now been met given the previous set of active CompartmentEntities $S_{4,i-1}$; the set of such processes becomes $P_{2,i}$. Next, a temporary set $T_{1,i}$ containing the union of all CompartmentEntities that are creatable by any of the processes in $P_{2,i}$ is built. The set S_6 (definitely inactive CompartmentEntities) is subtracted from $T_{1,i}$ to give the updated new active CompartmentEntities set, $S_{5,i}$. Finally, to maintain the invariants presented above, the set of all active CompartmentEntities $S_{4,i}$ is computed as the union of the previous set of active CompartmentEntities and the set of new CompartmentEntities, $S_{4,i} = S_{5,i} \cup S_{4,i-1}$. Likewise, the active process list is updated as $P_{1,i} = P_{2,i} \cup P_{1,i-1}$. The new active processes are removed from the set of candidate processes, giving $P_{3,i} = P_{3,i-1} \setminus P_{2,i}$.

The algorithm terminates on the first i for which there is neither any new active Processes or any new active CompartmentEntities, i.e. $(S_{5,i} = \emptyset) \wedge (P_{2,i} = \emptyset)$. As this i is referred to as n, $S_{5,n} = \emptyset$ and $P_{2,n} = \emptyset$.

Theorem 10.3.2. *Provided that there are only finitely many Processes and CompartmentEntities in S, the algorithm described above will always terminate in finite time.*

Proof. Every step i is made up of a finite and fixed series of computations that either take a constant time, or iterate over sets of Processes and CompartmentEntities to perform a task that takes finite time. As each set of CompartmentEntities or Processes can only contain each CompartmentEntity or Process once, and there are finitely many CompartmentEntities or Processes, the sets are therefore of finite size. Iterating over a finite set to perform a task that takes a finite amount of time takes a finite amount of time, so it follows that each step takes finite time.

To prove that the algorithm terminates, it is therefore sufficient to prove that, under the assumptions of this theorem, that the termination condition will be met after a finite number of steps.

Because candidate processes are only ever removed from the set of candidate processes, $P_{3,i}$, the assertion $P_{3,i} \subseteq P_{3,i-1}$ holds. It is only possible for an element to appear in the set of new active processes $P_{2,i}$ if it was a candidate process in $P_{3,i-1}$. An active process is removed from the set of candidate processes. Therefore, either $|P_{2,i}| = 0$ or $|P_{3,i}| < |P_{3,i-1}|$. As $|P_{3,0}|$ is finite by the assumptions of the theorem, there can only be a finite number of times where $P_{2,i} \neq \emptyset$.

In addition, because $T_{1,i}$ is computed as a union over each member of $P_{2,i}$, $P_{2,i} = \emptyset \implies T_{1,i} = \emptyset$. But $S_{5,i} = T_{1,i} \setminus S_6$, so $P_{2,i} = \emptyset \implies S_{5,i} = \emptyset$. Therefore, there can only be a finite number of times when $S_{5,i} \neq \emptyset$.

Therefore, there can only be a finite number of times when $(\S_{5,i} \neq \emptyset) \vee (P_{2,i} \neq \emptyset)$, and so a finite number of times when the termination condition is not met. The termination condition is therefore met after a finite number of iterations,

each of which takes finite time, and so it has been proved that the algorithm terminates in finite time. □

Theorem 10.3.3. *A CompartmentEntity that is not in the final set of active CompartmentEntities $S_{4,n}$ has amount zero at all times, or in more formal terms,* $(x \notin S_{4,n}) \implies (x \in S_0)$.

Lemma 10.3.4. *If f is a function from $\mathbb{R}$ to $\mathbb{R}$, then at least one of the following conditions must hold. In the following conditions, $f(t) \neq 0$ is held to be true for t outside the domain of f, and $\frac{df(t)}{dt} \neq 0$ is held to be true for t outside the domain of $\frac{d}{dt}f(t)$.*

1. *For every $t \in \mathbb{R}$ where $t \geq 0$: $f(t) = 0$*
2. *$f(0) \neq 0$*
3. *For some $t \in \mathbb{R}, t \geq 0$, it holds that $\frac{df(t)}{dt} \neq 0$*

Proof. Suppose, for contradiction, that there was a function f such that none of the three conditions held. Then, by virtue of condition 3 being false:

$$\frac{df(t)}{dt} = 0 \text{ for every } t \in \mathbb{R}, t \geq 0 \tag{10.1}$$

By integrating both sides of equation 10.1, we get:

$$f(t) = c \text{ for every } t \in \mathbb{R}, t \geq 0 \tag{10.2}$$

By virtue of condition 2 being false, $f(0) = 0$, so $c = 0$. Therefore, $f(t) = 0$ for every $t \in \mathbb{R}, x \geq 0$.

However, this contradicts the assumption that condition 1 is false. Therefore, the lemma holds. □

Lemma 10.3.5. *Given that all CompartmentEntities that have non-zero amount at any time $t \in \mathbb{R}$, $0 \leq t < \tau$ are in the final set of active CompartmentEntities $S_{4,n}$, all CompartmentEntities that have non-zero value at any time $t \in \mathbb{R}$, $0 \leq t \leq \tau$ are also in the final set of active CompartmentEntities $S_{4,n}$*

Proof. Suppose, for contradiction, that there is a CompartmentEntity that has a non-zero value at some time $t \in \mathbb{R}$, $0 \leq t \leq \tau$ that the algorithm does not place in $S_{4,n}$. Let $f(t)$ be the function describing the amount of that CompartmentEntity with respect to time.

If $f(0) \neq 0$, then it follows that the initial value expression has some value other than an expression equivalent to constant zero. The algorithm places CompartmentEntities that have an initial value that does not simplify to constant zero in $S_{4,0}$. Because nothing is ever removed from the active CompartmentEntity set, $(x \in S_{4,i}) \implies (x \in S_{4,i+1})$, so by induction, $(x \in S_{4,0}) \implies (x \in S_{4,n})$. Therefore $(f(0) \neq 0) \implies (x \in S_{4,n})$. But we are considering a CompartmentEntity not in $S_{4,n}$, so therefore $f(0) = 0$.

If the the CompartmentEntity under consideration is not in S_0, then by the definition of S_0, there must be a $t \in \mathbb{R}, t \geq 0$ where $f(t) \neq 0$. We have shown two of the three conditions in lemma 10.3.4 to be false, and so the third condition must be true: for some $t \in \mathbb{R}, 0 \leq t \leq \tau$, it holds that $\frac{df(t)}{dt} \neq 0$.

Therefore, under the original supposition, $\frac{df(t)}{dt} \neq 0$, where $\frac{df(t)}{dt} = r(t) + \sum_i^l \Phi_i(t)$, and Φ_i is the flux on the CompartmentEntity due to the ith Process, l is the number of active Processes, and $r(t)$ is the flux contribution described externally to ModML Reactions. If $r(t) \neq 0$, then it follows that the initial value expression has some value other than an expression that simplifies to constant zero. But CompartmentEntities for which $r(t)$ does not simplify to constant zero are placed in $S_{4,0}$, and so by the same logic as above, $(\neg(\forall t.r(t) = 0)) \implies (x \in S_{4,n})$. As we are considering a CompartmentEntity not in $S_{4,n}$, $r(t) = 0$, and so $\frac{df(t)}{dt} = \sum_i \Phi_i(t) \neq 0$.

All Processes are initially placed in the set of candidate Processes, $P_{3,0}$, and Processes in $P_{3,i-1}$ are always in either $P_{3,i}$ or $P_{1,i}$, and a Process in $P_{1,i-1}$ is always in $P_{1,i}$ Therefore, $P_{3,n} \cup P_{1,n}$ contains the set of all Processes.

Suppose there was a Process p_i such that $\Phi_i(t) \neq 0$ for some $t \in \mathbb{R}, 0 \leq t \leq \tau$. By the definition of creatable CompartmentEntities, that Process would include the CompartmentEntity under consideration in the set of creatable CompartmentEntities. As $P_{3,n} \cup P_{1,n}$ contains all Processes, $(p_i \in P_{3,n}) \vee (p_i \in$

$P_{1,n}$). Suppose for contradiction that $p_i \in P_{1,n}$, that is, Process p_i is in the final set of active Processes. As $P_{1,0} = P_{2,0} = \emptyset$, and each $P_{1,j} = P_{1,j-1} \cup P_{2,j}$, it follows that there is a j such that $p_i \in P_{2,j}$. However, if $P_{2,j}$ contains p_i, and p_i lists the CompartmentEntity under discussion here as a creatable CompartmentEntity, as assumed, then the CompartmentEntity would be added to $S_{5,j}$ and then $S_{4,j}$. As no CompartmentEntities are removed from S_4 between iterations, the CompartmentEntity would be in $S_{4,n}$. But we are considering a CompartmentEntity not in $S_{4,n}$, so by contradiction, $p_i \notin P_{1,n}$. Therefore $p_i \in P_{3,n}$.

The definition of process activation requires that Processes be activated if the CompartmentEntities required for the process to have any effect are present, and that the requisite CompartmentEntities for activation must have had a non-zero amount at some point at least an infinitesimally small amount of time before activation. As we have made the supposition that $\Phi_i(t) \neq 0$ for some $0 \leq t \leq \tau$, it follows that p_i has amount non-zero at some time $t \in \mathbb{R}, 0 \leq t \leq \tau$. Therefore, the activation criterion function would have evaluated to True, when given the list $S_{4,n}$, that, by the conditions of this lemma, contains all CompartmentEntities that have non-zero values in $t \in \mathbb{R}, 0 \leq t < \tau$. As a result, p_i would have been added to the active Process list $P_{1,n}$ and not been moved to $P_{3,n}$. We have therefore shown that under the assumption, $p_i \in P_{1,n}$ and $p_i \in P_{3,n}$. But we have also shown that $P_{1,n}$ and $P_{3,n}$ are mutually exclusive. This is a contradiction to the assumption that the lemma does not hold. □

Lemma 10.3.6. *All CompartmentEntities that have non-zero amount at time zero are in the final set of active CompartmentEntities $S_{4,n}$.*

Proof. Suppose, for contradiction, that there was a CompartmentEntity that had non-zero amount at time zero that was not in $S_{4,n}$.

Then, by definition, the initial amount of that CompartmentEntity would not simplify to constant zero. Therefore, the CompartmentEntity would be added to the initial set of active CompartmentEntities, $S_{4,0}$. As has been shown already, $S_{4,i-1} \subseteq S_{4,i}$, and so by induction, the CompartmentEntity would be

present in $S_{4,n}$. This is a contradiction to the assumption that the lemma does not hold. □

Proof of theorem. By lemma 10.3.6, all CompartmentEntities that have non-zero amount at time zero are in the final set of active CompartmentEntities $S_{4,n}$. By induction, applying lemma 10.3.5, this result also holds for all later times. □

A Domain Specific Language for making ModML Reactions models

As with B.BasicDAEModel and U.UnitsDAEModel described previously, building a ReactionModel data-structure directly would be error-prone and repetitive, and reaction networks described this way would not be easily composable. The solution discussed here, in a similar vein to the introduction of B.ModelBuilder and U.ModelBuilder, is to introduce a monad transformer for building ReactionModels, ModelBuilderT, with a specialisation, ModelBuilder, where the underlying monad is the Identity monad.

Monad Transformer stack

Because building a ReactionModel requires building U.RealExpressions, and building U.RealExpressions is far more convenient when done in a U.ModelBuilder, ModelBuilderT is built as a monad transformer stack containing a Control.**Monad**.State.StateT ReactionModel monad transformer at the top, with a U.ModelBuilderT underneath that. The monad underneath the ModelBuilderT is underneath that, so that the underlying monad for ModelBuilder is U.ModelBuilder (i.e. U.ModelBuilderT Identity).

Operations on ModelBuilder can therefore either modify the ReactionModel, or be a U.ModelBuilder lifted to the top of the monad stack to describe a modification of the underlying UnitsDAEModel. To avoid the need for an explicit call to lift, type-classes ReactionModelBuilderAccess and

U.UnitsModelBuilderAccess are used, and both U.ModelBuilder and ModelBuilder implement U.UnitsModelBuilderAccess. This typeclass provides a function liftUnits, and most of the fundamental actions for U.ModelBuilder are actually described in terms of this typeclass, meaning that most functions that produce a U.ModelBuilder will work in ModML Reactions ModelBuilder (and the remainder can be converted explicitly using liftUnits).

Allocation functions

ModML Reactions provides functions for allocating newEntities, called newEntity, a function from a U.ModelBuilder Units to ModelBuilder Entity, and a monadic value for allocating new Compartments, newCompartment. In addition, for each of these functions, it includes variants that also set annotations to name the Compartment or Entity.

In addition, as with variables in ModML Units, it is useful to be able to define Compartments and Entities in one place and refer to them (including via Haskell's import mechanism) multiple times, and as before, for this to be useful, it is important that each time the monadic values are evaluated, they refer to the same Compartment or Entity, rather than allocating a new one (as with newEntity and newCompartment). As with variables, type-tagging is used to provide find-or-create functionality. The functions newNamedTaggedCompartment and newNamedTaggedEntity find or allocate a Compartment or Entity, respectively, based on a type-tag. Again as with variables in ModML Units, Template Haskell is used to provide syntactic sugar; declareNamedTaggedEntity and declareNamedTaggedCompartment can be used to declare such values. For example, the following is valid part of a ModML Reactions model (where R is ModML.Reactions.Reactions, and uConcentrationR is a modeller defined Haskell variable, describing the units of concentration):

```
R.declareNamedTaggedEntity [e|uConcentrationR|] "Water" "water"
R.declareNamedTaggedCompartment "Reactor Vessel" "vessel"
waterInVessel = water `R.inCompartment` vessel
```

Note that the type of waterInVessel is ModelBuilder CompartmentEntity.

New containment relationships can be added using addContainmentM, a function of type:

```
Compartment -> Compartment -> ModelBuilder ()
```

However, it is usually more convenient to use addContainment (also available as addContainmentX), of type:

```
ModelBuilder Compartment -> ModelBuilder Compartment ->
    ModelBuilder ()
```

In addition, EntityInstances can be added using addEntityInstance, a function of type:

```
ModelBuilder CompartmentEntity -> ModelBuilder
    EntityInstance -> ModelBuilder ()
```

ProcessBuilder

The Process data structure is a reasonably complex one, and so to simplify describing Processes, a monad is provided specifically for building Process data structures. ProcessBuilderT is a transformer that builds on top of ModelBuilderT, adding a Control.Monad.StateT Process model transformer to the top of the stack. As always, a non-transformer version, ProcessBuilder, is defined by placing the Identity monad underneath ProcessBuilderT.

Instances of ReactionModelBuilderAccess (and indirectly, UnitsModelBuilderAccess) are provided for ProcessBuilder, so monadic operations for units and reaction models can mostly be performed directly inside the ProcessBuilder, and liftReactions and liftUnits is available for the remaining cases.

Rather than require CompartmentEntities be explicitly added into the creatableCompartmentEntities and modifiableCompartmentEntities sets, and the activationCriterion function to be explicitly written, three data structures (that are used for operations in the ProcessBuilder monad, but are not reachable from the final ReactionModel structure) are defined. IsEssentialForProcess has two

constructors, EssentialForProcess and NotEssentialForProcess. CanBeCreatedByProcess has two constructors, CanBeCreatedByProcess and CantBeCreatedByProcess. CanBeModifiedByProcess has two constructors, ModifiedByProcess and NotModifiedByProcess. The function addEntity has the following data type:

```
addEntity :: IsEssentialForProcess ->
    CanBeCreatedByProcess -> CanBeModifiedByProcess ->
    Double -> ModelBuilder CompartmentEntity ->
    ProcessBuilder (U.ModelBuilder U.RealExpression)
```

This function performs almost all the work required to reference a CompartmentEntity from a Process. Firstly, it extracts the Units from the Entity, and allocates a placeholder variable with those Units.

Where the first argument is EssentialForProcess, the activationCriterion for the process is modified as follows: let the previous activationCriterion be $f_i(S)$, the new activationCriterion be $f_{i+1}(S)$, and c be the CompartmentEntity. Then $f_{i+1}(S) = (c \in S) \wedge f_i(S)$; in other words, in addition to all constraints so far, the essential CompartmentEntity is required for activation. If the first argument is NotEssentialForProcess, no change is made to the activationCriterion.

When the second argument is CanBeCreatedByProcess, the CompartmentEntity is added to the creatableCompartmentEntities; otherwise no change is made to the creatableCompartment entities. Likewise, when the third argument is ModifiedByProcess, the CompartmentEntity is added to modifiedCompartmentEntities, and otherwise no change is made.

The final step taken by addEntity is to create an entry in entityVariables mapping between the CompartmentEntity and the newly allocated variable, and an entry in stoichiometry between the CompartmentEntity and the stoichiometry value provided as the fourth argument. A U.RealExpression referencing the newly allocated U.RealVariable is returned.

The other fundamental operation available in the ProcessBuilder is the rateEquation function. This function takes a U.ModelBuilder RealExpression, and

produces a ProcessBuilder () that sets the rateTemplate to the RealExpression inside the monadic type of the first argument.

Adding Processes to models

There are three ways to reference Processes in ModML Reactions; as explicitCompartmentProcesses, as containedCompartmentProcesses, and as allCompartmentProcesses. The simplest case is explicitCompartmentProcesses. The function newExplicitProcess has type:

```
newExplicitProcess :: ProcessBuilder a -> ModelBuilder
    Process
```

This function executes the state monad for the ProcessBuilder, starting from an initial process. This initial Process is allocated a new processId. The initial activationCriterion is const True, meaning a function that evaluates to True for any argument. The creatableCompartmentEntities and modifiedCompartmentEntities start off as empty sets, and entityVariables and stoichiometry as empty maps. The rate is initially a constant U.RealExpression for dimensionless zero, although there is little point in defining a Process and not overriding this. After the state monad has been executed, the resulting Process is added to the list of explicitCompartmentProcesses.

It is more complicated to define a function for adding all Compartment Processes using the ProcessBuilder, because it is not possible, in general, to execute a function to monadic type to remove the monad. In other words, there is no general function like this:

```
-- The following doesn't exist.
execStateFunctionT :: Monad m => (b -> StateT s m a) ->
    s -> b -> m a
-- Even though the following does exist:
execStateT :: Monad m => StateT s m a -> s -> m a
```

There are at least two possible solutions to this problem; the first option would be to make a different monad transformer stack for all Compartment processes,

with additional state information allowing the Compartment to be fetched. However, this would add a lot of complexity because there would be three different monad stacks (for the explicit Compartment Processes, for the all Compartment Processes, and for the contained Compartment Processes). The visibility of this complexity to the modeller could be reduced by providing a type class allowing access to the state shared by all three.

The other solution, which is the one adopted by ModML Reactions, is to create a placeholder CompartmentEntity c_1, and then execute the state monad as for explict CompartmentProcesses. Finally, this Process is changed to a function that takes a CompartmentEntity argument c_2, and substitutes all occurrences of c_1 with c_2. This function from a CompartmentEntity to a Process is then added to the list allCompartmentProcesses.

This same strategy is used to implement newContainedCompartmentProcess, except that two place-holder variables are used to translate an argument of type:

```
Compartment -> Compartment -> ProcessBuilder a
```

into a function from two Compartments to a Process; the resulting function is added to the list of containedCompartmentProcesses.

In addition, it is expected that it will often be convenient to provide a long list of ProcessBuilders to be added. For this reason, additional functions in the plural form (like newAllCompartmentProcesses) are provided to add a list of Processes (or functions to Processes) at the same time.

Referencing CompartmentEntities

When defining CompartmentEntities, it is usually the case that Entities will be available in monadic form. Compartments that are defined as tagged types will usually be available in monadic form, but Compartments that are passed as an argument to a function are not in monadic form. To facilitate the use of these

two cases, two different functions are available [2]:

```
inCompartment :: ModelBuilder Entity -> ModelBuilder
    Compartment -> ModelBuilder CompartmentEntity
withCompartment :: ModelBuilder Entity -> Compartment ->
     ModelBuilder CompartmentEntity
```

Modellers can simply remember to use inCompartment with type-tagged compartments, and withCompartment with Compartments passed to a function used with newAllCompartmentProcesses or newContainedCompartmentProcess.

10.4 Example 1: A simple model built with ModML Reactions

To demonstrate the utility of ModML Reactions for building reaction networks, a basic network of two chemical reactions was chosen. The model includes two processes:

$$H_2 + O_2 \rightarrow H_2O + O\cdot \tag{10.3}$$

$$2O\cdot \rightarrow O_2 \tag{10.4}$$

This model was defined using the ModML Units and ModML Reactions mechanisms defined in this chapter. The full model is given in Listing 10.4, split across multiple pages.

