Computational Software for Building Biochemical Reaction Network Models with Differential Equations

Nicholas A. Allen

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Clifford A. Shaffer, Chairman
Lenwood S. Heath
Narendran Ramakrishnan
John J. Tyson
Layne T. Watson

November 11, 2005
Blacksburg, Virginia

Keywords: biological modeling, systems biology, modeling process, JigCell, modeling support environment

Computational Software for Building Biochemical Reaction Network Models with Differential Equations

Nicholas A. Allen

Abstract

The cell is a highly ordered and intricate machine within which a wide variety of chemical processes take place. The full scientific understanding of cellular physiology requires accurate mathematical models that depict the temporal dynamics of these chemical processes. Modelers build mathematical models of chemical processes primarily from systems of differential equations. Although developing new biological ideas is more of an art than a science, constructing a mathematical model from a biological idea is largely mechanical and automatable.

This dissertation describes the practices and processes that biological modelers use for modeling and simulation. Computational biologists struggle with existing tools for creating models of complex eukaryotic cells. This dissertation develops new processes for biological modeling that make model creation, verification, validation, and testing less of a struggle. This dissertation introduces computational software that automates parts of the biological modeling process, including model building, transformation, execution, analysis, and evaluation. User and methodological requirements heavily affect the suitability of software for biological modeling. This dissertation examines the modeling software in terms of these requirements.

Intelligent, automated model evaluation shows a tremendous potential to enable the rapid, repeatable, and cost-effective development of accurate models. This dissertation presents a case study that indicates that automated model evaluation can reduce the evaluation time for a budding yeast model from several hours to a few seconds, representing a more than 1000-fold improvement. Although constructing an automated model evaluation procedure requires considerable domain expertise and skill in modeling and simulation, applying an existing automated model evaluation procedure does not. With this automated model evaluation procedure, the computer can then search for and potentially discover models superior to those that the biological modelers developed previously.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A cell is a highly ordered and intricate biochemical machine surrounded by membranes, in which a wide variety of complex chemical processes take place. These chemical processes form a tightly-coupled biochemical reaction network that interacts with both the changing environment and the development that occurs within the cell. Understanding the behavior of these chemical processes is vital for developing many new biological applications.

A biochemical reaction network is more complicated than the sum of its pieces. Just as the dynamical behavior of a large and complex electrical system is not obviously apparent in its wiring diagram, the dynamical behavior of a large and complex regulatory network is not obviously apparent even when biologists know the component pieces, such as the genes and proteins that interact in the chemical processes. Furthermore, experimentalists struggle to make accurate measurements of processes without disturbing the living system. Although cells are collectively robust, an individual cell is a delicate and carefully-balanced chemical system that the experimentalist can easily perturb in uncontrollable ways. Without an effective means of experimental control, an experimentalist cannot readily gather information that advances the state of understanding about the biological system.

From a hypothesis about a biological system, the experimentalist must determine the important measurable quantities in the living cell. Then, the experimentalist must either carefully manipulate the cell to acquire measurements without disturbing the behavior of the system, or design and perform a new experiment that produces equivalent results. Often, the experimentalist must construct an artificial system that duplicates specific properties of the living cell, conduct measurements in the artificial system, and then demonstrate that those observations truly correspond to the behavior inside the cell. Performing biological laboratory experiments remains a laborious and time-consuming exercise despite the development of improved methods, techniques, and tools over the past centuries.

Acquiring high-quality experimental data is a slow and expensive process that cannot keep up with the pace at which biologists produce new hypotheses. Although a biologist can sketch out the idea for a biochemical reaction network in the form of a wiring diagram in a matter of hours, performing laboratory experiments to justify that biochemical reaction network can take months or years for an experimentalist to complete. Moreover, the complexity of the hypotheses is forever increasing. Many hypotheses must go untested because the biologist cannot perform or obtain the corresponding laboratory work. The biological community is in need of practices that can help validate hypotheses without consuming undue time or resources in experimental laboratories. Often, a biologist has a fixed collection of experimental evidence about a biochemical reaction network and must get the most out of that evidence when building a model. This dissertation focuses on maximizing the value of existing experimental evidence.

The latency in testing a hypothesis delays the application of new biological ideas to other scientific fields that are critically dependent upon basic biological research to develop products. One approach for alleviating this bottleneck is the use of quantitative and mathematical modeling of biological systems to test hypotheses without the laborious construction of laboratory experiments. Although many molecular biologists are not accustomed to this computational approach for scientific exploration, a growing minority of biological practitioners has adopted modeling methods and practices. These biological modelers unfortunately must work largely without the aid of robust and specialized modeling tools. In the past several years, many groups have attempted to remedy this lack of tools by creating software supportive of computational biology, biological modeling, and systems biology efforts.