Line 1 defines the Haskell language extensions required for the model; the set given on the line should be sufficient for most models. Lines 2-5 use the mechanism described in subsection 9.4 to load the required modules, ModML Units, ModML Reactions, ModML-PhysicalConstants, and typehash (needed for type-tagged values).

The model itself then begins on line 7 with a module definition. Lines 9 through 16 define what is imported, and under what qualified names. The recommended

[2] the functions are generalised to work with any monad; the more specialised type signature is shown here for simplicity.

Listing 10.4: Defining a simple chemical reaction network

```
{-# LANGUAGE NoMonomorphismRestriction,
    DeriveDataTypeable, TemplateHaskell #-}
-- +Require ModML-Units
-- +Require ModML-Reactions
-- +Require ModML-PhysicalConstants
-- +Require typehash

module ReactionModel
where
import qualified ModML.Units.UnitsDAEModel as U
import qualified ModML.Core.BasicDAEModel as B
import ModML.Units.UnitsDAEOpAliases
import qualified ModML.Reactions.Reactions as R
import qualified Data.Data as D
import qualified Data.TypeHash as D
import ModML.Units.SIUnits
import ModML.PhysicalConstants.Common (gasConstant)

uConcentration = uMole $*$ uLitre $**$ (-1)
uFlux = uConcentration $*$ uSecond $**$ (-1)
uNthOrderPerConcentration n = uConcentration $**$ (-n) $
    *$ uSecond $**$ (-1)

uConcentrationR = U.liftUnits uConcentration
R.declareNamedTaggedEntity [e|uConcentrationR|] "H2" "
    hydrogen2"
R.declareNamedTaggedEntity [e|uConcentrationR|] "O2" "
    oxygen2"
R.declareNamedTaggedEntity [e|uConcentrationR|] "O." "
    oxygenRadical"
R.declareNamedTaggedEntity [e|uConcentrationR|] "Water"
    "water"
R.declareNamedTaggedCompartment "Reactor␣Vessel" "vessel
    "


U.declareRealVariable [e|uKelvin|] "temperature" "
    temperature"
temperatureM = temperature >>= return . U.RealVariableE
...
```

convention, shown here, is to import ModML.Core.BasicDAEModel under the qualified name B, ModML.Units.UnitsDAEModel as U, and ModML.Reactions.Reactions as R.

Line 18 and 19 define straightforward units in terms of the SI Units imported from ModML.Units.SIUnits. Line 20 defines a function for generating units; the function parameter in this case is the order of the reaction.

Line 22 shows the utility of U.liftUnits to lift a U.ModelBuilder expression into an R.ModelBuilder expression; this is later used in lines 23-26 to define entities with these units called hydrogen2, oxygen2, oxygenRadical, and water. Line 27 then defines a single type-tagged compartment, vessel.

Line 30 shows how ModML Units can be combined with ModML Reactions by defining a ModML Units RealVariable, temperature. Line 31 defines temperatureM to be the result of wrapping the variable to make an expression, so it is more convenient to refer to the temperature later in the model.

Line 34 defines a function from a Compartment (called compartment here) to a ProcessBuilder; it is used later to add a Process. Line 35 is the Haskell syntax for the **do** notation, syntactic sugar for making monadic composition easier. Line 36 and 37 add H_2 to the Process; it is EssentialForProcess because the Process won't change anything if there is no H_2; CantBeCreatedByProcess means that it is not possible for $[H_2]$ to become non-zero, solely due to the Process here, if it is zero. ModifiedByProcess means that the Process can change the concentration of $[H_2]$. The -1 is the stoichiometry of the Process. Finally, hydrogen2 'R.withCompartment' compartment specifies the CompartmentEntity that is being referred to; R.withCompartment combines the type-tagged Entity hydrogen2 with the argument to the function compartment, to build a CompartmentEntity. The placeholder variable for the Entity is stored in h2var.

Similarly, more CompartmentEntities are referenced in the Process in lines 38-43.

In line 44, the rate equation for the Process is defined using a ModML Units expression. The kinetics data used in this model were taken from the NIST

```
33 ...
34 h2_o2_React compartment =
35     do
36       h2var <- R.addEntity R.EssentialForProcess R.
             CantBeCreatedByProcess R.ModifiedByProcess (-1)
37                 (hydrogen2 `R.withCompartment`
                       compartment)
38       o2var <- R.addEntity R.EssentialForProcess R.
             CantBeCreatedByProcess R.ModifiedByProcess (-1)
39                 (oxygen2 `R.withCompartment`
                       compartment)
40       watervar <- R.addEntity R.NotEssentialForProcess R
             .CanBeCreatedByProcess R.ModifiedByProcess 1
41                       (water `R.withCompartment`
                             compartment)
42       oradvar <- R.addEntity R.NotEssentialForProcess R.
             CanBeCreatedByProcess R.ModifiedByProcess 1
43                       (oxygenRadical `R.withCompartment`
                             compartment)
44       R.rateEquation $ h2var .*. o2var .*.
45                         (U.realConstant (
                               uNthOrderPerConcentration 2)
                               1.37E9) .*.
46                         (U.expX $ U.realConstant (uJoule
                               $*$ uMole $**$ (-1)) (-295496)
                                ./.
47                          (gasConstant .*. temperatureM))
48
49 o_o_radical_Combine compartment =
50     do
51       oradvar <- R.addEntity R.EssentialForProcess R.
             CantBeCreatedByProcess R.ModifiedByProcess (-2)
              (oxygenRadical `R.withCompartment` compartment
             )
52       o2var <- R.addEntity R.NotEssentialForProcess R.
             CanBeCreatedByProcess R.ModifiedByProcess 1 (
             oxygen2 `R.withCompartment` compartment)
53       R.rateEquation $ oradvar.**.U.realConstant U.
             dimensionless 3 .*.
54                          (U.realConstant (
                               uNthOrderPerConcentration 3)
                               1.89E7) .*.
55                          (U.expX $ U.realConstant (uJoule
                                $*$ uMole $**$ (-1)) 7483
                               ./.
56                          (gasConstant .*. temperatureM))
57 ...
```

```
58 ...
59 reactionModel = do
60   R.newAllCompartmentProcess h2_o2_React
61   R.newAllCompartmentProcess o_o_radical_Combine
62   -- Declare the initial presence of oxygen in vessel
63   R.addEntityInstance (oxygen2 `R.inCompartment` vessel)
       (R.entityFromProcesses (U.realConstant
       uConcentration 1) (U.realConstant uFlux 0))
64   -- And hydrogen...
65   R.addEntityInstance (hydrogen2 `R.inCompartment`
       vessel) (R.entityFromProcesses (U.realConstant
       uConcentration 1) (U.realConstant uFlux 0))
66
67 unitsModel :: Monad m => U.ModelBuilderT m ()
68 unitsModel = do
69   R.runReactionBuilderInUnitBuilder reactionModel
70   temperatureM `U.newEq` (U.realConstant uKelvin 1300)
71
72 model = B.buildModel $ do
73   U.unitsToCore uSecond unitsModel
```

Chemical Kinetics Database [Mallard et al., 1998], and kinetics for this particular reaction are due to Karkach and Osherov [1999].

Lines 49-56 follow a similar structure to lines 34-46, but define a function for the second Process. The kinetics for this reaction are based on Tsang and Hampson [1986].

The function reactionModel, starting at line 59, defines how the parts of the model are combined to build the final model. Lines 60-61 define all Compartment Processes from the two functions defined earlier.

Line 63 adds an EntityInstance for O_2 in the vessel, specifying that the CompartmentEntity amount is determined by the Processes, that the initial concentration is equal to 1 molL^{-1}, and that no additional flux needs to be added to the model. This line is needed only to specify the initial concentration; if it was absent, the initial concentration would be treated as 0. Line 65 is similar, except that it is for H_2 rather than O_2.

Line 68 describes the unitsModel. Line 69 uses R.runReactionBuilderInUnitBuilder on the reactionModel to convert it into a Unit model. Line 70 then adds to the

model build using ModML Reactions, by adding a ModML Units equation equating the temperature to a constant 1300K (this is an isothermal model).

Finally, line 72 builds the model by converting the U.ModelBuilder into a B.ModelBuilder, and running that ModelBuilder to obtain the BasicDAEModel.

Using the model for simulations

The model, as described, can now be used to perform simulations. I used the CSV-AutoSolver program described in subsection 9.4 with default solver parameters to solve the model between time zero and time 1000s. The comma-separated value output identifies variables based on the variable annotations; for CompartmentEntity amounts, these originate with annotations on Compartments and Entities; ModML Reactions uses these to name variables when it creates them for CompartmentEntity amounts; ModML Units copies the variable annotations when converting to ModML Core. The final result is that variables have names like "Amount of H2 in Reactor Vessel".

```
csv-autosolver ReactionModel.hs --endtime 1000 --
    csvoutfile out.csv
```

Using R, selected amounts from this CSV file were plotted, shown in Figure 10.4.

As another demonstration, it would be interesting to see how the concentration of O· varies with time. Using CSV-AutoSolver, the temperature constraint was removed, and sampling based sensitivity analysis was performed in 10K steps between 1000K and 2000K at time 100s. Note that this will show two competing effects; higher temperatures mean faster reactions, but they also mean that the reactants may be used up sooner, so for a high enough temperature, by 100s, the reaction may almost be at completion (and so have a low intermediate radical concentration).

```
csv-autosolver --removeconstraint=temperature --
    sensitivityvariable=temperature --
    sensitivitylowerbound=1000 --sensitivityupperbound
```

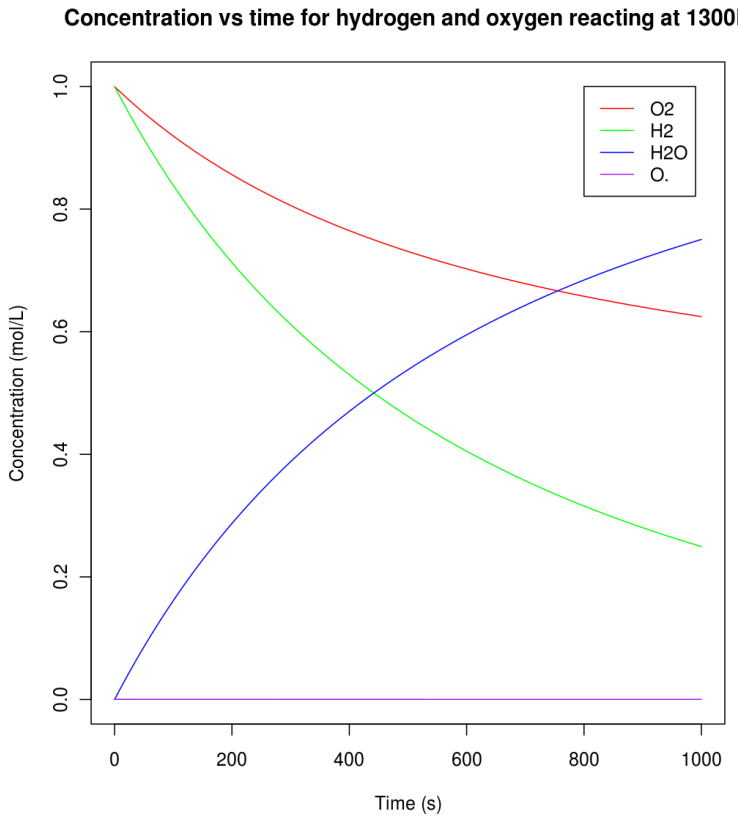

Figure 10.4: A simulation of hydrogen and oxygen reacting under isothermal conditions at 1300K

```
=2000 --sensitivitystep=10 --endtime=100 --csvoutfile
=temp-sens-100s.csv ReactionModel.hs
```

This data was plotted using R, shown in Figure 10.5.

10.5 Example 2: A biological model combining ModML Reactions and ModML Units

The model of the cardiac myofilament (at a single point) due to Rice et al. [2008] combines both the reactions involved in cross-bridge cycling, with a number of equations describing the mechanics of the muscle. As such, it is an

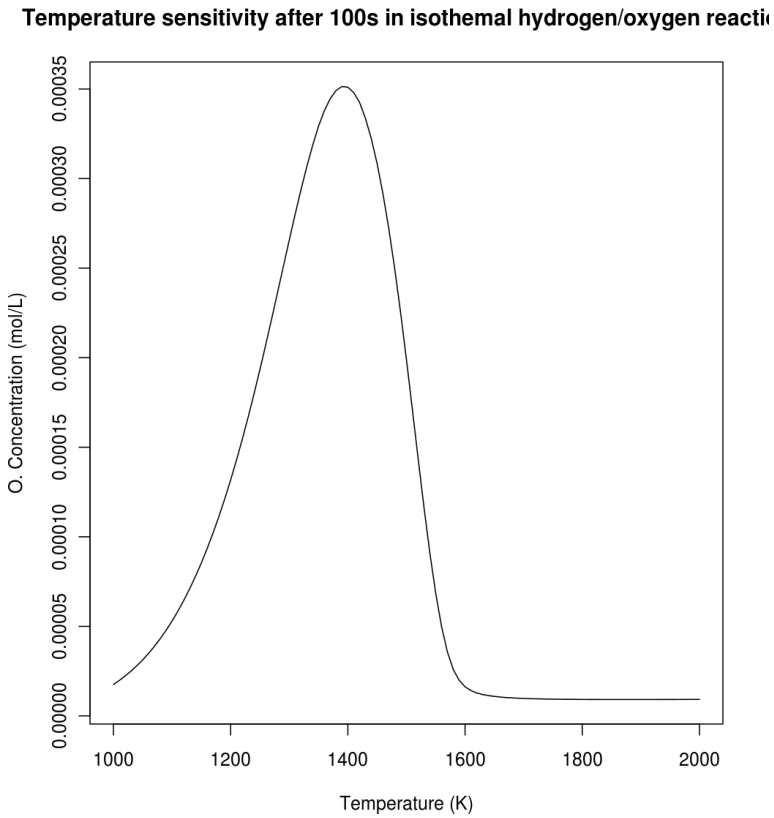

Figure 10.5: The isothermal temperature sensitivity of oxygen free radical concentrations 100s after mixing hydrogen and oxygen. The shape of the curve is because higher temperatures accelerate the reaction, but they also cause reactants to be depleted, slowing the reaction rate after 100s

ideal model to demonstrate the utility of composing ModML Reactions models with ModML Units models in a more extensive way than simply fixing the temperature as in the previous example.

To this end, I have built a ModML version of this model. This version of the model is given in Listing 10.5

Lines 1-4 will be familiar from the previous model. In addition, a few more Haskell libraries are imported in lines 5-6 to form part of the model.

Line 25 defines a new type for a ModML Units ModelBuilderT with a RealExpression monadic value.

Listing 10.5: The Rice 2008 model

```
1 {-# LANGUAGE NoMonomorphismRestriction,
      DeriveDataTypeable, TemplateHaskell #-}
2 -- +Require ModML-Units
3 -- +Require ModML-Reactions
4 -- +Require typehash
5 -- +Require containers
6 -- +Require mtl
7
8 module Rice2008
9 where
10 import qualified ModML.Units.UnitsDAEModel as U
11 import qualified ModML.Core.BasicDAEModel as B
12 import ModML.Units.UnitsDAEOpAliases
13 import qualified ModML.Reactions.Reactions as R
14 import qualified Data.Data as D
15 import qualified Data.TypeHash as D
16 import ModML.Units.SIUnits
17 import qualified Control.Monad as M
18 import qualified Control.Monad.Identity as I
19 import Data.Maybe
20 import Data.List
21 import qualified Data.Map as M
22 import Data.Map ((!))
23 import qualified Debug.Trace as DT
24
25 type RExB m = U.ModelBuilderT m U.RealExpression
```

Lines 26-74 define a data structure called Parameters, with a single constructor, Parameters. Defining a structure for parameters is not strictly necessary, but it helps to document what the parameters to a model are, and simplifies both passing parameters around and manipulating sets of parameters.

The first two named arguments of Parameters, maxSarcomereLength and minSarcomereLength (lines 28 and 29), carry the RealExpression builders for the maximum and minimum possible sarcomere lengths, respectively. Further sarcomere geometry parameters follow; hbareLength and thinFilamentLength (lines 30 and 31).

The temperature parameter (line 32) describes the constant temperature for the model.

Many of the Parameters describe the kinetics of different Reactions; because

```
26 data Parameters m = Parameters {
27       maxSarcomereLength :: RExB m,
                                        -- SL_{max}
28       minSarcomereLength :: RExB m,
                                        -- SL_{min}
29       thickFilamentLength :: RExB m,
                                      -- length_{thick}
30       hbareLength :: RExB m,
                                              -- length_{
             hbare}
31       thinFilamentLength :: RExB m,
                                        -- length_{thin}
32       temperature :: RExB m,
                                              -- TmpC
33       calciumOnTrop :: ReactionParameters m,
                                -- k_{on} / Qk_{on}
34       calciumOffTropL :: ReactionParameters m,
                               -- k_{offL} ...
35       calciumOffTropH :: ReactionParameters m,
                               -- k_{offH} ...
36       tropomyosinNToP :: ReactionParameters m,
                               -- k_{n_p} ...
37       tropomyosinPToN :: ReactionParameters m,
                               -- k_{p_n} ...
38       crossBridgeFormation :: ReactionParameters m,
                           -- f_{app} ...
39       crossBridgeDissociation :: ReactionParameters m,
                     -- g_{app} ...
40       crossBridgeRotation :: ReactionParameters m,
                          -- h_f...
41       crossBridgeReverseRotation :: ReactionParameters m
             ,      -- h_b...
42       rotatedCrossBridgeDissociation ::
             ReactionParameters m, -- g_{xb}...
43       permissiveHalfActivationConstant :: RExB m,
                           -- perm_{50}
44       permissiveHillCoefficient :: RExB m,
                                  -- n_{perm}
45       overlapModStrongToWeak :: RExB m,
                                    -- gslmod
46       preRotStrainFactor :: RExB m,
                                        -- hfmdc
47       strainEffectPositive :: RExB m,
                                      -- \sigma_p
48       strainEffectNegative :: RExB m,
                                      -- \sigma_n
49       meanStrain :: RExB m,
                                              -- x_0
50       strainScalingFactor :: RExB m,
                                       -- \phi
51       restingSarcomereLength :: RExB m,
                                    -- SL_{rest}
52       passiveTitinConstant :: RExB m,
                                      -- PCon_{titin}
```

Rates are described in a consistent form in the paper, it makes sense to create a data structure for the parameters to this consistent form of rate. This is discussed in more detail below; this structure, ReactionParameters, is used on lines 33-42 in arguments to the Parameters constructor.

Lines 43-44 define Parameter arguments for two parameters that control how much of the regulatory troponin is in the permissive state. Lines 45-48 control the relationship between the mechanics and the reaction kinetics. Lines 49-51 describe mechanical parameters for computing sarcomere distortion.

Lines 52-54 describe parameters controlling the relationship between the sarcomere length and the force due to titin, while lines 55-56 achieve the same thing for collagen. The argument on line 57 describes the mass of the system in the normalised units of the model, while normalisedViscosity (line 58) describes the viscosity in normalised units.

Line 59 introduces constantAfterload, an external force that is applied to the myofilament (depending on how the model is used). Line 60 describes the stiffness of a spring that can be connected to the model to perform certain experiments.

Lines 61-70 all define parameters for initial values for a range of state variables in the model.

Line 71 introduces the named argument calciumTransient. This has type TransientParameters, described later in the model, providing a way to specify a transient function in a standard form.

Lines 72-73 introduce the arguments modelContext of type ModelContext and contractionType of type ContractionType. The possible values for these parameters are discussed below.

The ReactionParameters data type is defined on lines 75-80, and describes the kinetics of a reaction in the form used throughout the model, in terms of a base rate, a species modified, a modifying multiplier (otherMod), and a parameter q10, describing the multiplicative sensitivity to a 10K temperature shift.

```
53        passiveTitinExponent :: RExB m,
                                      -- PExp_{titin}
54        sarcomereLengthCollagen :: RExB m,
                                     -- SL_{collagen}
55        passiveCollagenConstant :: RExB m,
                                     -- PCon_{collagen}
56        passiveCollagenExponent :: RExB m,
                                     -- PExp_{collagen}
57        normalisedMass :: RExB m,
                                          -- Mass
58        normalisedViscosity :: RExB m,
                                       -- Viscosity
59        constantAfterload :: RExB m,
                                        -- F^{constant}_{
              afterload}
60        stiffness :: RExB m,
                                                -- KSE
61        initialSarcomereLength :: RExB m,
62        initialtmNNoXB :: RExB m,
63        initialtmPNoXB :: RExB m,
64        initialtmPXB :: RExB m,
65        initialXBPreR :: RExB m,
66        initialXBPostR :: RExB m,
67        initialxXBPreR :: RExB m,
68        initialxXBPostR :: RExB m,
69        initialCaTropH :: RExB m,
70        initialCaTropL :: RExB m,
71        calciumTransient :: TransientParameters m,
72        modelContext :: ModelContext,
73        contractionType :: ContractionType
74      }
75 data ReactionParameters m = ReactionParameters {
76        baseRate :: RExB m,
77        otherMod :: Maybe (RExB m),
78        speciesMod :: Maybe (RExB m),
79        q10 :: RExB m
80      }
81 data TransientParameters m = TransientSpike {
82   transientStartTime :: RExB m,
83   transientBase :: RExB m,
84   transientAmplitude :: RExB m,
85   transientTime1 :: RExB m,
86   transientTime2 :: RExB m
87 } | TransientExpRamp { transientRampInitial :: RExB m,
      transientRampConstant :: RExB m }
88 data ModelContext = IsolatedCell | Trabeculae
89 data ContractionType = Isotonic |
90                        Isometric -- i.e. constant length
```

The TransientParameters data type is defined on lines 81-87. There are two constructors, one for a spike, and one for a ramp. TransientSpike has five arguments, describing the start time, base level, spike amplitude, and two time parameters. TransientExpRamp has two parameters, an initial value and an exponential constant.

ModelContext is defined on line 88. It has two constructors, with no arguments. The first one, IsolatedCell, indicates the model is an isolated cell, while the other, Trabeculae, indicates it is in the context of tissue.

ContractionType is defined on line 89, and again has two constructors, Isotonic, for constant force, and Isometric, for constant total length.

Line 93 declares a new base type, stored in variable normalisedForceBase. While types are declared for SI base units, this model uses normalised units. The units could be treated as dimensionless, but better error checking of the model is achieved by defining new base units. A builder for the Units, uNormalisedForce, is built from the BaseUnits builder on line 103.

Lines 94-102 define units used throughout the model, including versions lifted from U.ModelBuilder monad to R.ModelBuilder monad.

In line 104-107, units for mass, viscosity, stiffness, and the integral of normalised force are defined in terms of uNormalisedForce.

Lines 109-117 use the Template Haskell function R.declareNamedTaggedEntity to define all the Entities found in the model. Line 119 then defines the only Compartment in the model, cardiacMuscleSite.

Lines 120-129 define U.RealVariables that are used in the model, outside of the ModML Reactions part of the model.

Line 131 declares that the model is simply the parameterised model, with the default parameters applied.

Line 133 then defines the parameterised model; it is simply the units model, with the parameters given, converted into ModML Core.

```
92 -- Model units
93 U.declareBaseType "normalisedForce" "normalisedForceBase
       "
94 uProbability = U.dimensionless
95 uProbabilityR = U.liftUnits uProbability
96 uDistance = uMicro $*$ uMetre
97 uDistanceR = U.liftUnits uDistance
98 uConcentration = uMilli $*$ uMole $*$ uLitre $**$ (-1)
99 uConcentrationR = U.liftUnits uConcentration
100 uFlux = uConcentration $*$ uSecond $**$ (-1)
101 uFluxR = U.liftUnits uFlux
102 uNthOrderRate n = uConcentration $**$ (-n) $*$ uSecond $
       **$ (-1)
103 uNormalisedForce = M.liftM U.singletonUnit
       normalisedForceBase
104 uNormMass = uNormalisedForce $*$ uSecond$**$2 $*$
       uDistance$**$(-1)
105 uNormViscosity = uNormMass $*$ uSecond$**$(-1)
106 uNormStiffness = uNormalisedForce $*$ (uDistance $**$
       (-1))
107 uForceIntegral = uNormalisedForce $*$ uSecond
108
109 R.declareNamedTaggedEntity [e|uProbabilityR|] "Non-
       permissive tropomyosin not near cross-bridge" "
       tmNNoXB"
110 R.declareNamedTaggedEntity [e|uProbabilityR|] "
       Permissive tropomyosin not near cross-bridge" "
       tmPNoXB"
111 R.declareNamedTaggedEntity [e|uProbabilityR|] "Non-
       permissive tropomyosin near cross-bridge" "tmNXB"
112 R.declareNamedTaggedEntity [e|uProbabilityR|] "
       Permissive tropomyosin near cross-bridge" "tmPXB"
113 R.declareNamedTaggedEntity [e|uProbabilityR|] "Cross
       bridges (pre rotation)" "xbPreR"
114 R.declareNamedTaggedEntity [e|uProbabilityR|] "Cross
       bridges (post rotation)" "xbPostR"
115 R.declareNamedTaggedEntity [e|uProbabilityR|] "Troponin
       with calcium bound to the high-affinity regulatory
       site" "caTropH"
116 R.declareNamedTaggedEntity [e|uProbabilityR|] "Troponin
       with calcium bound to the low-affinity regulatory
       site" "caTropL"
117 R.declareNamedTaggedEntity [e|uConcentrationR|] "Calcium
       ^(2+) concentration" "calcium"
118
119 R.declareNamedTaggedCompartment "Cardiac Muscle Site" "
       cardiacMuscleSite"
```

```
120 U.declareRealVariable [e|uDistance|] "Sarcomere length"
      "sarcomereLength" -- SL
121 U.declareRealVariable [e|uDistance|] "Mean distortion
      pre-rotation" "meanDistortionPreR" -- xXBPreR
122 U.declareRealVariable [e|uDistance|] "Mean distortion
      post-rotation" "meanDistortionPostR" -- xXBPostR
123 U.declareRealVariable [e|U.dimensionless|] "Fraction of
      strongly bound crossbridges" "fractSBXB" -- Fract_{
      SBXB}
124 U.declareRealVariable [e|uForceIntegral|] "Integral of
      Force" "integralForce" -- Integral_{Force}
125 U.declareRealVariable [e|uNormStiffness|] "Stiffness" "
      stiffnessV"
126 U.declareRealVariable [e|uProbability|] "Cross-bridge
      duty fraction pre-rotation" "xbDutyFracPreR"
127 U.declareRealVariable [e|uProbability|] "Cross-bridge
      duty fraction post-rotation" "xbDutyFracPostR"
128 U.declareRealVariable [e|uProbability|] "Cross-bridge
      maximum pre-rotation" "xbMaxPreRotation"
129 U.declareRealVariable [e|uProbability|] "Cross-bridge
      maximum post-rotation" "xbMaxPostRotation"
130
131 model = parameterisedModel defaultParameters
132
133 parameterisedModel p = B.buildModel $ do
134   U.unitsToCore uSecond (unitsModel p)
135
136 unitsModel :: Monad m => Parameters m -> U.ModelBuilderT
      m ()
137 unitsModel p = do
138   fromPre <- unitsModelBeforeReaction p
139   (cem, _, ces) <-
140     R.runReactionBuilderInUnitBuilder' (
          reactionModelWithCalciumTransient fromPre p)
141   unitsModelAfterReaction p fromPre cem ces
142
143 unitsModelBeforeReaction :: Monad m => Parameters m -> U
      .ModelBuilderT m (RExB m)
144 unitsModelBeforeReaction p = do
145   (U.realVariable stiffnessV) `U.newEq` (stiffness p)
146   U.mkNamedRealVariable uProbability
147     "Amount of Permissive tropomyosin near cross bridge
          in Cardiac Muscle Site"
```

Line 136 then defines the units model. The units model is split into three parts; the units model before reactions (which defines things that are needed for the reaction model), the reactions model, translated into ModML Units, and the remainder of the units model (which can refer to variables allocated when processing the reactions model).

The function unitsModelBeforeReaction is defined on line 143. It equates stiffnessV to the computed RealExpression for stiffness; this is useful because it causes the solver to show the computed value of the stiffness, but could otherwise be avoided. Next, a variable is allocated to describe the amount of permissive tropomyosin near cross-bridges. This variable is allocated before the reaction model is processed, because it is controlled from outside of the reaction model but it affects the reaction model.

The function unitsModelAfterReaction is defined on line 152. It takes arguments containing the variable allocated by unitsModelBeforeReaction, as well the CompartmentEntities returned by the reaction builder and the map from CompartmentEntities to the corresponding variables for their amount. This information allows all the pieces of the model to be used. Lines 155-156 define the equation for the amount of permissive tropomyosin near cross bridges in terms of amount variables looked up in the map for CompartmentEntities. This equation ensures that normalised quantities that should add to one actually do; allowing ModML Reactions to control this amount directly could mean that the value would drift in simulations due to truncation error.

Lines 158 and 159 extract the variables for the amount of the two CompartmentEntities that affect the physical (mechanics) part of the model, namely the bound cross-bridges pre- and post- rotation in the model. These are then passed to the physical model on line 161.

The function physicalModel is itself defined on line 166, in terms of the parameters and the pre- and post- rotation cross-bridge amounts. This definition is split into the mean distortion model and the sarcomere length model.

The mean distortion model is defined starting on line 160. The standardRate function (discussed in more detail later), is used to obtain CommonSubexpres-

```
148 unitsModelAfterReaction ::
149   Monad m => Parameters m -> RExB m -> M.Map R.
          CompartmentEntity U.RealExpression ->
150       (R.CompartmentEntity, R.CompartmentEntity, R.
             CompartmentEntity, R.CompartmentEntity, R.
             CompartmentEntity) ->
151        U.ModelBuilderT m ()
152 unitsModelAfterReaction p permTropNorm cem (tmPXBMuscle,
       xbPreRMuscle, xbPostRMuscle, preRMuscle, postRMuscle
      ) =
153   do
154     -- Normalisation equation...
155     permTropNorm `U.newEq` (U.realConstant uProbability
          1 .-. (return $ cem!tmPXBMuscle) .-.
156                                 (return $ cem!xbPreRMuscle)
                                      .-. (return $ cem!
                                      xbPostRMuscle))
157
158     let preRVar = return $ cem!preRMuscle
159     let postRVar = return $ cem!postRMuscle
160
161     physicalModel p preRVar postRVar
162     return ()
163
164 -- The physical model...
165 physicalModel :: Monad m => Parameters m -> RExB m ->
       RExB m -> U.ModelBuilderT m ()
166 physicalModel p preRVar postRVar = do
167   meanDistortionModel p
168   sarcomereLengthModel p preRVar postRVar
169
170 meanDistortionModel p = do
171   fappT <- U.realCommonSubexpression $ standardRate (
          crossBridgeFormation p) (temperature p)
172   gappT <- U.realCommonSubexpression $
          xbPreRToPermissiveRate p
173   hbT <- U.realCommonSubexpression $ standardRate (
          crossBridgeReverseRotation p) (temperature p)
174   gxbT <- U.realCommonSubexpression $
          xbPostRToPermissiveRate p
175   hfT <- U.realCommonSubexpression $
          crossBridgeRotationRate p
176   preRTerm <- U.realCommonSubexpression $ fappT .*. (hbT
         .+. gxbT)
177   postRTerm <- U.realCommonSubexpression $ fappT .*. hfT
178   tot <- U.realCommonSubexpression $ preRTerm .+.
          postRTerm .+. gxbT .*. (hfT .+. gappT) .+. gappT
          .*. hbT
179   (U.realVariable xbDutyFracPreR) `U.newEq` (preRTerm
          ./. tot)
180   (U.realVariable xbDutyFracPostR) `U.newEq` (postRTerm
          ./. tot)
```

sions for the rates of different processes from the ReactionParameters structures, on lines 171-175. These are used to compute the pre- and post- rotation cross-bridge duty fractions on lines 179 and 180. The numerators and denominators of these fractions are split out into RealCommonSubexpressions.

Line 181 adds a boundary equation. The first part of the boundary equation is the conndition; in this case boundVariable (i.e. time) is zero. The second and third parts describe the equality that holds at this time; in this case, the effect is that at time zero, the mean distortion pre-rotation is equated to the initial value from the parameters.

Next, the derivative of the mean distortion, pre-rotation is described in an equation starting on line 183.

Similarly, line 185 describes the initial value for mean-distortion post-rotation, and line 187 describes the derivative.

The next part of the model is the sarcomere length model, the function for which is defined starting on line 190. RealCommonSubexpressions are allocated based on the defined base rates in the ReactionParameters from the Parameters structure, and these are further combined into other RealCommonSubexpressions on lines 196-198. These expressions are then used, on lines 199-201, to compute variables describing the maximal cross-bridge rotation for pre- and post- rotated cross-bridges. Variables, rather than CommonSubexpressions, are used here so the values can be inspected in the simulation output.

Line 202 provides an initial condition for the sarcomere length, while line 204 provides the equation for the derivative.

Because ModML Units doesn't directly support integral equations, the variable integralForce is used to represent the definite integral of the force from 0 until the current time. Line 208 adds an initial condition that integralForce is zero at time 0, and line 223 specifies the derivative of integralForce to be the total force.

The contributing forces of the total force are defined between lines 210 and 222. Lines 210-213 allocate variables for these contributing forces; as with some of

```
181   U.newBoundaryEq {- when -} (U.realConstant U.
          boundUnits 0 .==. U.boundVariable)
182                                (U.realVariable
                                       meanDistortionPreR) {-
                                       == -} (initialxXBPreR p)
183   (U.derivative (U.realVariable meanDistortionPreR)) `U.
          newEq`
184     (U.dConstant 0.5 .*. U.derivative (U.realVariable
            sarcomereLength) .+. (strainScalingFactor p ./. (
            U.realVariable xbDutyFracPreR)) .*. (fappT .*. (U
            .negateX (U.realVariable meanDistortionPreR)) .+.
             hbT .*. ((U.realVariable meanDistortionPostR)
            .-. meanStrain p .-. (U.realVariable
            meanDistortionPreR))))
185   U.newBoundaryEq {- when -} (U.realConstant U.
          boundUnits 0 .==. U.boundVariable)
186                                (U.realVariable
                                       meanDistortionPostR) {-
                                       == -} (initialxXBPostR p
                                       )
187   (U.derivative (U.realVariable meanDistortionPostR)) `U
          .newEq`
188     (U.dConstant 0.5 .*. U.derivative (U.realVariable
            sarcomereLength) .+. (strainScalingFactor p ./. (
            U.realVariable xbDutyFracPostR)) .*. (hfT .*. ((U
            .realVariable meanDistortionPreR) .+. meanStrain
            p .-. (U.realVariable meanDistortionPostR))))
189
190 sarcomereLengthModel p preRVar postRVar = do
191   fapp <- U.realCommonSubexpression $ baseRate (
          crossBridgeFormation p)
192   gapp <- U.realCommonSubexpression $ baseRate (
          crossBridgeDissociation p)
193   hb <- U.realCommonSubexpression $ baseRate (
          crossBridgeReverseRotation p)
194   gxb <- U.realCommonSubexpression $ baseRate (
          rotatedCrossBridgeDissociation p)
195   hf <- U.realCommonSubexpression $ baseRate (
          crossBridgeRotation p)
196   preRContrib <- U.realCommonSubexpression $ fapp .*. (
          hb .+. gxb)
197   postRContrib <- U.realCommonSubexpression $ fapp .*.
          hf
198   denom <- U.realCommonSubexpression $ preRContrib .+.
          postRContrib .+. gxb .*. hf .+. gapp .*. (hb .+.
          gxb)
199   (U.realVariable xbMaxPreRotation) `U.newEq` (
          preRContrib ./. denom)
200   (U.realVariable xbMaxPostRotation) `U.newEq` (
          postRContrib ./. denom)
201   (U.realVariable fractSBXB) `U.newEq` ((preRVar .+.
          postRVar) ./. ((U.realVariable xbMaxPreRotation)
          .+. (U.realVariable xbMaxPostRotation)))
```

```
202   U.newBoundaryEq {- when -} (U.realConstant U.
          boundUnits 0 .==. U.boundVariable)
203                                (U.realVariable
                                       sarcomereLength) {- ==
                                       -} (
                                       initialSarcomereLength p
                                       )
204   (U.derivative $ U.realVariable sarcomereLength) `U.
          newEq`
205     (((U.realVariable integralForce) .+.
206       (initialSarcomereLength p .-. (U.realVariable
              sarcomereLength)) .*. normalisedViscosity p)
207     ./. normalisedMass p)
208   U.newBoundaryEq {- when -} (U.realConstant U.
          boundUnits 0 .==. U.boundVariable)
209                                (U.realVariable
                                       integralForce) {- == -}
                                       (U.realConstant
                                       uForceIntegral 0)
210   vpassiveForce <- U.mkNamedRealVariable
          uNormalisedForce "Passive Force"
211   vpreloadForce <- U.mkNamedRealVariable
          uNormalisedForce "Preload Force"
212   vafterloadForce <- U.mkNamedRealVariable
          uNormalisedForce "Afterload Force"
213   vactiveForce <- U.mkNamedRealVariable uNormalisedForce
          "Active Force"
214   vpassiveForce `U.newEq` (passiveForce p (U.
          realVariable sarcomereLength))
215   vpreloadForce `U.newEq` (preloadForce p)
216   vafterloadForce `U.newEq` (afterloadForce p)
217   forceNormalisation <- U.mkNamedRealVariable (
          uProbability $*$ uDistance) "Force normalisation
          factor"
218   forceNormalisation `U.newEq` (meanStrain p .*. (U.
          realVariable xbMaxPostRotation))
219   vactiveForce `U.newEq` (singleOverlapThick p (U.
          realVariable sarcomereLength) .*.
220                              (U.realVariable
                                       meanDistortionPreR .*.
221                               preRVar .+. U.realVariable
                                       meanDistortionPostR .*.
                                       postRVar) ./.
222                             forceNormalisation)
223   (U.derivative $ U.realVariable integralForce) `U.newEq
          `
224     U.negateX (vactiveForce .+. vpassiveForce .-.
            vpreloadForce .-. vafterloadForce)
225
226 reactionModelWithCalciumTransient fromPre p = do
227   R.addEntityInstance
228         (calcium `R.inCompartment` cardiacMuscleSite)
229         (R.entityClamped (standardTransient (
              calciumTransient p) U.boundVariable))
230   reactionModel fromPre p
```

the other variables encountered, this is not strictly necessary, but forces these values to be available in the solver output. The equations for these variables, on lines 214-216, use functions for computing each force, defined later. A similar approach of defining a variable and adding an equation is used for the force normalisation factor on lines 217-218. The equation for the active force is then defined on lines 219-222.

Lines 226-230 define reactionModelWithCalciumTransient. This demonstrates the use of R.entityClamped to clamp the amount of a CompartmentEntity (in this case, calcium in the cardiac muscle site) to a specific value; in this case, the calcium transient specified in the parameters. This is composed with reactionModel. By splitting out the model with the calcium transient imposed from the main part of the reaction model, it becomes possible to use the main reaction model without imposing the calcium transient (for example, to build a larger model that includes a model of calcium dynamics).

Line 233 defines the function reactionModel, describing the main part of the reactions model. This is composed from three parts; a definition of the processes, a definition of the EntityInstances, and an extraction of the CompartmentEntities used externally to the ReactionModel (i.e. in the physical part of the model).

The reactionProcesses function on lines 238-248 uses R.newAllCompartmentProcesses to simply specify a list of processes that occur anywhere, as functions computing the ProcessBuilder; the model is defined entirely using all Compartment Processes.

Lines 250-279 define the EntityInstances for the model. EntityInstances don't need to be specified for CompartmentEntities that have an initial amount of zero, so only those with non-zero amounts are specified. The EntityInstances are straightforward specifications based on the initial values in Parameters.