1.1 Modeling and Simulation

A model is an artificial construct that reproduces the particularly desired behaviors and properties of a natural system. Although it is possible to build models with different forms, such as physical, logical, or mathematical models, this dissertation focuses on mathematical models of biological systems. Modelers attempt to capture the salient properties of their biological system with mathematical equations. After performing experiments on the model, a modeler then relates the results of the model experiment to an equivalent experiment in the biological system.

Simulation is a method for studying the evolution of model behaviors over time. The term 'simulation' implies that the model captures the studied system imperfectly, perhaps by omitting some details. When the modeler attempts to perfectly replicate a natural system, the preferred term for these activities is 'emulation'. The modelers that this dissertation describes do not attempt to perfectly replicate the cell and so use simulation as their primary means of studying a model. Moreover, it is doubtful that modelers could conceivably replicate the complex behaviors of a cell without error given the currently available computational resources.

Although simulation has a heavily developed theory within the field of computer science, the practice of modeling and simulation is still more of an art than a science. This dissertation explains how biological modelers perform certain modeling and simulation activities. Modelers often learn about modeling and simulation through the experience of building models rather than through formal learning. Many modeling and simulation experts have published guides based on specific past experiences. However, few handbooks and instructional materials explain how to perform modeling and simulation well for new and emerging fields.

Biologists can employ modeling and simulation to test their hypotheses about a biochemical reaction network. Although testing the actual system is either too dangerous or cost prohibitive, the cost of building a mathematical model of the biological system and performing experiments on that model frequently is less than the cost of laboratory experimentation. Since the decisions that biologists make as a result of modeling and experimentation have significant consequences, it is crucial that the developed models are credible and reliable, and that biologists understand the limitations of their modeling and simulation efforts. Developers, users, decision makers, and those impacted by the outcome of the model are all concerned with whether the model is correct [121].

Model verification, validation, and testing activities assess the accuracy of a model. Performing these activities also improves the credibility and reliability of a model. The practice of model verification, validation, and testing is essential to the consistent production of models that are useful and correct. Balci [19] and Sargent [121] provide a comprehensive overview of model verification, validation, testing, and accreditation.

Model verification is the process of certifying that transformations of the model from one form to another maintain the fidelity of a model. Model verification ensures that the modeler transforms the model as they intended and that the modeler preserves the accuracy of the model over time. Modeling tools that support model verification allow the modeler to check that the tool operates in the manner that the modeler assumes. Model verification checks that the process of building the model is correct.

Model validation is the process of determining whether a model sufficiently approximates the real system. The definition of the term 'sufficiently' depends upon the purpose of the model. Increasing the validity of a model has cost, so as Balci and Sargent [24] state, it is most efficient to evaluate the model with respect to its intended application. Therefore, the purpose of the model dictates which aspects are important to validate and the standards that the modeler should apply. Different modelers may have different intended purposes for a model. Therefore, it is possible for a model to pass the criteria for acceptability for one modeler but not for another modeler.

Model testing is the process of checking for errors in the model. Model testing determines if the model is functioning properly by subjecting the model to controlled inputs. The modeler designs a model test to perform model verification, model validation, or both activities.

The model experiments that the modeler uses for testing determine the domain of acceptability for the model. Although one form of model experimentation is to compare the model with historically-collected laboratory experiments, the modeler can use other applicable model verification and validation techniques. The intended application of the modeler determines the acceptable range of results from the model tests. Preferably, the modeler fixes the acceptable range before the development and testing of the model, and then works to make the model acceptable. One part of model accreditation, the certification that the model is acceptable for a particular application, comes from the documentation of model verification and validation that the modeler generates from these activities [121].

1.2 Overview of Dissertation

The first part of this dissertation motivates the utility of the mathematical models that the biological modelers construct by illustrating the mathematics, mechanics, and theory that underpin the connection between mathematical models and biological practice. Chapter 2 introduces the building blocks from which biological modelers construct their models. This dissertation studies biological modelers that orient their models in terms of chemical reactions and chemical reaction kinetics. Chemical reactions describe the structure of the biological processes that take place within a cell. Chemical reaction kinetics describe the rate at which the chemical reactions occur.

The corresponding primitive building block for mathematical models is the differential equation. A differential equation defines a family of functions in terms of their rate of change over time. The primary motivators of these rates of change are the chemical reaction kinetic equations from the biological models. This dissertation considers several types of differential equations that biological modelers commonly use. However, the remaining material focuses on ordinary differential equations, which are relatively simple for a computer to evaluate.