The externalEntityIdentification function from lines 280-286 extracts the CompartmentEntities that are useful for the physical part of the model and puts them in a tuple; in the physical model, they are used as keys for map lookups to find the corresponding variables for the amounts of the CompartmentEntities.

```
232 -- The reaction model...
233 reactionModel normalisedPermissive p = do
234   reactionProcesses p
235   reactionEntityInstances p normalisedPermissive
236   externalEntityIdentification
237
238 reactionProcesses p = do
239   R.newAllCompartmentProcesses [
240     calciumBindingToTroponinSite p caTropH,
241     calciumDisassociatingTroponinSite p (calciumOffTropH
          p) caTropH,
242     calciumBindingToTroponinSite p caTropL,
243     calciumDisassociatingTroponinSite p (calciumOffTropL
          p) caTropL,
244     nToPNotNearXB p, pToNNotNearXB p,
245     nToPNearXB p, pToNNearXB p, pToXBPreR p,
246     xbPreRToPermissive p, xbPostRToPermissive p,
247     crossBridgePreToPost p, crossBridgePostToPre p
248                   ]
249
250 reactionEntityInstances p normalisedPermissive = do
251   -- Now the entity instances where we have an initial
        value...
252   let zeroProbFlux = U.realConstant (uNthOrderRate 0) 0
253   R.addEntityInstance
254         (tmNNoXB `R.inCompartment` cardiacMuscleSite)
255         (R.entityFromProcesses (initialtmNNoXB p)
              zeroProbFlux)
256   R.addEntityInstance
257         (tmPNoXB `R.inCompartment` cardiacMuscleSite)
258         (R.entityFromProcesses (initialtmPNoXB p)
              zeroProbFlux)
259   R.addEntityInstance
260         (tmPXB `R.inCompartment` cardiacMuscleSite)
261         (R.entityFromProcesses (initialtmPXB p)
              zeroProbFlux)
262   R.addEntityInstance
263         (xbPreR `R.inCompartment` cardiacMuscleSite)
264         (R.entityFromProcesses (initialXBPreR p)
              zeroProbFlux)
265   R.addEntityInstance
266         (xbPostR `R.inCompartment` cardiacMuscleSite)
267         (R.entityFromProcesses (initialXBPostR p)
              zeroProbFlux)
268   R.addEntityInstance
269         (caTropH `R.inCompartment` cardiacMuscleSite)
270         (R.entityFromProcesses (initialCaTropH p)
              zeroProbFlux)
271   R.addEntityInstance
272         (caTropL `R.inCompartment` cardiacMuscleSite)
273         (R.entityFromProcesses (initialCaTropL p)
              zeroProbFlux)
```

```
275   -- Clamped so the relative amounts add to 1.0...
276   R.addEntityInstance
277         (tmNXB `R.inCompartment` cardiacMuscleSite)
278         (R.entityClamped normalisedPermissive)
279
280 externalEntityIdentification = do
281   tmPXBMuscle <- tmPXB `R.inCompartment`
          cardiacMuscleSite
282   xbPreRMuscle <- xbPreR `R.inCompartment`
          cardiacMuscleSite
283   xbPostRMuscle <- xbPostR `R.inCompartment`
          cardiacMuscleSite
284   preRMuscle <- xbPreR `R.inCompartment`
          cardiacMuscleSite
285   postRMuscle <- xbPostR `R.inCompartment`
          cardiacMuscleSite
286   return (tmPXBMuscle, xbPreRMuscle, xbPostRMuscle,
          preRMuscle, postRMuscle)
287
288 standardRate :: Monad m => ReactionParameters m -> RExB
      m -> RExB m
289 standardRate (ReactionParameters{baseRate=baseRate,
      otherMod=otherMod,speciesMod=speciesMod,q10=q10})
      temp = do
290     let l = catMaybes [Just baseRate, otherMod,
          speciesMod]
291     foldl' (.*.) (q10 .**. ((temp .-. U.realConstant
          uCelsius 37) ./. U.dConstant 10)) l
292
293 standardTransient :: Monad m => TransientParameters m ->
      RExB m -> RExB m
294 standardTransient p@(TransientSpike {}) t = do
295   timeRatio <- U.realCommonSubexpression ((
          transientTime1 p) ./. (transientTime2 p))
296   t' <- U.realCommonSubexpression (t .-. (
          transientStartTime p))
297   let dim1 = U.dConstant 1
298   let dimm1 = U.dConstant (-1)
299   let beta = timeRatio .**. (dimm1 ./. (timeRatio .-.
          dim1)) .-.
300               timeRatio .**. (dimm1 ./. (dim1 .-. (
                    transientTime2 p) ./. (transientTime1 p)
                    ))
301   U.ifX (t .<=. (transientStartTime p))
302     {- then -} (transientBase p)
303     {- else -} $ ((transientAmplitude p .-.
          transientBase p) ./. beta) .*.
304                   (U.expX (U.negateX (t' ./. (
                        transientTime1 p))) .-.
305                    U.expX (U.negateX (t' ./. (
                        transientTime2 p))))
306                   .+. (transientBase p)
307 standardTransient p@(TransientExpRamp initial c) t =
308   initial .*. U.expX (t .*. c)
```

The standardRate function, defined on lines 288-291, computes a reaction rate from the ReactionParameters; the form baseRate $\times$ otherMod $\times$ speciesMod $\times$ q10$^{\frac{\text{temp}-37}{10}}$ is used throughout the Rice et al. [2008] paper that the model is based on.

The standardTransient function defines the relation between a TransientParameters structure and time, and the value of the transient. It is defined using two separate formulae, one for each TransientParameters constructor. The formula for the standardTransient for a TransientSpike, starting on line 294, follows the equation for Calcium given in the paper, creating a sharp but continuous and differentiable transient at a given time point. The formula for when the TransientParameters is a TransientExpRamp, starting one line 307, on the other hand, describes a simple exponential relationship with time.

Lines 311-317 define a number of functions related to sarcomere geometry, in the same form they are described in the paper (but with more descriptive variable names).

The expression for the passive force due to titin is defined on lines 319-329 in the function titinForce. This force provides a good example of the use of $U.ifX$ to define conditional functions. A similar definition is given for the passive force due to collagen, on lines 330-337.

The passive force is defined on lines 339-348. This makes use of the titin force and the collagen force (but conditional on the modelContextParameter). The Haskell case syntax is used here to distinguish between IsolatedCell and Trabeculae contexts.

The preload force (line 350) is computed by applying the passive force function based on the initial sarcomere length, while the afterload force is computed (lines 351-354) from the parameters depending on the type of contraction.

The calciumBindingToTroponin site is an example of a function that computes a ProcessBuilder. When a model contains more than one Process that is mostly the same, but with differences in details, functions can be used to factor out the differences from the parts that remain constant. In the Rice et al. [2008] model,

```
310 -- Functions for sarcomere geometry...
311 singleOverlapNearestZ p x = U.minX (thickFilamentLength
        p) x ./. U.dConstant 2
312 singleOverlapNearestCentreLine p x = U.maxX (x ./. U.
        dConstant 2 .-.
313                                                 (x .-.
                                                        thinFilamentLength
                                                         p))
314                                               (hbareLength p
                                                    ./. U.
                                                    dConstant 2)
315 lengthSingleOverlap p x = singleOverlapNearestZ p x .-.
        singleOverlapNearestCentreLine p x
316 singleOverlapThick p x = (U.dConstant 2 .*.
        lengthSingleOverlap p x) ./. (thickFilamentLength p
        .-. hbareLength p)
317 singleOverlapThin p x = lengthSingleOverlap p x ./.
        thinFilamentLength p
318 -- Functions for normalised passive force...
319 titinForce p x = U.ifX (x .>=. restingSarcomereLength p)
320                   {- then -}
321                     (passiveTitinConstant p .*.
322                        (U.expX (passiveTitinExponent p
                               .*.
323                                    (x .-.
                                          restingSarcomereLength
                                           p)) .-.
324                         U.dConstant 1))
325                   {- else -}
326                   (U.negateX $ passiveTitinConstant p
                          .*.
327                        (U.expX (passiveTitinExponent p
                               .*.
328                                     (restingSarcomereLength p
                                          .-. x)) .-.
329                         U.dConstant 1))
330 collagenForce p x = {- U.ifX (x .>=.
        sarcomereLengthCollagen p) If not in XPP code -
        causes instabililty -}
331                       {- then -}
332                       (passiveCollagenConstant p .*.
333                          (U.expX (passiveCollagenExponent
                                p .*.
334                                     (x .-.
                                          sarcomereLengthCollagen
                                           p)) ) {- .-.
335                          U.dConstant 1 Note: -1 in paper,
                                not in author's XPP code -})
336                       {- else -}
337                       {- (U.realConstant uNormalisedForce
                            0) -}
```

```
339 passiveForce p x =
340   do
341     vtitinForce <- U.mkNamedRealVariable
            uNormalisedForce "Titin Force"
342     vtitinForce `U.newEq` (titinForce p x)
343     vcollagenForce <- U.mkNamedRealVariable
            uNormalisedForce "Collagen Force"
344     vcollagenForce `U.newEq` (collagenForce p x)
345     case (modelContext p)
346       of
347         IsolatedCell -> vtitinForce
348         Trabeculae -> vtitinForce .+. vcollagenForce
349
350 preloadForce p = passiveForce p (initialSarcomereLength
      p)
351 afterloadForce p = case (contractionType p)
352                    of
353                      Isometric -> U.realVariable
                             stiffnessV .*. (
                             initialSarcomereLength p .-. U.
                             realVariable sarcomereLength)
354                      Isotonic -> constantAfterload p
355
356 calciumBindingToTroponinSite p site compartment = do
357   cavar <- R.addEntity R.EssentialForProcess R.
          CantBeCreatedByProcess R.NotModifiedByProcess 0 (
          calcium `R.withCompartment` compartment)
358   sitevar <- R.addEntity R.NotEssentialForProcess R.
          CanBeCreatedByProcess R.ModifiedByProcess 1 (site `
          R.withCompartment` compartment)
359   let calciumTroponinBindingRateT = standardRate (
          calciumOnTrop p) (temperature p)
360   R.rateEquation $ calciumTroponinBindingRateT .*. cavar
          .*. (U.realConstant uProbability 1 .-.  sitevar)
361
362 calciumDisassociatingTroponinSite p rp site compartment
      = do
363   sitevar <- R.addEntity R.EssentialForProcess R.
          CantBeCreatedByProcess R.ModifiedByProcess (-1) (
          site `R.withCompartment` compartment)
364   R.rateEquation $ (standardRate rp (temperature p)) .*.
          sitevar
365
366 tropRegulatory p catroph catropl = (U.dConstant 1 .-.
      singleOverlapThin p (U.realVariable sarcomereLength))
       .*. catropl .+.
367                                      singleOverlapThin p (
                                             U.realVariable
                                             sarcomereLength)
                                             .*. catroph
368 permissiveTotal p catroph catropl = (U.dConstant 1 ./. (
      U.dConstant 1 .+. (permissiveHalfActivationConstant p
       ./. tropRegulatory p catroph catropl).**.
      permissiveHillCoefficient p)) .**. U.dConstant 0.5
```

calcium binds to two different tropomyosin sites at the same rate, and this function is used to describe the process for both (after the Entity representing tropomyosin with calcium bound to the site is passed in as an argument). Note that this particular model is peculiar in that it models these quantities as proportions, and so there is no explicit reactant represented in the Process. For models being built from scratch, it would be more natural to represent everything in terms of concentrations and include an Entity for tropomyosin without calcium bound to any site (as well as allowing for the possibility of better integrating with a calcium dynamics model).

A similar function is provided for calcium disassociating from the troponin site, on line 362. In this case, the dissociation rate is different depending on the site, so the ReactionParameters is also passed as an argument to the function.

Line 366 defines the amount of regulatory troponin, in terms of the Parameters and the amount of troponin with calcium bound to the low and high affinity sites. Likewise, permissiveTotal, defined on line 368 estimates the total proportion of permissive tropomyosin.

Line 369 defines the inverse of the permissive total, with small values restricted to ensure the value never falls below 100 (as in Rice et al. [2008]; avoiding numerical issues).

The nToP function on lines 371-376 uses the functions that have just been defined to describe a Process from a non-permissive to a permissive form. The specific Entities for the permissive entity and the non-permissive entity are arguments to the function. Likewise, a similar function pToN is defined for the reverse transition.

Partial function application is used on lines 385-388 to create functions for specific variables, namely the variants close to cross-bridge sites, and those not close to cross-bridge sites.

The function pToXBPreR (line 390) defines an all Compartment Process where permissive tropomyosin near a cross-bridge forms pre-rotation cross-bridges.

The function xbToPermissive (line 395) describes the reverse process; cross-

```
369 inversePermissiveTotal p catroph catropl = U.minX (U.
      dConstant 1 ./. permissiveTotal p catroph catropl) $
     U.dConstant 100
370
371 nToP pent nent p c = do
372   pvar <- R.addEntity R.NotEssentialForProcess R.
        CanBeCreatedByProcess R.ModifiedByProcess 1 (pent `
        R.withCompartment` c)
373   nvar <- R.addEntity R.EssentialForProcess R.
        CantBeCreatedByProcess R.ModifiedByProcess (-1) (
        nent `R.withCompartment` c)
374   catroph <- R.addEntity R.NotEssentialForProcess R.
        CantBeCreatedByProcess R.NotModifiedByProcess 0 (
        caTropH `R.withCompartment` c)
375   catropl <- R.addEntity R.NotEssentialForProcess R.
        CantBeCreatedByProcess R.NotModifiedByProcess 0 (
        caTropL `R.withCompartment` c)
376   R.rateEquation $ (standardRate ((tropomyosinNToP p) {
        otherMod = Just $ permissiveTotal p catroph catropl
        }) (temperature p)) .*. nvar
377
378 pToN pent nent p c = do
379   pvar <- R.addEntity R.EssentialForProcess R.
        CantBeCreatedByProcess R.ModifiedByProcess (-1) (
        pent `R.withCompartment` c)
380   nvar <- R.addEntity R.NotEssentialForProcess R.
        CanBeCreatedByProcess R.ModifiedByProcess 1 (nent `
        R.withCompartment` c)
381   catroph <- R.addEntity R.NotEssentialForProcess R.
        CantBeCreatedByProcess R.NotModifiedByProcess 0 (
        caTropH `R.withCompartment` c)
382   catropl <- R.addEntity R.NotEssentialForProcess R.
        CantBeCreatedByProcess R.NotModifiedByProcess 0 (
        caTropL `R.withCompartment` c)
383   R.rateEquation $ (standardRate ((tropomyosinPToN p) {
        otherMod = Just $ inversePermissiveTotal p catroph
        catropl}) (temperature p)) .*. pvar
384
385 nToPNotNearXB = nToP tmPNoXB tmNNoXB
386 pToNNotNearXB = pToN tmPNoXB tmNNoXB
387 nToPNearXB = nToP tmPXB tmNXB
388 pToNNearXB = pToN tmPXB tmNXB
389
390 pToXBPreR p c = do
391   pvar <- R.addEntity R.EssentialForProcess R.
        CantBeCreatedByProcess R.ModifiedByProcess (-1) (
        tmPXB `R.withCompartment` c)
392   xbprervar <- R.addEntity R.NotEssentialForProcess R.
        CanBeCreatedByProcess R.ModifiedByProcess 1 (xbPreR
         `R.withCompartment` c)
393   R.rateEquation $ (standardRate (crossBridgeFormation p
        ) (temperature p)) .*. pvar
```

```
395 xbToPermissive rate fromXB p c = do
396   xbvar <- R.addEntity R.EssentialForProcess R.
          CantBeCreatedByProcess R.ModifiedByProcess (-1) (
          fromXB `R.withCompartment` c)
397   pvar <- R.addEntity R.NotEssentialForProcess R.
          CanBeCreatedByProcess R.ModifiedByProcess 1 (tmPXB
          `R.withCompartment` c)
398   R.rateEquation $ rate .*. xbvar
399
400 xbPreRToPermissiveRate p =
401     standardRate
402       ((crossBridgeDissociation p){
403          otherMod=Just (U.dConstant 1 .+.
404                         (U.dConstant 1 .-.
                                singleOverlapThick p (U.
                                realVariable sarcomereLength)
                                ) .*.
405                         (overlapModStrongToWeak p))})
406       (temperature p)
407
408 xbPreRToPermissive p c =
409   xbToPermissive (xbPreRToPermissiveRate p) xbPreR p c
410
411 xbPostRToPermissiveRate p =
412     let
413         mod = U.ifX (U.realVariable meanDistortionPostR
                .<. meanStrain p)
414                 {- then -} (U.expX $
                                strainEffectPositive p .*.
415                                 (((meanStrain p .-. U.
                                    realVariable
                                    meanDistortionPostR)
                                     ./. meanStrain p)
                                    .**.U.dConstant 2))
416                 {- else -} (U.expX $
                                strainEffectNegative p .*.
417                                 (((meanStrain p .-. U.
                                    realVariable
                                    meanDistortionPostR)
                                     ./. meanStrain p)
                                    .**.U.dConstant 2))
418     in
419       standardRate ((rotatedCrossBridgeDissociation p){
420                       otherMod=Just mod }) (temperature
                            p)
421
422 xbPostRToPermissive p c =
423       xbToPermissive (xbPostRToPermissiveRate p) xbPostR
              p c
424
425 crossBridgeRotationRate p = standardRate ((
      crossBridgeRotation p){otherMod = Just . U.expX $ (U.
      negateX (U.signX $ U.realVariable meanDistortionPreR)
      ) .*. preRotStrainFactor p .*. (U.realVariable
      meanDistortionPreR ./. meanStrain p).**.U.dConstant
      2}) (temperature p)
```

bridges dissociate to release permissive tropomyosin. However, in this case, dissociation can occur from either the pre- or post- rotation cross-bridge state, and so the function is expressed with arguments for the rate constant and the applicable cross-bridge Entity.

The function xbPreRToPermissiveRate (line 400) describes how to compute the rate for pre-rotation cross-bridges, while xbPostRToPermissiveRate (line 411) computes the corresponding rate for post-rotation cross-bridges. The functions xbPreRToPermissive (line 308) and xbPostRToPermissive (line 422) use xbToPermissive with the appropriate rates and parameters to create the specialised form.

Line 425 defines crossBridgeRotationRate, for computing the rate at which cross-bridges rotate, given the Parameters structure.

These rotation rates are then used to define the all Compartment ProcessBuilder crossBridgePreToPost on lines 426-429, defining the Process for cross-bridge rotation. The reverse rotation process is then defined on lines 430-433.

The default ReactionParameters are then defined on line 435; this definition is a simple empty definition from which other definitions can be derived. The default Parameters are defined starting on line 444, using the parameter values in Rice et al. [2008].

This model can be used to reproduce figures in the paper. By changing the Parameters to match the type of figure required, and using the command-line option to override specific variables like sarcomere length, different figures from the paper are produced.

Figure 3A in Rice et al. [2008] shows a series of curves showing the isosarcometric relationship between the calcium concentration and normalised active force. Rather than performing a sampling sensitivity analysis, the authors simply set up an exponential ramp protocol for calcium of $[Ca^{2+}] = 0.0002e^{0.05t}$, where t is time, and plotted $[Ca^{2+}]$ against normalised active force. This protocol gives different results to sampling based sensitivity analysis, but it has been used here in the interests of reproducing the results in the paper. The figure

```
426 crossBridgePreToPost p c = do
427   xbPreRvar <- R.addEntity R.EssentialForProcess R.
        CantBeCreatedByProcess R.ModifiedByProcess (-1) (
        xbPreR `R.withCompartment` c)
428   xbPostRvar <- R.addEntity R.NotEssentialForProcess R.
        CanBeCreatedByProcess R.ModifiedByProcess 1 (
        xbPostR `R.withCompartment` c)
429   R.rateEquation $ crossBridgeRotationRate p .*.
        xbPreRvar
430 crossBridgePostToPre p c = do
431   xbPreRvar <- R.addEntity R.NotEssentialForProcess R.
        CanBeCreatedByProcess R.ModifiedByProcess 1 (xbPreR
         `R.withCompartment` c)
432   xbPostRvar <- R.addEntity R.EssentialForProcess R.
        CantBeCreatedByProcess R.ModifiedByProcess (-1) (
        xbPostR `R.withCompartment` c)
433   R.rateEquation $ standardRate (
        crossBridgeReverseRotation p) (temperature p) .*.
        xbPostRvar
434
435 defaultReactionParameters =
436     ReactionParameters {
437       baseRate=U.realConstant (uNthOrderRate 0) 0,
438       otherMod=Nothing,
439       speciesMod=Nothing,
440       q10=U.dConstant 1
441     }
442
443 defaultParameters :: Monad m => Parameters m
444 defaultParameters =
445   Parameters {
446     maxSarcomereLength = U.realConstant uDistance 2.4,
447     minSarcomereLength = U.realConstant uDistance 1.4,
448     thickFilamentLength = U.realConstant uDistance 1.65,
449     hbareLength = U.realConstant uDistance 0.1,
450     thinFilamentLength = U.realConstant uDistance 1.2,
451     temperature = U.realConstant uCelsius 22.5,
452     calciumOnTrop = defaultReactionParameters {
453        baseRate=U.realConstant (uNthOrderRate 1) 5E4,
454        q10=U.dConstant 1.5 },
455     calciumOffTropL = defaultReactionParameters {
456        baseRate=U.realConstant (uNthOrderRate 0) 250,
457        q10=U.dConstant 1.3 },
458     calciumOffTropH = defaultReactionParameters {
459        baseRate=U.realConstant (uNthOrderRate 0) 25,
460        q10=U.dConstant 1.3 },
461     tropomyosinNToP = defaultReactionParameters {
462        baseRate=U.realConstant (uNthOrderRate 0) 500,
463        q10=U.dConstant 1.6 },
464     tropomyosinPToN = defaultReactionParameters {
465        baseRate=U.realConstant (uNthOrderRate 0) 50,
466        q10=U.dConstant 1.6 },
```

```
467     crossBridgeFormation = defaultReactionParameters {
468        baseRate=U.realConstant (uNthOrderRate 0) 500,
469        q10=U.dConstant 6.25 },
470     crossBridgeDissociation = defaultReactionParameters
           {
471        baseRate=U.realConstant (uNthOrderRate 0) 70,
472        q10=U.dConstant 2.5 },
473     crossBridgeRotation = defaultReactionParameters {
474        baseRate=U.realConstant (uNthOrderRate 0) 2000,
475        q10=U.dConstant 6.25 },
476     crossBridgeReverseRotation =
           defaultReactionParameters {
477        baseRate=U.realConstant (uNthOrderRate 0) 400,
478        q10=U.dConstant 6.25 },
479     rotatedCrossBridgeDissociation =
           defaultReactionParameters {
480        baseRate=U.realConstant (uNthOrderRate 0) 70,
481        q10 = U.dConstant 6.25 },
482     permissiveHalfActivationConstant = U.dConstant 0.5,
483     permissiveHillCoefficient = U.dConstant 15,
484     overlapModStrongToWeak = U.dConstant 6,
485     preRotStrainFactor = U.dConstant 5,
486     strainEffectPositive = U.dConstant 8,
487     strainEffectNegative = U.dConstant 1,
488     meanStrain = U.realConstant uDistance 0.007,
489     strainScalingFactor = U.dConstant 2,
490     restingSarcomereLength = U.realConstant uDistance
           1.9,
491     passiveTitinConstant = U.realConstant
           uNormalisedForce 0.002,
492     passiveTitinExponent = U.dConstant 10,
493     sarcomereLengthCollagen = U.realConstant uDistance
           2.25,
494     passiveCollagenConstant = U.realConstant
           uNormalisedForce 0.02,
495     passiveCollagenExponent = U.dConstant 70,
496     normalisedMass = U.realConstant uNormMass 0.00005,
497     normalisedViscosity = U.realConstant uNormViscosity
           0.003,
498     constantAfterload = U.realConstant uNormalisedForce
           0.001,
499     stiffness = U.realConstant uNormStiffness 50.0,
500     initialSarcomereLength = U.realConstant uDistance
           2.2,
501     initialtmNNoXB = U.realConstant uProbability 0.9999,
502     initialtmPNoXB = U.realConstant uProbability 0.0001,
503     initialtmPXB = U.realConstant uProbability 0,
504     initialXBPreR = U.realConstant uProbability 0.0001,
505     initialXBPostR = U.realConstant uProbability 0.0001,
506     initialxXBPreR = U.realConstant uDistance 0,
507     initialxXBPostR = U.realConstant uDistance 0.007,
508     initialCaTropH = U.realConstant uProbability 0.001,
```

```
509     initialCaTropL = U.realConstant uProbability 0.001,
510     calciumTransient = {- TransientSpike {
511                             transientStartTime = U.
                                    realConstant uSecond 0.0,
512                             transientBase = U.realConstant
                                    uConcentration 0.09,
513                             transientAmplitude = U.
                                    realConstant uConcentration
                                    1.45,
514                             transientTime1 = U.realConstant
                                     uSecond 0.02,
515                             transientTime2 = U.realConstant
                                     uSecond 0.11 } -}
516       TransientExpRamp (U.realConstant uConcentration
              0.0002) (U.realConstant uFlux 0.05),
517     modelContext = Trabeculae,
518     contractionType = Isometric
519   }
```

produced from the ModML version of this model is shown in Figure 10.6.

Figure 3B in Rice et al. [2008] shows a similar series of curves, but this time contracting against a spring, with variation in the stiffness of a spring connected to the filament in series. This figure can also be reproduced, and is also shown in Figure 10.6.

These examples show that it is possible to build useful models based on differential-algebraic equations with ModML Units and ModML Reactions.

10.6 Creating derivative models

In section 10.5, it was shown that ModML Units and ModML Reactions can be used to build a model based on the equations in a paper.

Another important aspect of building models is the ability to build upon and extend models that have previously been coded up. The simplest approach is to simply copy a model representation and make changes to it. However, as the second file diverges more from the original model, it becomes progressively harder to automatically merge changes to the original model into the changed

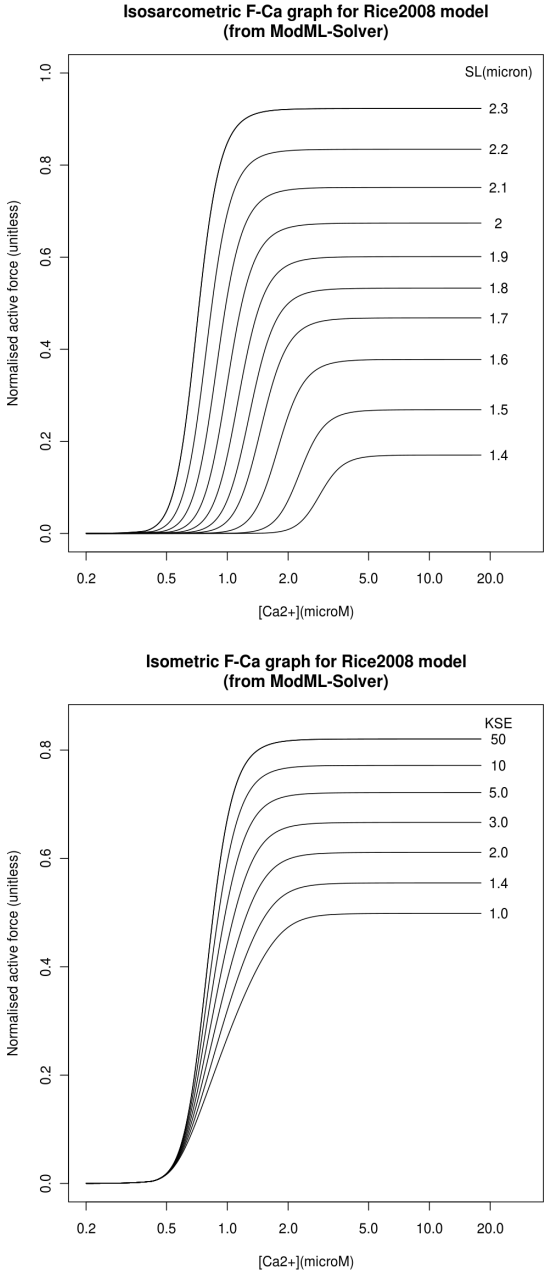

Figure 10.6: A figure showing results from the ModML version of the Rice et al. model, reproducing the results shown in Figures 3A and 3B of that paper. In the first figure, the sarcomere length is fixed at a series of different values; in the second, stiffness of a spring in series (KSE) with the filament is fixed at a series of different values. A single time-course simulation is run for each value, with an exponential $[Ca^{2+}]$ ramp. Normalised Active Force is plotted against Amount of Calcium.

model, especially when there are multiple levels of modified versions of modified versions. In addition, if two models make different modifications to an original base model, and a model including both modifications is desired, it becomes necessary to merge the changes.

Storing models in a distributed version control system can help to manage this complexity, as described in Miller et al. [2011]. However, wherever possible, it would be better to avoid the need to do this, and describe models by reference to the base model with specified changes made. Version Control Systems typically describe changes in a domain-neutral manner; what is needed is a way to describe changes in a more domain-specific manner, so that rules for combining parts of models can be more accurate.

ModML Reactions and ModML Units already provide a natural way of adding two models to each other, namely monadic composition. In a state monad, it is also possible to perform subtractive changes; mechanisms for doing so are discussed below.

The Tran et al. [2010] model extends the Rice et al. [2008] model coded up in section 10.5, by adding metabolite sensitivity to the model. This makes the model an ideal candidate to demonstrate that it is possible to extend ModML models.

The ModML version of the Tran et al. [2010] model imports the Rice2008 ModML model, and then redefines model, parameterisedModel, unitsModel, unitsModelBeforeReaction, and unitsModelAfterReaction. A Parameters data type is also defined, including the Rice2008 Parameters as a constructor argument, but also defining the new parameters in the Tran model.

The Reaction model is converted to ModML Units by composing both the Rice and Tran R.ModelBuilder monads, as follows:

```
(cem, _, ces) <-
  R.runReactionBuilderInUnitBuilder' (
    do
      Rice.reactionModelWithCalciumTransient fromPre p
      reactionModel p
```

```
)
```

The reactionModel function adds a Process that isn't present in the Rice model (pre-rotation cross-bridges converting back to the permissive state). However, there are also a number of Processes that are modified from Rice2008. The approach taken was to remove the Processes that have been changed, and then add new Processes to replace them.

To remove Processes, three functions have been added to ModML Reactions to support subtractive modelling: removeExplicitCompartmentProcessesInvolving, removeAllCompartmentProcessesInvolving, and removeContainedCompartmentProcessesInvolving. The explicit Compartment form takes a list of lists of R.ModelBuilder (CompartmentEntity, Double). Each entry in the top-level list is a description of a Process to remove. The second-level list describes all the CompartmentEntities that must be present in the stoichiometry map for the Process to match on a list, as well as the the value the stoichiometry for that CompartmentEntity must match. The signature for removeAllCompartmentProcessesInvolving is the same except the CompartmentEntity is replaced with just an Entity; a place-holder Compartment is provided to the Process, and each Entity is paired with that placeholder Compartment to determine if there is a match. The same signature is used for removeContainedCompartmentProcessesInvolving, but the implementation uses two placeholder Compartments, and each Entity is paired with each Compartment in turn; at least one of the two CompartmentEntities created this way for each Entity must have a stoichiometry map entry matching the specified stoichiometry.

The reactionModel makes use of removeAllCompartmentProcessesInvolving as follows:

```
R.removeAllCompartmentProcessesInvolving
    [[(Rice.xbPreR, -1), (Rice.tmPXB, 1)],
     [(Rice.xbPostR, -1), (Rice.xbPreR, 1)],
     [(Rice.xbPostR, -1), (Rice.tmPXB, 1)]]
```

Metabolite sensitive versions of these processes are then added back into the

model.

10.7 Discussion and Future Directions

The ModML Units and ModML Reactions modules presented in this chapter allow models from a number of different problem domains to be represented. One area of development that would make ModML more useful would be the development of a wide range of other domain-specific translations, building on top of ModML Core, ModML Units, or ModML Reactions. As long as these systems can ultimately be represented as systems of differential-ordinary equations, no changes to ModML Core or the solver libraries are required; instead, domain-specific translation modules like ModML Units and ModML Reactions should be considered part of the model and not part of language specification or the solver tool. Extensions to the core, such as to allow spatial modelling, would increase the range of DSLs possible even further.

Libraries providing domain-specific support for mechanical primitives, population dynamics, economic modelling, and explicitly discretised spatial models could potentially be added and composed with each other to allow easier multi-scale and multi-domain modelling.

The process and entity model used in ModML Reactions is intentionally general. However, in future, more domain specific ways of representing reactions could be useful. For example, paralogy (gene duplication), post-transcriptional and translational modifications, and substrate binding can mean that there are different variants of a protein, where there is a difference between variants for some processes (involving some sites on the protein, for example), but not for other ones. This means there could potentially be a proliferation of different Entities, where the distinction between Entities doesn't matter for Processes at a particular site. Using plain ModML Reactions, the way that a protein is divided up into Entities matters greatly when building a model; if the Entities all act as enzymes with identical rate constants, the Process needs to include all the Entities and add the amounts together to compute a total. The situation is even more complex if substrates are involved. For example, suppose that

phosphorylation at site two is independent of phosphorylation at site one. Then two Processes are needed for phosphorylation at site two: phosphorylating the protein with site one unphosphorylated, giving a product with only site two phosphorylated, and phosphorylating a protein with site one phosphorylated, giving a product with site two phosphorylated. If the model was composed with another model that considered a third independent site, all parts of the model involving that protein would need to be revised. However, it would be possible to define a DSL that is aware of such modifications, using additional metadata about the dependence of modifications on each other, allowing much greater composability.

Another future direction would be to improve the reversibility of the translations by adding additional annotations. Reversible translations are useful for several reasons. Firstly, it would mean that rather than needing to compose all ModML Reactions models to each other before composing them with the rest of the model, it would be possible to take a ModML Units or ModML Core model data structure, re-generate the ModML Reactions builder, and compose that builder with another Reactions builder. In addition, reversibility would make the development of tools to edit ModML in a format other than text possible without losing information about problem domains the tool doesn't understand; tools would convert the model to ModML Core, and other tools would be able to regenerate, say, ModML Reactions if that needed to be edited and the equations for the reactions were still valid after editing the model. Because domain-specific specifications already contain all the information to be stored in the model, making these translations reversible could be done simply by changing the translation.

ModML Reactions is an example of a domain specific language that could be useful for more than translation to ModML Units. For example, the ModML Reactions Processes and CompartmentEntities could be translated into a stochastic simulation of the reaction network rather than an initial value DAE problem.

10.8 Conclusions

ModML allows a wide range of domain-specific modules for representing models to be developed; the examples presented in this section show how units and reactions models can be represented and translated into ModML Core. Figure 10.7 shows where some of the DSLs and example models presented in this chapter fit together with ModML Core and the solver tools presented in the previous chapter.

One aspect that would improve the usability of ModML and ModML domain specific languages in the future would be the development of tools that allowed ModML models to be manipulated graphically. Such tools could either work at the level of ModML Core, or have knowledge about particular domain specific languages. Reversibility of translations into ModML Core would mean that domain-specific tools could work on the metadata, rather than the original data structures; this would have the benefit that editing a model in a generic tool would not remove domain specific information.

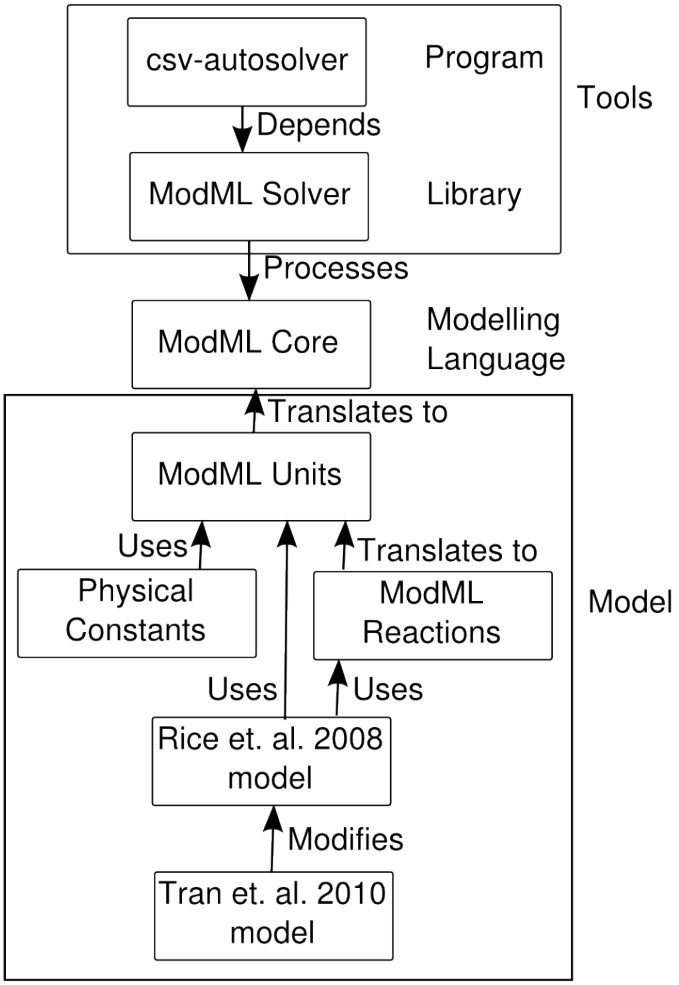

Figure 10.7: An overview of how some of the models presented in this chapter fit together with the domain-specific languages (DSLs) present here and ModML Core and the associated solvers presented in Chapter 9

Chapter 11

Summary and Conclusions

11.1 Building GRN models

In Chapter 2, methods for building gene regulatory networks from expression microarray data were reviewed.

In Chapter 6, a method for building a gene regulatory network from putative transcription factor binding sites was presented. A novel method for predicting the location of these sites was presented in Chapter 3. These methods, taken together, allow a putative gene regulatory network to be built, using information on the binding specificities of transcription factors and the genome sequence.

In Chapter 5, a method for predicting which regulated genes are missing regulators was introduced. This method uses support vector regression to convert a qualitative gene regulatory network into a quantitative one, and uses these gene regulatory networks along with an iterative algorithm to predict the expression of all genes on each expression microarray. Leave-one-out cross-validation is then used to assess an error for each transcription factor, and the errors are then used to detect which transcription factors are most likely to be missing a regulator. By using an error cut-off, the method can be used to classify whether a regulator is missing, and by adjusting the cut-off, the trade-off between sensitivity and selectivity can be adjusted. When this method was tested using a model where edges had been deleted, to detect where edges had been deleted, it detected missing edges better than would be expected by chance for a range of different cut-offs.

11.2 Validating GRN models

In Chapter 6, a method for validating gene regulatory networks against expression microarray data was presented. The method presented firstly utilises Support Vector Regression to build a quantitative model of the expression levels from a qualitative model using a subset of the expression microarrays, and then tests the quantitative model against a remaining subset of the data.

This experiment showed one of the challenges of determining gene regulation based on expression microarray data, as overall, in human, scrambling the models in various ways improved the fit of the model to the data.

11.3 Representing mathematical models

In Chapter 7, methods for representing mathematical models were reviewed, with a particular focus on methods for representing models consisting of systems of differential-algebraic equations.

In Chapter 8, an application programming interface (API) for working with mathematical models in CellML was presented. The API provides for both basic manipulation of the structure of CellML models, and higher level services providing functionality to more easily deal with compartments, imports and physical units, validate models against the CellML specification, convert models into imperative code, and work with annotations. This API facilitates the use of CellML for model exchange, as evidenced by its use in the work described in Chapter 5 for working with models of gene regulatory networks.

In Chapter 9, a novel approach for representing models was presented, along with a language implementing this approach (called ModML). A functional programming language is used to represent a transformation from a domain specific language (DSL) representation into data structures representing the model, along with annotations that aid in interpreting results from the model. In the same Chapter, methodologies and tools for generating results from ModML models were also presented.

The utility of the approach described in Chapter 9 is demonstrated in Chapter 10, where several DSLs are defined using the functionality of ModML. Firstly, a ModML DSL for representing expressions alongside physical units, with validation for units and dimensional consistency, was presented. This language was used to define the standard SI units, and also a set of physical units.

The ModML Units DSL was then used as a basis for a more specific DSL, ModML Reactions, for systems of reactions (described in terms of concepts called amounts, compartments, entities, and processes). Algorithms for converting from ModML Reactions to ModML Units were presented, along with two examples demonstrating how ModML Reactions can be used. The second example was a translation of an existing model into ModML, and described cross-bridge cycling in cardiac muscle, showing the utility of combining different DSLs (ModML Reactions and ModML Units, in this case) to build a model. This model was extended to describe another model derived from the first model, showing that the approach taken by ModML allows for re-usability of models without manually duplicating information from the original model.

11.4 Future Directions

GRN models

One of the limiting factors currently restricting the development of accurate GRN models in human is the shortage of readily accessible data on transcription factor binding specificity in human. The development of comprehensive data sets for all human transcription factors, as has been done in yeast, would greatly facilitate understanding of gene regulation in human. Due to the large numbers of transcription factors in human, and the differing expression profiles between different types of cell, obtaining this data set will require a large amount of effort.

Another major limiting factor is the limited view available of gene expression. Throughout this thesis, gene expression levels of both regulators and regulated genes have been measured through expression microarray readings. These

readings measure cDNA, which has been synthetically reverse transcribed from the mRNA present in the cell. This introduces several important limitations. Transcriptional regulation is only one of the many processes controlling the expression of genes. The fact that an mRNA transcript for a transcription factor is abundant does not necessarily provide much information about the actual levels of the transcription factor proteins.

In addition, methods for working with metabolic networks and gene regulatory networks are currently treated as entirely separate sub-fields within systems biology. However, the complex emergent behaviours that cells can exhibit almost certainly involve indirect interactions between gene regulation and the metabolic activity of the cell. To fully understand these emergent behaviours, it will be necessary to model all metabolic activities in cells. This will require the simultaneous knowledge of the levels of transcripts and other metabolites in cells at a given time.