The remainder of Chapter 2 explains the algorithm that biological modelers use to convert the biological model to the mathematical model and the justification that they use to interpret the biological significance from the mathematical results. Converting from chemical reactions to differential equations is a laborious yet mechanical process that later chapters in this dissertation address. The justification that this algorithm and the differential equations that it produces are meaningful requires that that chemical reactions describe instantaneous physical events, that chemical reactions take place within an essentially thermodynamically-fixed environment, that Brownian motion evenly mixes the contents of the discretely-bounded compartments of the cell, and that the chemical reactants and products exist in sufficiently-large amounts. The first two of these conditions generally apply to biological systems, while modelers must verify that the second two conditions apply to the problem that they are trying to solve. There is no general method for verifying that these conditions hold for a particular biological system.

The second part of this dissertation describes the development of new theory and practice that biological modelers can use when dealing with these biological and mathematical models. Chapter 3 explores the development of models from a theoretical perspective. Biological modelers today commonly employ ad-hoc modeling processes that they developed through years of practice and experience. Like many other fields that compose or construct works, biological modelers use detail-oriented bottom-up modeling processes, goal-oriented top-down modeling processes, or, more frequently, a combination of the two types of modeling processes. This dissertation examines a particular group of biological modelers that developed a strongly bottom-up modeling process and describes their modeling process.

The remainder of Chapter 3 is then the theoretical development of a new modeling process that is later shown to compare favorably with the original modeling process in terms of reliability, repeatability, efficiency, and correctness. The primary drivers of this new modeling process are well-tested and documented techniques from modeling methodologies. Additionally, the new modeling process focuses heavily on the domain of the studied biological modelers, allowing for further refinements that increase power and ease-of-use of modeling tools but decrease generality. This new modeling process is the basis of the practical developments and tools in the remainder of the dissertation.

Chapter 4 applies the new modeling process to develop tools that the biological modelers can use. The JigCell modeling environment is a suite of applications, programming libraries, and utility programs that focuses on the production, execution, and analysis of models of biochemical reaction networks. The JigCell modeling environment employs the new modeling process to significantly aid biological modelers and improve the model-development experience. The JigCell modeling environment consists of the Model Builder, Run Manager, and Comparator applications that perform major modeling tasks and an experimental tool for parameter estimation. The programming libraries support these applications and further provide the capability to construct new customized applications to meet the needs of the modelers. The JigCell modeling environment and all of its component pieces are open source and employ documented standards for interoperable communication with other tools and applications.

The final part of this dissertation measures the efficacy of the JigCell modeling environment from both theoretical and practical perspectives. Chapter 5 develops a systematized collection of requirements for biological modeling tools based on modeling methodology, domain experience, and user interviews. The major portion of Chapter 5 is an examination of the JigCell applications in relation to these tool standards. The collection of requirements consists of both functional requirements and performance requirements based on the expected trends in model development over the next several years.

Chapter 6 presents a case study applying the JigCell applications to a biological model for cell cycle control in budding yeast. The same modeling group that developed the original modeling process also developed this budding yeast model. The budding yeast model is a large, constructively-built model with many constraining experimental observations. Historically, the modeling group has had great difficulty fine-tuning the budding yeast model due to the number of adjustable control parameters and the expense of model evaluation. The case study in this dissertation examines the speedup of model evaluation when an expert modeler employs automated model evaluation with the JigCell applications as compared to when the expert modeler employs manual model evaluation.

Chapter 7 contains the conclusions of this dissertation, summarizes the contributions that this dissertation makes, and presents the software engineering and evaluation experiences of the JigCell project.

Connections to other works

Much of the material in this dissertation previously appeared in part in publications or online. In most cases, these publications contain abridged versions of the material in this dissertation. This dissertation generally supersedes the earlier publications by providing a more thorough treatment of the material, up-to-date accounts of the software, methods, results, analyses, and experiences, and corrections to the errors found subsequent to publication.

The observational account of the original modeling process and the ensuing modeling process formalizations of Chapter 3 appeared in earlier publications. Allen et al. [6] originally gave the larger model of Section 3.1 as an example. Allen et al. [9] and later Allen et al. [10] first described the original modeling process of Section 3.1, the modeling methodology of Section 3.2, and the revised modeling process of Section 3.3.

Several earlier publications described the components of the JigCell modeling environment of Chapter 4. The introduction to Chapter 4 came from Allen et al. [9] and Allen et al. [10]. Publications that provided portions of the description of the JigCell Model Builder in Section 4.2 and the JigCell Run Manager in Section 4.3 include Allen et al. [6], Allen et al. [9], Allen et al. [10], Vass et al. [135], and online by Allen et al. [8]. Publications that provided portions of the description of the JigCell Comparator in Section 4.4 include Allen et al. [6], Allen et al. [9], Allen et al. [10], and online by Allen et al. [5] and Allen [8]. The description of parameter estimation in Section 4.6 came from Allen et al. [6], Allen et al. [9], Allen et al. [10], and Panning et al. [113].

Chapter 5 has never previously appeared in this form, although portions appeared in other publications. Allen et al. [9] and Allen et al. [10] originally included the introduction to Chapter 5. Allen, Shaffer, and Watson [11] provided the original form of the support techniques in Section 5.1. The remaining sections of Chapter 5 previously appeared online by Allen et al. [8] for an earlier version of the JigCell modeling environment.

Chapter 6 comes largely from the material in Allen et al. [7]. Portions of the introduction to Chapter 6 and the description of experimental phenotypes in Section 6.2.1 first appeared in Panning et al. [113].

Chapter 2

Differential Equations for Biological Models

The cell is a highly ordered and intricate machine in which a wide variety of complex chemical processes take place. Bounded by membranes, cells take in materials and energy from their environment, use these inputs for maintenance and growth, and release back into the environment the waste products and heat. Put together, the chemical processes that take place inside the cell form a tightly-coupled regulatory network that interacts with both the changing environment and the development that occurs within the cell. Alberts et al. [3] provide a general overview of cell biology.

Cells contain many fundamental building blocks, such as nucleotides, amino acids, carbohydrates, and lipids. The cell assembles these fundamental building blocks into macromolecular structures, including enzymes, proteins, and ribosomes, that the DNA and RNA encode using nucleotide sequences. The regulatory network of chemical processes coordinates and directs the construction process of macromolecular structures. The term 'biomolecule' collectively describes all of the molecules involved in the chemical processes that take place within the cell.

Biological modelers seek to map and understand chemical processes through quantitative methods. As biological modelers map the details of individual biological processes, they hope to assemble their discoveries into a roadmap of integrated biological systems. Researchers expect this roadmap to accumulate first into an understanding of simple cellular systems, and then ultimately into an understanding of large and complex eukaryotic organisms. A key technique that biological modelers employ is the use of modeling and simulation to study the temporal evolution of the action and effects of regulatory networks on populations of biomolecules.

Enzymes heavily drive the actions of a regulatory network. An enzyme is a protein that catalyzes a specific chemical process. Catalysis lowers the activation energy that a chemical process requires to occur and thereby greatly enhances the rate at which that chemical process takes place. Although enzymes increase the rate at which a chemical process takes place, enzymes cannot enable the occurrence of thermodynamically unfavorable chemical processes. The presence of enzymes does not change the actual value of the equilibrium point. Instead, the catalytic properties of enzymes result in a more rapid redirection of the population of biomolecules to the equilibrium point than in the equivalent, uncatalyzed chemical process.

Enzyme catalysis is important because the available energy for performing work is limited at constant temperature and pressure. The cell is an essentially isothermal enclosing system around the chemical processes that take place inside. Chemical processes cannot extract useful work from heat energy without changing the temperature or pressure of the enclosing system. The requirement to maintain the highly-ordered structure of the cell disfavors the use of chemical processes that greatly increase the disorder of the cell. Instead, chemical processes must release energy in the form of heat. Enzymes help facilitate the transfer of chemical energy between chemical processes and thereby reduce the energy requirements of the cell.

The chemical processes that cellular enzymes catalyze typically proceed at maximum yield, producing no chemical byproducts. This is unlike many industrial catalytic processes that require refreshment and maintenance due to the formation of chemical byproducts. By making use of enzymes that do not form chemical byproducts during catalysis, the cell further reduces its energy and material needs.

Biologists classify the chemical processes that occur in the cell according to their function. Transport processes move proteins, materials, and products from one location in the cell to another and across cellular membranes. Diffu-

sion transports many biomolecules throughout the cell, while other biomolecules have directed, active transport that delivers the biomolecules to specific sites within the cell. Intercellular and intracellular signaling processes communicate information about the cell and coordinate the response of the cell to environmental stimuli.

Many of the chemical processes that this dissertation uses for examples regulate cellular metabolism and the cell division cycle. Cellular metabolism and cell cycle regulation consist of chemical processes that are usually extraordinarily stable and highly conserved, present in nearly identical forms across many organisms. The cell responds to environmental perturbations by employing a controlled response that restores the cell to its basal state. A mutation that disrupts a particular chemical process involved in cellular metabolism or cell cycle regulation is sometimes survivable. The resulting mutant is capable of surviving under standard conditions. However, the mutant is less competitive in stressful environments. Biological modelers consider metabolic and cell cycle functional processes important to study because these processes control how cells grow, maintain themselves, and divide. The cell cycle tightly interweaves with all of the activities that occur within the cell.