Another issue is the need to understand how expression levels vary with time. All of the methodologies presented in this thesis work with so-called steady state data, produced by measuring expression levels in a cell lysate from many cells, and so measure an average expression level across many cells. In practice, it is likely that there is considerable variation inside individual cells. Furthermore, one of the most important aspects of cell behaviour is change in response to environmental changes; again, to understand this, time course data will be required. However, useful time course data requires measuring expression level at a high sampling frequency, and so is expensive to collect; most time course microarray data-sets available at present only have a few time-points available. The introduction of time variation also creates the possibility that models will need to consider spatial variation rather than assuming cells are well-stirred reactors.

When working with data from multicellular organisms, it is likely that epigenetic changes from cell differentiation between different tissue types, as well as genetic variation between different populations of cells have an effect on gene regulation. Therefore, when sufficient data becomes available, gene regulatory networks could be separated by type of tissue, the presence of SNPs known to affect

regulation of certain genes, and the type of oncogenetic mutations in the cell populations sampled.

Mathematical model representation

The model representation approach presented in Chapters 9 and 10 could be extended in a number of ways, as discussed in more detail in section 10.7.

One of the most important future developments is likely to be the development of DSLs for representing models from a wider range of problem domains, including more specific DSLs for transcriptional regulation and gene regulatory networks.

Another important future development would be the development of a library representing standard biological parts that could be assembled to rapidly build a model of a process. If many parts of biological systems were described in this way, they could be connected together to describe a model of an entire biological system (a project that is difficult for any one research group to do due to the number of different systems that need to be modelled). Such comprehensive models will allow observations of system behaviour from a holistic viewpoint to be combined with the reductionist models, allowing the emergent properties of the full system to be understood.

Another major future direction would be to expand the approach taken in ModML for other types of model, such as for spatially varying models of partial differential equations.

Bibliography

Albert and Barabasi. Topology of evolving networks: Local events and universality. *Phys Rev Lett*, 85:5234–5237, Dec 2000. ISSN 0031-9007.

M. Ashburner, C.A. Ball, J.A. Blake, D. Botstein, H. Butler, J.M. Cherry, A.P. Davis, K. Dolinski, S.S. Dwight, J.T. Eppig, et al. Gene Ontology: Tool for the unification of biology. *Nature genetics*, 25(1):25–29, 2000.

M. Madan Babu, Nicholas M. Luscombe, L. Aravind, Mark Gerstein, and Sarah A. Teichmann. Structure and evolution of transcriptional regulatory networks. *Curr Opin Struct Biol*, 14:283–291, Jun 2004. ISSN 0959-440X. doi: 10.1016/j.sbi.2004.05.004.

Timothy L. Bailey and Charles Elkan. Unsupervised Learning of Multiple Motifs in Biopolymers Using Expectation Maximization. *Mach. Learn.*, 21(1-2): 51–80, 1995. ISSN 0885-6125. doi: http://dx.doi.org/10.1007/BF00993379.

T.L. Bailey and M. Gribskov. Combining evidence using p-values: Application to sequence homology searches. *Bioinformatics*, 14(1):48, 1998.

T.L. Bailey, N. Williams, C. Misleh, and W.W. Li. MEME: Discovering and analyzing DNA and protein sequence motifs. *Nucleic Acids Research*, 34(Web Server issue):W369, 2006.

Barabasi and Albert. Emergence of scaling in random networks. *Science*, 286: 509–512, Oct 1999. ISSN 1095-9203.

Yaman Barlas. Formal aspects of model validity and validation in system dynamics. *System Dynamics Review*, 12:183–210, 1996. ISSN 1099-1727. URL `http://dx.doi.org/10.1002/(SICI)1099-1727(199623)12:3<183::AID-SDR103>3.0.CO;2-4`.

B.A. Barshop, R.F. Wrenn, and C. Frieden. Analysis of numerical methods for computer simulation of kinetic processes: Development of KINSIM–a

flexible, portable system. *Analytical biochemistry*, 130(1):134–145, 1983. ISSN 0003-2697.

D.A. Beard, R. Britten, M.T. Cooling, A. Garny, M.D.B. Halstead, P.J. Hunter, J. Lawson, C.M. Lloyd, J. Marsh, A. Miller, et al. CellML metadata standards, associated tools and repositories. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1895):1845, 2009. ISSN 1364-503X.

D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, B.A. Rapp, and D.L. Wheeler. GenBank. *Nucleic acids research*, 28(1):15, 2000. ISSN 0305-1048.

DI Bevan. Distributed garbage collection using reference counting. *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, pages 176–187, 1987.

Mathieu Blanchette, Benno Schwikowski, and Martin Tompa. Algorithms for phylogenetic footprinting. *J Comput Biol*, 9:211–223, 2002. ISSN 1066-5277. doi: 10.1089/10665270252935421.

B.J. Bornstein, S.M. Keating, A. Jouraku, and M. Hucka. LibSBML: An API Library for SBML. *Bioinformatics*, 24(6):880, 2008.

Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. A training algorithm for optimal margin classifiers. In *Computational Learing Theory*, pages 144–152, 1992. URL `citeseer.ist.psu.edu/boser92training.html`.

MS Branicky. Analog computation with continuous ODEs. In *Physics and Computation, 1994. PhysComp'94, Proceedings., Workshop on*, pages 265–274. IEEE, 2002. ISBN 081866715X.

T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0. *W3C recommendation*, 2000.

A. Brazma, I. Jonassen, J. Vilo, and E. Ukkonen. Predicting gene regulatory elements in silico on a genomic scale. *Genome Res*, 8:1202–1215, Nov 1998. ISSN 1088-9051.

M. P. Brown, W. N. Grundy, D. Lin, N. Cristianini, C. W. Sugnet, T. S. Furey, M. Ares, and D. Haussler. Knowledge-based analysis of microarray gene expression data by using support vector machines. *Proc Natl Acad Sci U S A*, 97:262–267, Jan 2000. ISSN 0027-8424.

P.N. Brown, A.C. Hindmarsh, and L.R. Petzold. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM Journal on Scientific Computing*, 15:1467, 1994.

E.A. Bruford, M.J. Lush, M.W. Wright, T.P. Sneddon, S. Povey, and E. Birney. The HGNC Database in 2008: A resource for the human genome. *Nucleic acids research*, 36(suppl 1):D445, 2008. ISSN 0305-1048.

J C Bryne, E Valen, M H Tang, T Marstrand, O Winther, I da Piedade, A Krogh, B Lenhard, and A Sandelin. JASPAR, the open access database of transcription factor-binding profiles: New content and tools in the 2008 update. *Nucleic Acids Res*, Nov 2007. doi: 10.1093/nar/gkm955. URL `http://www.hubmed.org/display.cgi?uids=18006571`.

J.C. Butcher. Implicit Runge-Kutta processes. *Mathematics of Computation*, 18(85):50–64, 1964. ISSN 0025-5718.

A. J. Butte and I. S. Kohane. Mutual information relevance networks: functional genomic clustering using pairwise entropy measurements. *Pac Symp Biocomput*, pages 418–429, 2000. ISSN 1793-5091.

D. Carlisle, P. Ion, R. Miner, and N. Poppelier. Mathematical markup language (mathml) version 2.0. *W3C Recommendation*, 21, 2001.

Jonathan M. Carlson, Arijit Chakravarty, and Robert H. Gross. BEAM: a beam search algorithm for the identification of cis-regulatory elements in groups of genes. *J Comput Biol*, 13:686–701, Apr 2006. ISSN 1066-5277. doi: 10.1089/cmb.2006.13.686.

Arijit Chakravarty, Jonathan M. Carlson, Radhika S. Khetani, Charles E. DeZiel, and Robert H. Gross. SPACER: identification of cis-regulatory elements with non-contiguous critical residues. *Bioinformatics*, 23:1029–1031, Apr 2007. ISSN 1460-2059. doi: 10.1093/bioinformatics/btm041.

J.M. Cherry, C. Adler, C. Ball, S.A. Chervitz, S.S. Dwight, E.T. Hester, Y. Jia, G. Juvik, T.Y. Roe, M. Schroeder, et al. SGD: Saccharomyces genome database. *Nucleic acids research*, 26(1):73, 1998. ISSN 0305-1048.

David Maxwell Chickering, David Heckerman, and Christopher Meek. Large-sample learning of bayesian networks is np-hard. *J. Mach. Learn. Res.*, 5: 1287–1330, 2004. ISSN 1533-7928.

W. S. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74:859–836, 1979.

William S. Cleveland and Susan J. Devlin. Locally weighted regression: An approach to regression analysis by local fitting. *Journal of the American Statistical Association*, 83(403):596–610, 1988. ISSN 01621459. URL `http://www.jstor.org/stable/2289282`.

William S. Cleveland, Susan J. Devlin, and Eric Grosse. Regression by local fitting : Methods, properties, and computational algorithms. *Journal of Econometrics*, 37(1):87–114, January 1988. URL `http://ideas.repec.org/a/eee/econom/v37y1988i1p87-114.html`.

Francis Crick. Central Dogma of Molecular Biology. *Nature*, 227:561–563, Aug 1970. URL `http://dx.doi.org/10.1038/227561a0`. 10.1038/227561a0.

L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982. ISBN 0897910656.

M. Das and H.K. Dai. A survey of DNA motif finding algorithms. *BMC Bioinformatics*, 8(Suppl 7):S21, 2007.

Eric H. Davidson, Jonathan P. Rast, Paola Oliveri, Andrew Ransick, Cristina Calestani, Chiou-Hwa Yuh, Takuya Minokawa, Gabriele Amore, Veronica Hinman, Cesar Arenas-Mena, Ochan Otim, C. Titus Brown, Carolina B. Livi, Pei Yun Lee, Roger Revilla, Alistair G. Rust, Zheng jun Pan, Maria J. Schilstra, Peter J. C. Clarke, Maria I. Arnone, Lee Rowen, R. Andrew Cameron, David R. McClay, Leroy Hood, and Hamid Bolouri. A genomic

regulatory network for development. *Science*, 295:1669–1678, Mar 2002. ISSN 1095-9203. doi: 10.1126/science.1069883.

R. Wayne Davies. Active RNA: RNA enzymes in RNA splicing and processing. *Bioscience Reports*, 4:707–727, Sep 1984. URL `http://dx.doi.org/10.1007/BF01128812`. 10.1007/BF01128812.

P.N. de Souza, R.J. Fateman, J. Moses, and C. Yapp. The Maxima Book. 2003.

Doxygen developers. Doxygen manual. `http://www.stack.nl/~dimitri/doxygen/manual.html`.

R. Edgar, M. Domrachev, and A.E. Lash. Gene Expression Omnibus: NCBI gene expression and hybridization array data repository. *Nucleic acids research*, 30(1):207, 2002. ISSN 0305-1048.

David Edwards. Non-linear normalization and background correction in one-channel cDNA microarray studies. *Bioinformatics*, 19:825–833, May 2003. ISSN 1367-4803.

M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *Proc Natl Acad Sci U S A*, 95:14863–14868, Dec 1998. ISSN 0027-8424.

I Elias. Settling the intractability of multiple alignment. *J Comput Biol*, 13(7):1323–1339, Sep 2006. doi: 10.1089/cmb.2006.13.1323. URL `http://www.hubmed.org/display.cgi?uids=17037961`.

A. Eulalio, E. Huntzinger, T. Nishihara, J. Rehwinkel, M. Fauser, and E. Izaurralde. Deadenylation is a widespread effect of miRNA regulation. *Rna*, 15(1):21, 2009. ISSN 1355-8382.

J.J. Faith, B. Hayete, J.T. Thaden, I. Mogno, J. Wierzbowski, G. Cottarel, S. Kasif, J.J. Collins, and T.S. Gardner. Large-scale mapping and validation of Escherichia coli transcriptional regulation from a compendium of expression profiles. *PLoS Biol*, 5(1):e8, 2007.

A. Finney and M. Hucka. Systems biology markup language: Level 2 and beyond. *Biochemical Society Transactions*, 31(6):1472–1473, 2003. ISSN 0300-5127.

Gary William Flake and Steve Lawrence. Efficient svm regression training with smo. *Machine Learning*, 2001. URL `citeseer.ist.psu.edu/flake01efficient.html`.

Stephen P A Fodor, Lubert Stryer, Michael C Pirrung, and J Leighton Read. Very large scale immobilized polymer synthesis. United States Patent 5,424,186, December 1991.

N. Friedman, M. Linial, I. Nachman, and D. Pe'er. Using Bayesian networks to analyze expression data. *J Comput Biol*, 7:601–620, 2000. ISSN 1066-5277. doi: 10.1089/106652700750050961.

F. Friedrichs and C. Igel. Evolutionary tuning of multiple SVM parameters. *Neurocomputing*, 64:107–117, 2005. ISSN 0925-2312.

D.L. Fry. The use of computers in physiologic diagnosis. *Medical Electronics, IRE Transactions on*, (4):269–273, 1960. ISSN 0097-1049.

D. J. Galas, M. Eggert, and M. S. Waterman. Rigorous pattern-recognition methods for DNA sequences. Analysis of promoter sequences from Escherichia coli. *J Mol Biol*, 186:117–128, Nov 1985. ISSN 0022-2836.

D. Garfinkel. A machine-independent language for the simulation of complex chemical and biochemical systems* 1. *Computers and Biomedical Research*, 2(1):31–44, 1968. ISSN 0010-4809.

A. Garny, D.P. Nickerson, J. Cooper, R.W. Santos, A.K. Miller, S. McKeever, P.M.F. Nielsen, and P.J. Hunter. CellML and associated tools and techniques. *Philosophical Transactions A*, 366(1878):3017, 2008.

Laurent Gautier, Leslie Cope, Benjamin M. Bolstad, and Rafael A. Irizarry. affy–analysis of Affymetrix GeneChip data at the probe level. *Bioinformatics*, 20:307–315, Feb 2004. ISSN 1367-4803. doi: 10.1093/bioinformatics/btg405.

C. Gear. Simultaneous numerical solution of differential-algebraic equations. *IEEE transactions on circuit theory*, 18(1):89–95, 1971. ISSN 0018-9324.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java (TM) Language Specification, The (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

Timothy J. Griffin, Steven P. Gygi, Trey Ideker, Beate Rist, Jimmy Eng, Leroy Hood, and Ruedi Aebersold. Complementary profiling of gene expression at the transcriptome and proteome levels in Saccharomyces cerevisiae. *Mol Cell Proteomics*, 1:323–333, Apr 2002. ISSN 1535-9476.

F.S. Grodins, J. Buell, and A.J. Bart. Mathematical analysis and digital simulation of the respiratory control system. *Journal of Applied Physiology*, 22(2):260, 1967. ISSN 8750-7587.

W. Gropp, R. Thakur, and E. Lusk. *Using MPI-2: Advanced features of the message passing interface*. MIT Press Cambridge, MA, USA, 1999.

D. L. Gumucio, H. Heilstedt-Williamson, T. A. Gray, S. A. Tarlé, D. A. Shelton, D. A. Tagle, J. L. Slightom, M. Goodman, and F. S. Collins. Phylogenetic footprinting reveals a nuclear protein which binds to silencer sequences in the human gamma and epsilon globin genes. *Mol Cell Biol*, 12:4919–4929, Nov 1992. ISSN 0270-7306.

H. Guo, N.T. Ingolia, J.S. Weissman, and D.P. Bartel. Mammalian microRNAs predominantly act to decrease target mRNA levels. *Nature*, 466(7308):835–840, 2010. ISSN 0028-0836.

C. Guziolowski, J. Gruel, O. Radulescu, and A. Siegel. Curating a large-scale regulatory network by evaluating its consistency with expression datasets. *Computational Intelligence Methods for Bioinformatics and Biostatistics*, pages 144–155, 2009a.

Carito Guziolowski, Annabel Bourde, Francois Moreews, and Anne Siegel. Bioquali cytoscape plugin: analysing the global consistency of regulatory networks. *BMC Genomics*, 10(1):244, 2009b. ISSN 1471-2164. doi: 10.1186/1471-2164-10-244. URL `http://www.biomedcentral.com/1471-2164/10/244`.

C T Harbison, D B Gordon, T I Lee, N J Rinaldi, K D Macisaac, T W Danford, N M Hannett, J B Tagne, D B Reynolds, J Yoo, E G Jennings, J Zeitlinger, D K Pokholok, M Kellis, P A Rolfe, K T Takusagawa, E S Lander, D K Gifford, E Fraenkel, and R A Young. Transcriptional regulatory code of a eukaryotic

genome. *Nature*, 431(7004):99–104, Sep 2004. doi: 10.1038/nature02800. URL `http://www.hubmed.org/display.cgi?uids=15343339`.

B. Harr and C. Schlotterer. Comparison of algorithms for the analysis of Affymetrix microarray data as evaluated by co-expression of genes in known operons. *Nucleic Acids Research*, 34(2):e8, 2006.

W.J. Hedley, M.R. Nelson, DP Bellivant, and P.F. Nielsen. A short introduction to CellML. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 359(1783):1073, 2001. ISSN 1364-503X.

P. Hegde, R. Qi, K. Abernathy, C. Gay, S. Dharap, R. Gaspard, J. E. Hughes, E. Snesrud, N. Lee, and J. Quackenbush. A concise guide to cDNA microarray analysis. *Biotechniques*, 29:548–50, 552–4, 556 passim, Sep 2000. ISSN 0736-6205.

G. Z. Hertz and G. D. Stormo. Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. *Bioinformatics*, 15: 563–577, Jul/Aug 1999. ISSN 1367-4803.

A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, and C.S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31 (3):363–396, 2005. ISSN 0098-3500.

A.L. Hodgkin and A.F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500, 1952. ISSN 0022-3751.

S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979.

Z. Hu, P.J. Killion, and V.R. Iyer. Genetic reconstruction of a functional transcriptional regulatory network. *Nature genetics*, 39(5):683–687, 2007. ISSN 1061-4036.

M. Hucka, A. Finney, H. Sauro, H. Bolouri, J. Doyle, and H. Kitano. The ERATO Systems Biology Workbench: An integrated environment for multiscale

and multitheoretic simulations in systems biology. *Foundations of Systems Biology*, pages 125–143, 2001.

M. Hucka, A. Finney, HM Sauro, H. Bolouri, JC Doyle, H. Kitano, et al. The systems biology markup language (SBML): A medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524, 2003.

P Hunter and P Nielsen. A strategy for integrative computational physiology. *Physiology (Bethesda)*, 20:316–325, Oct 2005a. doi: 10.1152/physiol.00022.2005. URL `http://www.hubmed.org/display.cgi?uids=16174871`.

P. Hunter and P. Nielsen. A strategy for integrative computational physiology. *Physiology*, 20(5):316–325, 2005b.

T Ideker, T Galitski, and L Hood. A new approach to decoding life: systems biology. *Annu Rev Genomics Hum Genet*, 2:343–372, 2001a. doi: 10.1146/annurev.genom.2.1.343. URL `http://www.hubmed.org/display.cgi?uids=11701654`.

T. Ideker, V. Thorsson, J. A. Ranish, R. Christmas, J. Buhler, J. K. Eng, R. Bumgarner, D. R. Goodlett, R. Aebersold, and L. Hood. Integrated genomic and proteomic analyses of a systematically perturbed metabolic network. *Science*, 292:929–934, May 2001b. ISSN 0036-8075. doi: 10.1126/science.292.5518.929.

S Imoto, T Higuchi, T Goto, K Tashiro, S Kuhara, and S Miyano. Combining microarrays and biological knowledge for estimating gene networks via bayesian networks. *J Bioinform Comput Biol*, 2(1):77–98, Mar 2004. URL `http://www.hubmed.org/display.cgi?uids=15272434`.

Seiya Imoto, Takao Goto, and Satoru Miyano. Estimation of genetic networks and functional structures between genes by using Bayesian networks and nonparametric regression. *Pac Symp Biocomput*, pages 175–186, 2002. ISSN 1793-5091.

Rafael A. Irizarry, Bridget Hobbs, Francois Collin, Yasmin D. Beazer-Barclay, Kristen J. Antonellis, Uwe Scherf, and Terence P. Speed. Exploration, normalization, and summaries of high density oligonucleotide array probe

level data. *Biostatistics*, 4:249–264, Apr 2003. ISSN 1465-4644. doi: 10.1093/biostatistics/4.2.249.

William G. Jacoby. LOESS: A nonparametric, graphical tool for depicting relationships between variables. *Electoral Studies*, 19:577–613, Dec 2000. URL http://www.sciencedirect.com/science/article/B6V9P-40S0D42-7/2/3212e93796349ba4681a97c4b112053c.

Tor-Kristian Jenssen, Astrid Laegreid, Jan Komorowski, and Eivind Hovig. A literature network of human genes for high-throughput analysis of gene expression. *Nat Genet*, 28:21–28, May 2001. ISSN 1061-4036. URL http://dx.doi.org/10.1038/ng0501-21. 10.1038/ng0501-21.