This chapter introduces the building blocks from which modelers construct biological and mathematical models. Section 2.1 describes chemical reactions and Section 2.2 describes chemical reaction kinetics. These two components are together the basis of the biological models that this dissertation describes. The modeling process and modeling software that later chapters describe treat chemical reactions as indivisible, primitive elements in building models. Then, Section 2.3 describes the corresponding primitive element for mathematical models, the differential equation.

Section 2.4 ties together the biological and mathematical forms of a model by illustrating the process of converting from chemical reactions and kinetics to differential equations. Later chapters draw heavily upon the algorithms that Section 2.4 introduces, and this dissertation afterwards frequently treats biological and mathematical models as equivalents. Finally, Section 2.5 gives the physical chemical background that explains why modelers can apply the algorithms in Section 2.4 to their problems.

Contents

2.1 Chemical Reactions

Chemical reactions are processes that convert a fixed collection of chemical species, the chemical reactants, to another fixed collection of chemical species, the chemical products. Chemical reactions can involve one chemical reactant (monomolecular chemical reactions), two chemical reactants (bimolecular chemical reactions), or a greater number of chemical reactants (trimolecular chemical reactions, etc.). Most of the chemical reactions that this dissertation considers are monomolecular or bimolecular chemical reactions. In special cases, such as open chemical systems, there are chemical reactions that have zero chemical reactants.

A chemical reaction equation consists of chemical reactants followed by an arrow pointing to the chemical products, as in $\mathrm{A_{red}} + \mathrm{B_{oxi}} \longrightarrow \mathrm{A_{oxi}} + \mathrm{B_{red}}$. This particular chemical reaction is an example of an oxidation-reduction chemical reaction that transfers electrons from one chemical species to another.

Sometimes, a chemical reaction requires more than one instance of a chemical reactant or a chemical reaction produces more than one instance of a chemical product. The stoichiometry is the number of instances of a chemical

reactant or product that a single chemical reaction event involves. Writing a number before the chemical reactant or product declares the stoichiometry of that chemical species in the chemical reaction. When no number appears with a chemical species, the stoichiometry of that chemical species in the chemical reaction is one. For example, the oxidation-reduction chemical reaction equation,

$$2\mathrm{Fe}^{3+} + \mathrm{Sn}^{2+} \longrightarrow 2\mathrm{Fe}^{2+} + \mathrm{Sn}^{4+},$$

corresponds to the chemical reaction equation with two copies of the chemical species Fe^{3+} and two copies of the chemical species Fe^{2+},

$$\mathrm{Fe}^{3+} + \mathrm{Fe}^{3+} + \mathrm{Sn}^{2+} \longrightarrow \mathrm{Fe}^{2+} + \mathrm{Fe}^{2+} + \mathrm{Sn}^{4+}.$$

The chemical reactions given so far are irreversible chemical reaction equations. An irreversible chemical reaction equation only proceeds from the chemical reactants to the chemical products. A reversible chemical reaction equation can proceed in either direction. Switch the labels 'reactant' and 'product' when the chemical reaction proceeds in reverse. Reversible chemical reaction equations have a two-way arrow, as in $2\mathrm{NO}_2 \rightleftharpoons \mathrm{N}_2\mathrm{O}_4$, which is equivalent to the pair of irreversible chemical reaction equations,

$$2\mathrm{NO}_2 \longrightarrow \mathrm{N}_2\mathrm{O}_4 \qquad \text{and} \qquad \mathrm{N}_2\mathrm{O}_4 \longrightarrow 2\mathrm{NO}_2.$$

Many of the chemical reactions that take place within the cell are reversible chemical reactions. However, the pair of chemical reactions generally has an equilibrium point that favors one of the chemical reactions over the other at any particular moment.

Chemical reactions take place within an enclosed universe, known as the chemical system. Modelers frequently do not represent every chemical species that is present inside the cell when describing the chemical system. Simplifying models by omitting chemical species makes feasible the simulation of interesting biological problems. A model commonly omits readily available chemical species. The chemical system contains these chemical species in ample amounts, and an instance of the species is always available when a chemical reaction needs one. For example, a modeler can write the phosphorylation and dephosphorylation of the chemical species A using the chemical reaction equation $\mathrm{A} \rightleftharpoons \mathrm{A}^{\mathrm{P}}$ without regard to the availability of the phosphate groups that this chemical reaction transfers.

The chemical reaction equations that modelers write sometime indicate that a chemical species spontaneously appears or disappears because of the chemical reaction. Chemical reactions and systems in which a chemical species spontaneously appears or disappears are 'open'. Similarly, chemical reactions and systems in which none of the chemical species spontaneously appear or disappear are 'closed'. Open chemical systems exchange energy and matter between the system and its containing environment. The distinction between open and closed chemical systems plays an important role in the existence of moiety conservation relations, which Section 2.4.2 describes in detail.

Chemical reactions in which no chemical reactants take part are 'synthesis' reactions and written with a dot for the reactants, $\bullet \longrightarrow \mathrm{X}$. Chemical reactions in which no chemical products take part are 'degradation' reactions and written with a dot for the products, $\mathrm{X} \longrightarrow \bullet$. It does not make sense to talk about a chemical reaction with neither chemical reactants nor products. Synthesis and degradation reactions arise when the chemical system omits chemical species and when chemical reactions transport chemical species into and out of the enclosing universe. A chemical system that includes synthesis or degradation reactions is an example of an open chemical system.

2.2 Chemical Reaction Kinetics

In addition to the structure of a chemical reaction, the chemical reaction equation, simulation requires knowing the rate at which the chemical reaction events take place, called the chemical reaction velocity. The kinetic formula of a chemical reaction is a formula that has chemical species concentrations as the variables and parameters called kinetic rate constants. Evaluating the kinetic formula of a chemical reaction using the chemical species population known for the chemical system gives the chemical reaction velocity for that chemical reaction.

The order of a kinetic formula is the number of chemical species concentrations that the formula contains. Mathematicians often write an order with respect to a particular set. In this case, the set typically is the chemical reactants

for the chemical reaction. Thus, the chemical reaction equation $A \longrightarrow B$ with the kinetic formula $[\mathrm{A}]$, where $[\mathrm{A}]$ is the concentration of the chemical species A, is a first-order kinetic formula with respect to the chemical reactants. The chemical reaction equation $A \longrightarrow B$ with the kinetic formula $[\mathrm{C}]$ is a zeroth-order kinetic formula with respect to the chemical reactants. When this dissertation gives the order of a kinetic formula, the order is with respect to the set of chemical reactants for the chemical reaction unless stated otherwise.

The most fundamental type of kinetic formula is a mass action kinetic formula. Mass action kinetic formulas result from the physical observation that the propensity of chemical species to interact is proportional to the product of the concentrations of those chemical species. The chemical reaction velocity, v, of the general chemical reaction equation

$$r_1\mathrm{S}_1 + r_2\mathrm{S}_2 + \ldots + r_n\mathrm{S}_\mathrm{n} \longrightarrow \quad \cdots$$

with mass action kinetics is a kinetic rate constant k multiplied by the concentrations of the chemical reactants,

$$v = k \prod_i^n [\mathrm{S}_\mathrm{i}]^{r_i},$$

where the thermodynamics of the chemical reaction dictate the kinetic rate constant k. Mass action kinetic formulas have an order equal to the number of chemical reactants that a chemical reaction event requires to occur, $\sum_{i=1}^{n} r_i$.

When writing a chemical reaction equation that uses mass action kinetics, it is typical to write the kinetic rate constant atop the arrow between the chemical reactants and products. Thus, the chemical reaction equation $\mathrm{A} + \mathrm{B} \longrightarrow \mathrm{C}$ with mass action kinetics and a kinetic rate constant k_f is written as

$$\mathrm{A} + \mathrm{B} \xrightarrow{k_f} \mathrm{C}.$$

Similarly, the reversible chemical reaction equation $\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$ with mass action kinetics, a kinetic rate constant k_f in the forward direction, and a kinetic rate constant k_r in the reverse direction is written as

$$\mathrm{A} + \mathrm{B} \underset{k_r}{\overset{k_f}{\rightleftharpoons}} \mathrm{C}.$$

The forward kinetic formula of this chemical reaction equation is $k_f[\mathrm{A}][\mathrm{B}]$, and the backward kinetic formula of this chemical reaction equation is $k_r[\mathrm{C}]$.

Michaelis-Menten kinetics

The two-step process that describes simple enzyme catalysis of the chemical reactant S into the chemical product P via the enzyme E is:

1. The chemical species S meets with the enzyme E to form the complex ES,
2. and the complex ES proceeds to form the chemical product P.

The formation of the chemical product liberates the enzyme E for future chemical reactions. Of course, these two chemical reactions can also proceed in reverse. The chemical species P meets with the enzyme E, forming a complex. Finally, that complex then dissociates back to its individual components, the chemical species S and the enzyme E. These two reversible chemical reactions join to give the chemical reaction equation

$$\mathrm{E} + \mathrm{S} \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} \mathrm{ES} \underset{k_{-2}}{\overset{k_2}{\rightleftharpoons}} \mathrm{E} + \mathrm{P}. \tag{2.1}$$

The Michaelis-Menten kinetic formula [37, 96] approximates the simple scheme for enzyme catalysis that uses the system of four chemical reactions of Equation 2.1 and gives a single formula for the chemical reaction equation

$\mathrm{E} + \mathrm{S} \longrightarrow \mathrm{E} + \mathrm{P}$. The standard derivation of Michaelis-Menten kinetics invokes two assumptions to create the approximation. Other approximate schemes for enzyme catalysis provide results similar to Michaelis-Menten kinetics but use different underlying assumptions. The biological models that this dissertation examines are generally invariant to the particular approximation scheme that the modeler chooses to use.