Thorsten Joachims. Transductive inference for text classification using support vector machines. In Ivan Bratko and Saso Dzeroski, editors, *Proceedings of ICML-99, 16th International Conference on Machine Learning*, pages 200–209, Bled, SL, 1999. Morgan Kaufmann Publishers, San Francisco, US. URL citeseer.ist.psu.edu/joachims99transductive.html.

S.L.P. Jones, CV Hall, K. Hammond, WD Partain, PL Wadler, and S. Peyton-Jones. The Glasgow Haskell Compiler: A technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference*, 1993.

S.P. Jones and S.L.P. Jones. *Haskell 98 language and libraries: The revised report*. Cambridge Univ Pr, 2003. ISBN 0521826144.

S.P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Annual Symposium on Principles of Programming Languages: Proceedings of the 23 rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 1996. ISBN 0897917693.

W Just. Computational complexity of multiple sequence alignment with SP-score. *J Comput Biol*, 8(6):615–623, 2001. doi: 10.1089/106652701753307511. URL http://www.hubmed.org/display.cgi?uids=11747615.

R S Kamath and J Ahringer. Genome-wide RNAi screening in Caenorhabditis elegans. *Methods*, 30(4):313–321, Aug 2003. URL http://www.hubmed.org/display.cgi?uids=12828945.

S.P. Karkach and V.I. Osherov. Ab initio analysis of the transition states on the lowest triplet HO potential surface. *The Journal of Chemical Physics*, 110:11918, 1999.

A. E. Kel, E. Gössling, I. Reuter, E. Cheremushkin, O. V. Kel-Margoulis, and E. Wingender. MATCH: A tool for searching transcription factor binding sites in DNA sequences. *Nucleic Acids Res*, 31:3576–3579, Jul 2003. ISSN 1362-4962.

O V Kel, A G Romaschenko, A E Kel, E Wingender, and N A Kolchanov. A compilation of composite regulatory elements affecting gene transcription in vertebrates. *Nucleic Acids Res*, 23(20):4097–4103, Oct 1995. URL `http://www.hubmed.org/display.cgi?uids=7479071`.

G. Klyne and J.J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. 2006.

A. Koenig. The C++ Language Standard. Report ISO/IEC 14882: 1998.

W. Kutta. *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. BG Teubner, 1901.

H. Lähdesmäki, A.G. Rust, and I. Shmulevich. Probabilistic inference of transcription factor binding from multiple data sources. *PLoS One*, 3(3), 2008.

C. Laibe and N. Le Novère. MIRIAM Resources: Tools to generate and resolve robust cross-references in Systems Biology. *BMC Systems Biology*, 1(1):58, 2007. ISSN 1752-0509.

N. C. Lau, L. P. Lim, E. G. Weinstein, and D. P. Bartel. An abundant class of tiny RNAs with probable regulatory roles in Caenorhabditis elegans. *Science*, 294:858–862, Oct 2001. ISSN 0036-8075. doi: 10.1126/science.1065062.

C E Lawrence and A A Reilly. An expectation maximization (EM) algorithm for the identification and characterization of common sites in unaligned biopolymer sequences. *Proteins*, 7(1):41–51, 1990. doi: 10.1002/prot.340070105. URL `http://www.hubmed.org/display.cgi?uids=2184437`.

C E Lawrence, S F Altschul, M S Boguski, J S Liu, A F Neuwald, and J C Wootton. Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment. *Science*, 262(5131):208–214, Oct 1993. URL `http://www.hubmed.org/display.cgi?uids=8211139`.

N. Le Novère. Model storage, exchange and integration. *BMC neuroscience*, 7 (Suppl 1):S11, 2006. ISSN 1471-2202.

N. Le Novère, A. Finney, M. Hucka, U.S. Bhalla, F. Campagne, J. Collado-Vides, E.J. Crampin, M. Halstead, E. Klipp, P. Mendes, et al. Minimum information requested in the annotation of biochemical models (MIRIAM). *Nature biotechnology*, 23(12):1509–1515, 2005. ISSN 1087-0156.

N. Le Novere, B. Bornstein, A. Broicher, M. Courtot, M. Donizelli, H. Dharuri, L. Li, H. Sauro, M. Schilstra, B. Shapiro, et al. BioModels Database: A free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems. *Nucleic acids research*, 34(suppl 1):D689, 2006. ISSN 0305-1048.

K. Levenberg. A method for the solution of certain nonlinear problems in least squares. *Quart. Appl. Math*, 2(2):164–168, 1944.

C. Li and W.H. Wong. Model-based analysis of oligonucleotide arrays: Model validation, design issues and standard error application. *Genome Biol*, 2(8): 0032–1, 2001.

Z. Li, E. Butterworth, and J. Bassingthwaighte. Building an infrastructure for the physiome project. *Annals of Biomedical Engineering*, 28, 2000. ISSN 0090-6964.

J S Liu, A F Neuwald, and C E Lawrence. Bayesian models for multiple local sequence alignment and gibbs sampling strategies. *J Am Stat Assoc*, 90(432): 1156–1170, Dec 1995.

L A Liu and J S Bader. Ab initio prediction of transcription factor binding sites. *Pac Symp Biocomput*, pages 484–495, 2007. URL `http://www.hubmed.org/display.cgi?uids=17990512`.

X Liu, D L Brutlag, and J S Liu. Bioprospector: discovering conserved dna motifs in upstream regulatory regions of co-expressed genes. *Pac Symp Biocomput*, pages 127–138, 2001. URL `http://www.hubmed.org/display.cgi?uids=11262934`.

C.M. Lloyd, M.D.B. Halstead, and P.F. Nielsen. CellML: Its future, present and past. *Progress in Biophysics and Molecular Biology*, 85(2-3):433–450, 2004. ISSN 0079-6107.

C.M. Lloyd, J.R. Lawson, P.J. Hunter, and P.F. Nielsen. The CellML model repository. *Bioinformatics*, 24(18):2122, 2008.

P Lu, C Vogel, R Wang, X Yao, and E M Marcotte. Absolute protein expression profiling estimates the relative contributions of transcriptional and translational regulation. *Nat Biotechnol*, 25(1):117–124, Jan 2007. doi: 10.1038/nbt1270. URL `http://www.hubmed.org/display.cgi?uids=17187058`.

M. Lutz. *Programming python*. O'Reilly Media, Inc., 2006.

S. Mahony, P.E. Auron, and P.V. Benos. DNA familial binding profiles made easy: Comparison of various motif alignment and clustering strategies. *PLoS Comput Biol*, 3(3):e61, 2007.

WG Mallard, F. Westley, JT Herron, RF Hampson, and DH Frizzell. NIST Chemical Kinetics Database. 1998.

A. Mallavarapu, M. Thomson, B. Ullian, and J. Gunawardena. Programming with models: Modularity and abstraction provide powerful capabilities for systems biology. *Journal of The Royal Society Interface*, 6(32):257, 2009. ISSN 1742-5689.

A A Margolin, I Nemenman, K Basso, C Wiggins, G Stolovitzky, R Dalla Favera, and A Califano. Aracne: an algorithm for the reconstruction of gene regulatory networks in a mammalian cellular context. *BMC Bioinformatics*, 7 Suppl 1, 2006. doi: 10.1186/1471-2105-7-S1-S7. URL `http://www.hubmed.org/display.cgi?uids=16723010`.

M. Margulies, M. Egholm, W.E. Altman, S. Attiya, J.S. Bader, L.A. Bemben, J. Berka, M.S. Braverman, Y.J. Chen, Z. Chen, et al. Genome sequencing in

microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, 2005. ISSN 0028-0836.

V.D. Marinescu, I.S. Kohane, and A. Riva. MAPPER: A search engine for the computational identification of putative transcription factor binding sites in multiple genomes. *BMC Bioinformatics*, 6(1):79, 2005.

D.W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11 (2):431–441, 1963. ISSN 0368-4245.

E.E. Matos, F. Campos, R. Braga, and D. Palazzi. CelOWS: An ontology based framework for the provision of semantic web services related to biological models. *Journal of biomedical informatics*, 43(1):125–136, 2010. ISSN 1532-0464.

J.S. Mattick and I.V. Makunin. Non-coding RNA. *Human molecular genetics*, 15(suppl 1):R17, 2006. ISSN 0964-6906.

S.E. Mattsson, H. Elmqvist, and M. Otter. Physical system modeling with modelica. *Control Engineering Practice*, 6(4):501–510, 1998.

V Matys, O V Kel-Margoulis, E Fricke, I Liebich, S Land, A Barre-Dirrie, I Reuter, D Chekmenev, M Krull, K Hornischer, N Voss, P Stegmaier, B Lewicki-Potapov, H Saxel, A E Kel, and E Wingender. TRANSFAC and its module TRANSCompel: Transcriptional gene regulation in eukaryotes. *Nucleic Acids Res*, 34(Database issue):108–110, Jan 2006. doi: 10.1093/nar/gkj143. URL `http://www.hubmed.org/display.cgi?uids=16381825`.

D.L. McGuinness, F. Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 2004.

Pedro Mendes, Wei Sha, and Keying Ye. Artificial gene networks for objective comparison of analysis algorithms. *Bioinformatics*, 19(suppl 2):ii122–129, 2003. doi: 10.1093/bioinformatics/btg1069.

A.K. Miller, J. Marsh, A. Reeve, A. Garny, R. Britten, M. Halstead, J. Cooper, D.P. Nickerson, and P.F. Nielsen. An overview of the CellML API and its implementation. *BMC bioinformatics*, 11(1):178, 2010a. ISSN 1471-2105.

A.K. Miller, T. Yu, R. Britten, M.T. Cooling, J. Lawson, D. Cowan, A. Garny, M.D.B. Halstead, P.J. Hunter, D.P. Nickerson, et al. Revision history aware repositories of computational models of biological systems. *BMC bioinformatics*, 12(1):22, 2011. ISSN 1471-2105.

Andrew K. Miller, Cristin G. Print, Poul M. F. Nielsen, and Edmund J. Crampin. A bayesian search for transcriptional motifs. *PLoS ONE*, 5(11):e13897, 11 2010b. doi: 10.1371/journal.pone.0013897. URL `http://dx.doi.org/10.1371%2Fjournal.pone.0013897`.

R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978. ISSN 0022-0000.

J N Miner and K R Yamamoto. Regulatory crosstalk at composite response elements. *Trends Biochem Sci*, 16(11):423–426, Nov 1991. URL `http://www.hubmed.org/display.cgi?uids=1776172`.

P.J. Mohr, B.N. Taylor, and D.B. Newell. CODATA recommended values of the fundamental physical constants: 2006. *Reviews of modern physics*, 80(2):633–730, 2008. ISSN 1539-0756.

I.I. Moraru, J.C. Schaff, B.M. Slepchenko, ML Blinov, F. Morgan, A. Lakshminarayana, F. Gao, Y. Li, and L.M. Loew. Virtual cell modelling and simulation software environment. *Systems Biology, IET*, 2(5):352–362, 2008.

A V Morozov, J J Havranek, D Baker, and E D Siggia. Protein-DNA binding specificity predictions with structural models. *Nucleic Acids Res*, 33(18):5781–5798, 2005. doi: 10.1093/nar/gki875. URL `http://www.hubmed.org/display.cgi?uids=16246914`.

A. Mortazavi, B.A. Williams, K. McCue, L. Schaeffer, and B. Wold. Mapping and quantifying mammalian transcriptomes by RNA-Seq. *Nature methods*, 5(7):621–628, 2008.

A F Neuwald, J S Liu, and C E Lawrence. Gibbs motif sampling: detection of bacterial outer membrane protein repeats. *Protein Sci*, 4(8):1618–1632, Aug 1995. URL `http://www.hubmed.org/display.cgi?uids=8520488`.

D. Nickerson and M. Buist. Practical application of CellML 1.1: The integration of new mechanisms into a human ventricular myocyte model. *Progress in Biophysics and Molecular Biology*, 98(1):38–51, 2008.

K Oda and H Kitano. A comprehensive map of the toll-like receptor signaling network. *Mol Syst Biol*, 2:2006–2006, 2006. doi: 10.1038/msb4100057. URL `http://www.hubmed.org/display.cgi?uids=16738560`.

F. W. J. Olver, D. W. Lozier, R. F. Boisvert, and C. W. Clark. *NIST Handbook of Mathematical Functions - 5.12. Beta Function.* Cambridge University Press, 2010. ISBN 9780521192250. URL `http://dlmf.nist.gov/5.12`.

E. Osuna, R. Freund, and F. Girosi. An improved training algorithm for support vector machines. *Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop*, pages 276–285, 24-26 Sep 1997. doi: 10.1109/NNSP.1997.622408.

S.V. Pandit, R.B. Clark, W.R. Giles, and S.S. Demir. A mathematical model of action potential heterogeneity in adult rat left ventricular myocytes. *Biophysical Journal*, 81(6):3029–3051, 2001.

D.A. Patterson and D.R. Ditzel. The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, 8(6):33, 1980. ISSN 0163-5964.

G. Pavesi, G. Mauri, and G. Pesole. An algorithm for finding signals of unknown length in DNA sequences. *Bioinformatics*, 17 Suppl 1:S207–S214, 2001. ISSN 1367-4803.

J. Pearl and T. Verma. A theory of inferred causation, 1991. URL `citeseer.comp.nus.edu.sg/125484.html`.

D. Pe'er, A. Regev, G. Elidan, and N. Friedman. Inferring subnetworks from perturbed expression profiles. *Bioinformatics*, 17 Suppl 1:S215–S224, 2001. ISSN 1367-4803.

LR Petzold. Description of DASSL: A differential/algebraic system solver. Technical report, Sandia National Labs., Livermore, CA (USA), 1982.

S. Peyton-Jones, S. Marlow, et al. Glasgow Haskell Compiler. URL `http://www.haskell.org/ghc/`.

J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines, 1998. URL `citeseer.ist.psu.edu/platt98sequential.html`.

K.R. Popper, J. Freed, and L. Freed. The logic of scientific discovery.[Transl. from the German by the author, with the assistance of J. Freed and L. Freed]., 1959.

Jiang Qian, Jimmy Lin, Nicholas M. Luscombe, Haiyuan Yu, and Mark Gerstein. Prediction of regulatory networks: Genome-wide identification of transcription factor targets from gene expression data. *Bioinformatics*, 19(15):1917–1926, Oct 2003. ISSN 1367-4803.

Li-Xuan Qin and Kathleen F. Kerr. Empirical evaluation of data transformations and ranking statistics for microarray analysis. *Nucleic Acids Res*, 32:5471–5479, Oct 2004. ISSN 1362-4962. doi: 10.1093/nar/gkh866.

J. Quackenbush. Computational analysis of microarray data. *Nat Rev Genet*, 2 (6):418–427, Jun 2001. ISSN 1471-0056. doi: 10.1038/35076576.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

Stephen Ramsey, David Orrell, and Hamid Bolouri. Dizzy: Stochastic simulation of large-scale genetic regulatory networks. *J Bioinform Comput Biol*, 3:415–436, Apr 2005. ISSN 0219-7200.

R.H. Rand. *Computer algebra in applied mathematics: An introduction to MACSYMA*. Pitman Boston, 1984. ISBN 0273086324.

J.J. Rice, F. Wang, D.M. Bers, and P.P. De Tombe. Approximate model of cooperative activation and crossbridge cycling in cardiac muscle using ordinary differential equations. *Biophysical Journal*, 95(5):2368–2390, 2008. ISSN 0006-3495.

Matthew E. Ritchie, Jeremy Silver, Alicia Oshlack, Melissa Holmes, Dileepa Diyagama, Andrew Holloway, and Gordon K. Smyth. A comparison of background correction methods for two-colour microarrays. *Bioinformatics*, 23:2700–2707, Oct 2007. ISSN 1460-2059. doi: 10.1093/bioinformatics/btm412.

G.C. Roman and D. Garfinkel. BIOSSIM–a structured machine-independent biological simulation language. *Computers and Biomedical Research*, 11(1): 3–15, 1978. ISSN 0010-4809.

F P Roth, J D Hughes, P W Estep, and G M Church. Finding DNA regulatory motifs within unaligned noncoding sequences clustered by whole-genome mRNA quantitation. *Nat Biotechnol*, 16(10):939–945, Oct 1998. doi: 10.1038/nbt1098-939. URL `http://www.hubmed.org/display.cgi?uids=9788350`.

C. Runge. Uber die numerische Aufl
"osung totaler Differetialgleichungen. *Mathematische Annalen*, 46:167–178, 1895.

Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3): 856–869, 1986.

M Schena, D Shalon, R W Davis, and P O Brown. Quantitative monitoring of gene expression patterns with a complementary DNA microarray. *Science*, 270(5235):467–470, Oct 1995. URL `http://www.hubmed.org/display.cgi?uids=7569999`.

T Schlitt and A Brazma. Current approaches to gene regulatory network modelling. *BMC Bioinformatics*, 8 Suppl 6, 2007. doi: 10.1186/1471-2105-8-S6-S9. URL `http://www.hubmed.org/display.cgi?uids=17903290`.

Scholkopf, Smola, Williamson, and Bartlett. New support vector algorithms. *Neural Comput*, 12:1207–1245, May 2000. ISSN 0899-7667.

B. Scholkopf, P. Bartlett, A. Smola, and R. Williamson. Support vector regression with automatic accuracy control, 1998. URL `citeseer.ist.psu.edu/article/scholkopf98support.html`.

Peter Schuster. A beginning of the end of the holism versus reductionism debate?: Molecular biology goes cellular and organismic: The Simply Complex. *Complex.*, 13(1):10–13, 2007. ISSN 1076-2787. doi: http://dx.doi.org/10.1002/cplx.v13:1.

Shai S. Shen-Orr, Ron Milo, Shmoolik Mangan, and Uri Alon. Network motifs in the transcriptional regulation network of Escherichia coli. *Nat Genet*, 31: 64–68, May 2002. ISSN 1061-4036. doi: 10.1038/ng881.

A. Siegel, C. Guziolowski, P. Veber, O. Radulescu, and M. Le Borgne. Qualitative response of interaction networks: application to the validation of biological models. *PAMM*, 7(1):1121803–1121804, 2007.

J. Siegel. OMG overview: CORBA and the OMA in enterprise computing, 1998.

Saurabh Sinha and Martin Tompa. Discovery of novel transcription factor binding sites by statistical overrepresentation. *Nucleic Acids Res*, 30:5549–5560, Dec 2002. ISSN 1362-4962.

Saurabh Sinha and Martin Tompa. YMF: A program for discovery of novel transcription factor binding sites by statistical overrepresentation. *Nucleic Acids Res*, 31:3586–3588, Jul 2003. ISSN 1362-4962.

T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981. URL `citeseer.ist.psu.edu/smith81identification.html`.

P. T. Spellman, G. Sherlock, M. Q. Zhang, V. R. Iyer, K. Anders, M. B. Eisen, P. O. Brown, D. Botstein, and B. Futcher. Comprehensive identification of cell cycle-regulated genes of the yeast Saccharomyces cerevisiae by microarray hybridization. *Mol Biol Cell*, 9:3273–3297, Dec 1998. ISSN 1059-1524.

B. Stearn. XULRunner: A New Approach for Developing Rich Internet Applications. *IEEE Internet Computing*, pages 67–73, 2007.

Yoshinori Tamada, SunYong Kim, Hideo Bannai, Seiya Imoto, Kousuke Tashiro, Satoru Kuhara, and Satoru Miyano. Estimating gene networks from gene

expression data by combining Bayesian network model with promoter element detection. *Bioinformatics*, 19 Suppl 2:ii227–ii236, Oct 2003. ISSN 1460-2059.

R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.*, 22:215–225, 1975.

S. Tavazoie, J. D. Hughes, M. J. Campbell, R. J. Cho, and G. M. Church. Systematic determination of genetic network architecture. *Nat Genet*, 22: 281–285, Jul 1999. ISSN 1061-4036. doi: 10.1038/10343.

B.N. Taylor and A. Thompson. *The international system of units (SI)*. International Bureau of Weights and Measures, 2001.

M C Teixeira, P Monteiro, P Jain, S Tenreiro, A R Fernandes, N P Mira, M Alenquer, A T Freitas, A L Oliveira, and I Sá-Correia. The YEASTRACT database: a tool for the analysis of transcription regulatory associations in Saccharomyces cerevisiae. *Nucleic Acids Res*, 34(Database issue):446–451, Jan 2006a. doi: 10.1093/nar/gkj013. URL `http://www.hubmed.org/display.cgi?uids=16381908`.