- The first assumption is that none of the chemical species P reforms into the complex. This assumption, equivalent to setting the kinetic rate constant k_{-2} to zero, is essentially true when the concentration of the chemical species P is small.

- The second assumption, called the steady state assumption, is that the concentration of the complex ES does not change even though the concentrations of the chemical species S and P are changing. The steady state assumption requires that the rate of formation of the complex ES equal the catalytic rate of the overall chemical reaction plus the rate at which the complex ES reverts to its individual components, the chemical species S and the enzyme E.

Applying these two assumptions to Equation 2.1, the reaction velocity of the chemical reaction equation

$$\mathrm{ES} \xrightarrow{k_2} \mathrm{E} + \mathrm{P}$$

limits the overall catalytic rate. Thus, the catalytic rate $v = k_2[\mathrm{ES}]$. Also, by the steady state assumption, $k_1[\mathrm{E}][\mathrm{S}] = (k_{-1} + k_2)[\mathrm{ES}]$. Solving this equation for the concentration of the complex ES gives

$$[\mathrm{ES}] = \frac{[\mathrm{E}][\mathrm{S}]}{(k_{-1} + k_2)/k_1}. \tag{2.2}$$

Biologists call the constant expression $(k_{-1} + k_2)/k_1$ in Equation 2.2 the Michaelis constant and use the symbol k_m.

The enzyme converts between the free form E and the complex ES. The total amount of the enzyme present in the chemical system does not change over time, and the sum $[\mathrm{E}] + [\mathrm{ES}] = E_T$ holds true for some constant total amount E_T. Substituting this conservation relation into Equation 2.2 and solving for the concentration $[\mathrm{ES}]$ produces

$$[\mathrm{ES}] = \frac{E_T[\mathrm{S}]}{k_m} - \frac{[\mathrm{ES}][\mathrm{S}]}{k_m} = \frac{E_T[\mathrm{S}]}{k_m(1 + [\mathrm{S}]/k_m)} = \frac{E_T[\mathrm{S}]}{k_m + [\mathrm{S}]}. \tag{2.3}$$

Therefore, substituting Equation 2.3 into the original catalytic rate equation gives the kinetic formula,

$$v = \frac{k_2 E_T[\mathrm{S}]}{k_m + [\mathrm{S}]}.$$

However, the product $k_2 E_T$ is simply the catalytic rate when all of the enzyme is in the form of the complex ES. Since the rate-limiting step in this system of chemical reactions is the production of chemical products from the complex ES, the product $k_2 E_T$ is also the maximum rate at which the chemical reaction can take place, called $v_{\max}$. Thus, it is customary to instead write a Michaelis-Menten kinetic formula as

$$v = \frac{v_{\max}[\mathrm{S}]}{k_m + [\mathrm{S}]}, \tag{2.4}$$

using the parameters $v_{\max}$ and k_m. When the concentration of the chemical reactant S is much greater than the concentration of the enzyme E, the Michaelis-Menten formula gives the limiting catalytic rate $v_{\max}$, which is zeroth-order in terms of the reactants.

The models in this dissertation use Michaelis-Menten kinetic formulas to reduce the number of intermediate chemical products written. Reducing the number of intermediate chemical species makes the models simpler and easier to understand. However, some model simulation schemes, such as those that Section 2.5.1 describes, require that the

modeler 'unpack' complex kinetic formulas, such as Michaelis-Menten kinetic formulas, into chemical reaction equations with elementary mass action kinetic formulas.

Hill equation

Although the Michaelis-Menten approximation for enzyme catalysis is a useful simplification, some enzymes do not have Michaelis-Menten kinetics. The solution curve for the kinetic formula of Equation 2.4 has a hyperbolic shape. The slope of the catalytic rate curve is initially steep as the concentration of the chemical product P is small. As time progresses, the action of the enzyme moves the system towards equilibrium and the catalytic rate curve levels off.

Allosteric enzymes are enzymes that have multiple binding sites that interact as the enzyme encounters chemical species. When the binding sites of an allosteric enzyme are cooperative, the binding of a chemical species to the enzyme at one of the binding sites improves the chances of a chemical species successfully binding to the enzyme at another binding site. Enzymes that exhibit this cooperative behavior have a catalytic rate curve with a sigmoidal shape. Initially, the slope of the catalytic rate curve is nearly flat. As time progresses, chemical species occupy a few of the binding sites, and the catalytic rate curve grows steeper. Finally, the action of the enzyme moves the system near equilibrium, and the catalytic rate curve levels off again.