M.C. Teixeira, P. Monteiro, P. Jain, S. Tenreiro, A.R. Fernandes, N.P. Mira, M. Alenquer, A.T. Freitas, A.L. Oliveira, and I. Sá-Correia. The YEASTRACT database: A tool for the analysis of transcription regulatory associations in Saccharomyces cerevisiae. *Nucleic acids research*, 34(suppl 1):D446, 2006b. ISSN 0305-1048.

W Thompson, E C Rouchka, and C E Lawrence. Gibbs recursive sampler: finding transcription factor binding sites. *Nucleic Acids Res*, 31(13):3580–3585, Jul 2003. URL `http://www.hubmed.org/display.cgi?uids=12824370`.

M. Tompa, N. Li, T.L. Bailey, G.M. Church, B. De Moor, E. Eskin, A.V. Favorov, M.C. Frith, Y. Fu, W.J. Kent, et al. Assessing computational tools for the discovery of transcription factor binding sites. *Nature Biotechnology*, 23(1):137–144, 2005.

Amy Hin Yan Tong, Marie Evangelista, Ainslie B. Parsons, Hong Xu, Gary D. Bader, Nicholas Page, Mark Robinson, Sasan Raghibizadeh, Christopher W. V. Hogue, Howard Bussey, Brenda Andrews, Mike Tyers, and Charles Boone.

Systematic Genetic Analysis with Ordered Arrays of Yeast Deletion Mutants. *Science*, 294(5550):2364–2368, 2001. doi: 10.1126/science.1065810. URL `http://www.sciencemag.org/cgi/content/abstract/294/5550/2364`.

K. Tran, N.P. Smith, D.S. Loiselle, and E.J. Crampin. A metabolite-sensitive, thermodynamically constrained model of cardiac cross-bridge cycling: Implications for force development during ischemia. *Biophys J*, 98:267–276, 2010.

W. Tsang and RF Hampson. Chemical kinetic data base for combustion chemistry. Part I. Methane and related compounds. *J. Phys. Chem. Ref. Data*, 15(3):1087, 1986.

A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230, 1937. ISSN 0024-6115.

L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1968.1972.

J. van Helden, B. André, and J. Collado-Vides. Extracting regulatory sites from the upstream region of yeast genes by computational analysis of oligonucleotide frequencies. *J Mol Biol*, 281:827–842, Sep 1998. ISSN 0022-2836. doi: 10.1006/jmbi.1998.1947.

J. van Helden, A. F. Rios, and J. Collado-Vides. Discovering regulatory elements in non-coding sequences by analysis of spaced dyads. *Nucleic Acids Res*, 28: 1808–1818, Apr 2000. ISSN 1362-4962.

V Vapnik. *Estimation of Dependences Based on Empirical Data [in Russian]*. Nauka, Moscow, 1979. (English translation: Springer Verlag, New York, 1982).

V Vapnik and A Lerner. Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24, 1963.

Vladimir Vapnik. *Estimation of Dependences Based on Empirical Data: Springer Series in Statistics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387907335.

Vladimir N. Vapnik. *The Nature of Statistical Learning Theory (Information Science and Statistics)*. Springer, November 1999. ISBN 0387987800. URL `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0387987800`.

VN Vapnik and A Chervonenkis. About structural risk minimization principle. *Automation and Remote Control*, 1974.

G J Veenstra and A P Wolffe. Gene-selective developmental roles of general transcription factors. *Trends Biochem Sci*, 26(11):665–671, Nov 2001. URL `http://www.hubmed.org/display.cgi?uids=11701325`.

D. Veillard. Libxml2: The XML C parser and toolkit of Gnome.

V.E. Velculescu, L. Zhang, B. Vogelstein, and K.W. Kinzler. Serial analysis of gene expression. *Science*, 270(5235):484, 1995. ISSN 0036-8075.

J.C. Venter, M.D. Adams, E.W. Myers, P.W. Li, R.J. Mural, G.G. Sutton, H.O. Smith, M. Yandell, C.A. Evans, R.A. Holt, et al. The sequence of the human genome. *science*, 291(5507):1304, 2001. ISSN 0036-8075.

V. Volterra. Variations and fluctuations of the number of individuals in animal species living together. *ICES Journal of Marine Science*, 3(1):3, 1928.

A. Wagner. How to reconstruct a large genetic network from n gene perturbations in fewer than n(2) easy steps. *Bioinformatics*, 17:1183–1197, Dec 2001. ISSN 1367-4803.

D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, Jun 1998. ISSN 0028-0836. doi: 10.1038/30918.

J. Weidendorfer. Sequential Performance analysis with Callgrind and KCachegrind. In *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, page 93. Springer, 2008.

X. Wen, S. Fuhrman, G. S. Michaels, D. B. Carr, S. Smith, J. L. Barker, and R. Somogyi. Large-scale temporal gene expression mapping of central nervous system development. *Proc Natl Acad Sci U S A*, 95:334–339, Jan 1998. ISSN 0027-8424.

S. Wolfram. *Mathematica: A system for doing mathematics by computer*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1988. ISBN 0201193302.

L. Wood. Programming the Web: The W3C DOM specification. *IEEE Internet Computing*, 3(1):48–54, 1999.

Z. Wu, R.A. Irizarry, R. Gentleman, F. Martinez-Murillo, and F. Spencer. A Model-Based Background Adjustment for Oligonucleotide Expression Arrays. *Journal of the American Statistical Association*, 99(468):909–918, 2004.

Y. Zhao and J. Sun. Robust support vector regression in the primal. *Neural Networks*, 21(10):1548–1555, 2008. ISSN 0893-6080.

J. Zhu and M. Q. Zhang. SCPD: A promoter database of the yeast Saccharomyces cerevisiae. *Bioinformatics*, 15(7-8):607–611, Jul/Aug 1999. ISSN 1367-4803.

Appendix A

Introduction to selected aspects of Haskell

This section provides a brief overview of the programming language Haskell, to assist readers who are unfamiliar with the language to understand the remainder of this chapter, and the following chapter. Most of the material in this section is formally specified in Jones and Jones [2003].

Haskell is a pure functional programming language, meaning that program variables cannot be changed; instead all data structures are immutable.

It is also a lazy programming language, which, as noted before, means that by default, expressions are only evaluated when their value is required.

Furthermore, it is strictly typed, meaning that every expression has a type that is known at compile time. However, Haskell makes use of Hindley-Milner [Milner, 1978, Damas and Milner, 1982] type inference, and so type declarations are only required where it would be ambiguous not to provide them. However, providing types declarations more often than is strictly required can help both to document code and to isolate bugs in source code (otherwise errors that result in the wrong type could propagate, making it hard to understand why the types in a program do not pass type checking).

A.1 Types and constructors

In Haskell, every expression can ultimately be resolved to a value. Every value (apart from functions and built-in values) has an associated constructor. A constructor gives a value its type, and has zero or more arguments. The number

of arguments to a given constructor is always the same.

There can be more than one constructor for a given type, but all the constructors for a type are declared in one place (and so new constructors for the same type cannot be declared). Types and their associated constructors can be declared using the following syntax:

```
data TypeName = Constructor1Name c1arg1 c1arg2 |
                Constructor2Name c2arg1 c2arg2 c2arg3 | ... |
                ConstructorNName cnarg1 cnarg2 ... cnargm
```

In the above listing, TypeName is a type name; type names always start with a capital letter, and can be made up of letters, digits, underscores, and single quotes. Constructor1Name, Constructor2Name, and so on through to ConstructorNName are arbitrary constructor names. Constructor names follow the same rules as for type names. The strings c1arg1 and so on describe the type of the arguments to the constructors.

Type names and constructor names exist in a different namespace, so it is valid (and common) for the type name and constructor to be the same. For example:

```
data Something = Something Int Int
```

In this case, Something is both a type and a constructor that produces a value of type Something from two integers.

Data declarations are used in Haskell both to structure information and to enumerate possible choices. For example:

```
data SomeStructuredInformation = SomeStructuredInformation String Int
data Colour = Red | Green | Blue | RGB Double Double Double
```

In this case, a value of type SomeStructuredInformation must be made up of a String followed by an Int; a value of type Colour can the Red, Green or Blue constructor, or can be an RGB value consisting of three double precision floating point values.

In addition, constructors of two arguments can use constructor symbols for in-order constructors:

```
data SpecialPair = Int :@@ Int
```

In this case, :@@ is the constructor symbol, and the arguments are of type Int; constructor symbols can be any string of punctuation characters (excluding the single quote) that starts with the colon (:) character.

Constructors require that some information be stored around each value. In cases where there is exactly one argument, this is wasteful, and so a separate syntactic construct, newtype is provided:

```
newtype TypeName = ConstructorName argumentType
```

The semantics are effectively the same as if data had been written instead of newtype, and the same type checking behaviour occurs by the compiler. However, the resulting compiled program will directly refer to an argumentType value rather than creating an extra wrapper around it. Newtype therefore is useful for enforcing code correctness, but does not impact on the peformance of a compiled program. For example:

```
newtype Height = Height Int
newtype Width = Width Int
```

In this case, specific types of Height and Width are declared, and could be used as arguments to a function that takes a Height and a Width (functions are discussed later). The behaviour of the program at run time will be no different to if the function was written to take two Ints instead of a Height and a Width, but at compile time, the compiler will check that the correct type is being passed to the function and give a type checking error if, say, the Height argument and the Width argument were accidentally swapped when calling the function.

In addition, Haskell allows for types to be aliased:

```
type SomeAliasType = Int
```

In this case, SomeAliasType is interchangeable with Int.

A.2 Type variables

Haskell allows type variables. Type variables appear after type names, and are substituted into a type to make the final type. For example:

```
data NamedSomething a = NamedSomething String a
```

In this string, a is the type variable. Type variable names follow the same rules as for type names, except that they start with a lower case letter. By convention, they are often a single letter. The first occurrence of a, on the left hand side of the equals sign declares that a is the first (and in this case, only) type variable. The second occurrence in this example makes the second argument of NamedSomething have type a. However, a is not a rigid type. Later, a can be substituted for any other type; for example, NamedSomething Int is a type (and in this type, the value of a is Int); the constructor for NamedSomething Int takes two parameters, a String and an Int.

Data declarations can give names to members; for example:

```
data Gene = Gene { length :: Int, sequence :: String }
```

Data, newtype, and type alias declarations can all define types that take type variable arguments.

A.3 Functions

A function is a mapping from a value argument to a value. In Haskell, a function is a type of value; anything that can be done with other values can be done with functions.

The type of a function from a value of type A to a value of type B is:

```
A -> B
```

Because a function is a value, a function can map from a value to a function. For example, for types a, b, and c, the following is a possible type for a function:

A -> (B -> C)

The above type is more commonly written without the brackets:

A -> B -> C

While this is the type of a function which maps from A to a function from B to C, it can be thought of as the type of a function of two arguments, A and B, to C.

Arguments can be applied to a function by giving the argument after an expression for a function, which causes the mapping to be performed. For example, if f is a variable (described below) containing a function of type A -> B -> C, and x is a variable of type A, then f x has type B -> C. This is in turn an expression, so if y is a variable of type B, then f x y has type C.

Constructors, presented already, are a specific type of function, and the arguments to the constructor are the types listed after the name of the constructor, mapping to the declared type. For example, following the declaration:

data A = B **String Int**

the constructor B is a function of type **String** -> **Int** -> A.

Likewise, named constructor arguments also have a corresponding function. After the following:

data Gene = Gene { **length** :: **Int**, **sequence** :: **String** }

the follow functions become available automatically: Gene, of type **Int** -> **String** -> Gene, length, of type Gene -> **Int** (for accessing the length from a gene), and sequence, of type Gene -> **String**.

A.4 Lambdas

A lambda is an anonymous function definition. A lambda expression uses the following syntax:

```
\x -> B x
```

Assuming B is defined as above, in this syntax, x is a local bound variable within the scope of the lambda expression. The above expression has type **String** −> **Int** −> A (the same as the constructor B, because constructing a lambda of one argument is the exact inverse of applying that argument to a function).

To produce a function that swaps the order of arguments to B, it would be possible to write:

```
\x -> \y -> B y x
```

In this expression, the value of the first lambda is a second lambda; the overall type is **Int** −> **String** −> A

However, syntactic sugar is provided to simplify lambdas for multiple arguments, and so instead it is possible to write:

```
\x y -> B y x
```

Functions are values, and so functions can operate on functions. For example, consider the following lambda:

```
\f x y -> f y x
```

The type of this expression is (a −> b −> c) −> b −> a −> c, where a, b, and c are examples of type variables. In other words, it is a function which can be applied to a function to flip the order of arguments to the function (such a function, called flip, is provided in the standard library).

A.5 Variables, equations and type definitions

A variable is a name with an associated type and value. The type is computed at compile time, and the value is computed lazily when it is required.

Variables have a name, which follow the same rules as for type variable names; types and values are orthogonal, and so variables can be given the same name as a type variable.

Top-level variables are in scope within an source entire file (even on lines prior to the definition of the variable). They are defined using equations:

```
myVariableName = 1 + 1
```

This states that the variable myVariableName has a value computed from an expression (expressions are discussed later) on the right hand side.

Because Haskell is a pure functional programming language, the value of a variable cannot change [1]. Declarations can appear in any order; for example, the following is valid:

```
z = x + y
x = 1
y = 2
```

Variables can optionally have type definitions associated with them; type definitions must appear immediately before the equation (with only whitespace and comments intervening):

```
z :: Int
z = x + y
```

The line `z :: Int` declares that the type of z is Int.

Functions can be stored in variables using the above notation combined with lambdas; for example:

```
f :: Int -> Int -> Int
f = \x y -> x + y
```

However, Haskell provides a more convenient notation:

```
f x y = x + y
```

[1] however, it is possible for a different variable with the same name to be declared in a more specific scope

There can be more than one formula for the same equation; all formulae must appear together, separated only by comments and whitespace. Each equation must match a different pattern of constructors. For example:

```
data Colour = Red | Green | Blue | RGB Double Double Double
getRedComponent :: Double
getRedComponent Red = 1.0
getRedComponent (RGB r _ _) = r
getRedComponent _ = 0.0
```

Patterns are attempted to be matched in the order they appear; once a match is found, no further matches are tried; if Colour is Red, for example, the first equation matches. Constructor arguments can be specified as local variables (r in the case of the second equation). The special symbol _ is called the irrefutable pattern because it matches anything. An exception is raised at run time if no patterns match for a function evaluation; the use of the irrefutable pattern makes this impossible.

Pattern matching is useful even when there is only one variable, to extract information from constructors. A special notation is provided for extracting information from named constructors:

```
data Gene = Gene { geneId :: Int, geneLength :: Int }
data TranscriptionFactor :: { tfId :: Int, tfGene :: Gene }
getTFGeneId :: TranscriptionFactor -> Int
getTFGeneId (TranscriptionFactor { tfGene = Gene { geneId = gid} }) = gid
```

Variables names can be symbols made up of punctuation (but can't start with a colon (:) and can't contain single quotes); the variable can be accessed by surrounding the name in brackets, or using the name as a function in-order. For example:

```
(+++) :: Int -> Int -> Int
x +++ y = x + (y * 2)
(++++) :: Int -> Int -> Int
(++++) x y = x + (y * 2)
a = 1 +++ 2
```

```
b = (+++) 1 2
c = 1 ++++ 2
```

In this case, +++ both do the same thing, they are just defined different ways. Similarly, a, b, and c are all equivalent.

A.6 Scoped variables

Variables can have more narrow scope through the use of the let or where notations. The notations allow type definitions and formulae to appear inside an expression rather than at the top level.

```
x = let
      y :: Int
      y = 1
      z = 2
    in
      y + z
a = b + c
      where
        b = 1
        c :: Int
        c = 2
```

The only major difference between let and where clauses is the order in which variable bindings are presented; the programmer can choose whichever form makes the program easier to understand.

A.7 Lists

In Haskell, the list constructor is called (:), and is defined something like the following:

```
data [] a = a:[a] | []
```

The construct [] is a special value meaning an empty list. The construct [a] means a list of type variable a. A list is either empty, or it is a value of the list type (called the head of the list) followed by a list (called the tail of the list). Values can be strung together until the end of the list, when the list is terminated with the empty list value. Note, however, that because Haskell is a lazy language, infinite lists are commonly used; the tail of a list is not computed until it is required.

Lists can be produced directly using the constructor; for example, a list with 5 items could be produced using the following syntax:

```
myList :: [Int]
myList = 1:2:3:4:5:[]
```

A more convenient syntax for building lists is provided:

```
myList :: [Int]
myList = [1,2,3,4,5]
```

Lists consisting of sequences can be produced as follows; for all integers between 0 and 10:

```
integers0To10 = [0..10]
```

For an infinite list of all non-negative integers, [0..] can be used. A second member can be used to specify the step. For example:

```
multiplesOfPoint1 :: [Double]
multiplesOfPoint1 = [0,0.1..]
```

List comprehensions can be used to build lists out of other lists. For example:

```
data ColourRGB = ColourRGB {
    redComp :: Double,
    greenComp :: Double,
    blueComp :: Double
                              }
allColours = [ ColourRGB { redComp = r,
```

```
                                greenComp = g, blueComp = b} |
        r <- [0,0.01..1], g <- [0,0.01..1],
        b <- [0,0.01..1]]
```

Lists cannot be changed, but parts of them can be reused to create a new list. For example, to add a new item to the front of a list, the old list is used as the tail, and the new item is the head.

A.8 Strings

A string is simply a list of characters; it has type [**Char**]. This type is aliased as String.

A.9 Literals

Literals are special syntax for defining values. Numeric literals (of which Int and Double literals are examples) can be represented by a sequence of digits (possibly with a decimal point). Strings can be represented by a series of characters between double quotes; closing quotes can be escaped with a slash. Character literals can be represented by a character between single quotes.

```
someString = "Test"
someNumber = 10
someChar = 'a'
```

A.10 Tuples

Haskell provides a series of built in constructors and types for tuples. A tuple is a collection of values, each of which can have a different type.

```
(,) :: a -> b -> (a, b)
(,,) :: a -> b -> c -> (a, b, c)
```

and so on. The notation (a, b, ..., c) is used both to represent the type of a tuple of types a, b and c, and a constructor for a tuple of values a, b, and c. An empty tuple () is commonlyy used as a placeholder for a value where no information needs to be stored.

A.11 Typeclasses

A typeclass is a property of one type (or with a popular extension called multiparameter type classes, one or more types).

A type class can be defined with the class keyword, with type parameters for specific types:

```
class MyClass a
```

A declaration that type belongs to a typeclass is called an instance:

```
instance MyClass Int
```

Instance definitions are open world, meaning that instances can always be extended to add new types, rather than needing to be defined up front.

Typeclasses can be used to restrict a type variable, by putting the restriction in what is called the instance head for a type signature:

```
f :: MyClass a => a -> a
```

Type classes can additionally provide functions. These functions must be provided for all instances (however, default functions can be provided). For example, the Num typeclass, for numbers, is defined as:

```
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
```

```
signum :: a -> a
fromInteger :: Integer -> a
```

Each instance then defines each of these functions.

A.12 Prelude

Haskell provides a set of standard functions and typeclasses, called the Prelude.

One widely used Prelude function is $, which acts like the identity function; it is used because it has the highest precedence of any function in Haskell; for example:

```
f x $ g y
```

is an alternative way of writing f x (g x)

A.13 Modules

Each Haskell file is a module; by default, a module is called Main, but it can be given an explicit name:

```
module SomeName
where
```

By default, all functions, types, and constructors are exported. However, specific functions can be listed for export instead:

```
module SomeName (someFunction,
                 SomeType(SomeConstructor1,SomeConstructor2),
                 (++++))
where
```

Modules can be imported into another module. By default, all exported functions, types, and constructors are imported:

```
import SomeName
```

Specific functions can be imported instead:

```
import SomeName (SomeType(SomeConstructor1))
```

Alternatively, specific functions could be excluded. This can be used with Prelude to make room for other functions with the same name (if no explicit import is given, Prelude is imported implicitly):

```
import Prelude hiding (head)
```

Qualified imports can be used to give imports a name:

```
import qualified Data.Map as M
```

Functions and types from Data.Map can then be referred to using the M prefix (for example, as M.Map).

A.14 Case and if

Pattern matching through functions has already been introduced. It is also possible using the case construct:

```
case someExpression
  of
    Pattern1 -> expression1
    Pattern2 -> expression2
    ...
```

The Bool type is simply a data declaration, and so it is possible to write code like:

```
a myData + (case (shouldIncludeB myData)
                 of
                   True -> b myData
                   False -> 0)
```

However, Haskell provides an if statement to make such code more readable:

```
a myData + (if (shouldIncludeB myData)
              then
                b myData
              else 0)
```