Modelers commonly use the Hill equation,

$$v = \frac{v_{\max}[\mathrm{S}]^h}{k_{0.5}^h + [\mathrm{S}]^h},$$

to empirically match sigmoidally-shaped catalytic rate curves. In the Hill equation, the parameter h controls the shape of the sigmoidal curve, with $h = 1$ reducing to a Michaelis-Menten kinetic formula. Note that if $k_m = [\mathrm{S}]$ in Equation 2.4, the chemical reaction velocity v is equal to $0.5v_{\max}$. Due to this physical interpretation, the parameter k_m is written as $k_{0.5}$ when it appears in non-Michaelis-Menten kinetic formulas.

This brief introduction to chemical reaction kinetics covered the types of kinetic equations that appear in the examples in this dissertation and in many of the models that the references describe. Modelers have named and used in models a vastly greater number of chemical reaction kinetic equations. Hammes [64] gives a general overview of the kinetic formulas that many typical enzymatic processes use. Hofmeyr and Cornish-Bowden [68] go into more detail about the Michaelis-Menten and Hill equations, in particular giving derivations for a general reversible form of each. A modeler can use any equation as a kinetic formula, but many modelers primarily use kinetic formulas that derive from known physical and biological phenomena.

2.3 Differential Equations

Modelers often want to know how a chemical species, say A, changes over time. The formula for the concentration of chemical species A as a function of time is labeled $[\mathrm{A}](t)$, or as the variable $[\mathrm{A}]$. The chemical reactions that involve the chemical species A dictate the formula for $[\mathrm{A}](t)$. As Section 2.4 shows, each chemical reaction that uses the chemical species A as a reactant contributes a negative term to the differential equation for $[\mathrm{A}](t)$ and each chemical reaction that forms A as a product contributes a positive term.

Often, it is not possible to write an explicit equation for $[\mathrm{A}](t)$. Instead, the modeler writes an equation that contains derivatives of $[\mathrm{A}](t)$ and other chemical species concentrations, creating a differential equation. If the chemical species A is synthesized at a constant rate, k, then the concentration $[\mathrm{A}]$ is given by the differential equation

$$\frac{d[\mathrm{A}]}{dt} = k. \tag{2.5}$$

Dependent variables are those variables whose derivative occurs in the differential equation. Independent variables are those variables for which derivatives are taken in respect of in the differential equation. A parameter is a constant that varies in other equations of the same general form. In Equation 2.5, $[\mathrm{A}]$ is a dependent variable, t is an independent variable, and k is a parameter. A differential equation can contain an unlimited number of dependent variables,

independent variables, and parameters. Most of the differential equations in this dissertation have a single independent variable but many dependent variables and parameters.

There are several important classifiers for differential equations. An ordinary differential equation is a differential equation that takes only ordinary derivatives of the dependent variables. Although this dissertation uses ordinary differential equations extensively, this is only a small introduction to the theory of ordinary differential equations. For a more thorough coverage, consult references such as Arnold and Cooke [13], Boyce and DiPrima [36], and Rainville [114].

The order of a differential equation is the highest order of derivative that occurs in the equation. A system of differential equations with order greater than one has a corresponding system of differential equations with order one. To produce such a system, introduce new variables for the higher order derivatives until every derivative is expressible as the first derivative of some existing variable.

Finally, a differential equation is linear if every term of the differential equation is linear in the dependent variables and the derivatives of the dependent variables. A linear, first-order ordinary differential equation has the general form

$$a_0(t)y' + a_1(t)y + a_2(t) = 0,$$

where y is a variable and y' is the first derivative of y with respect to t. Most of the differential equations that this dissertation discusses are first-order ordinary differential equations. However, only the simplest of the differential equations, such as Equation 2.5, are linear. Non-linear terms frequently arise in the differential equations generated from biological models due to the use of mass action kinetic formulas with chemical reactions that have multiple chemical reactants and due to the use of more complicated kinetic formulas, such as the Michaelis-Menten or Hill kinetic formulas, which lead to differential equations that contain products of dependent variables.

2.3.1 Systems of Ordinary Differential Equations

When a chemical reaction involves two or more chemical species, the chemical reaction equation gives rise to a system of differential equations. The differential equations in the system have different parameters and dependent variables, but share a single independent variable t. The chemical reaction equation $\mathrm{A} + 2\mathrm{B} \longrightarrow 3\mathrm{C}$ with mass action kinetics and a rate constant of k generates the system of simultaneous ordinary differential equations

$$\frac{d[\mathrm{A}]}{dt} = -k[\mathrm{A}][\mathrm{B}]^2, \tag{2.6}$$

$$\frac{d[\mathrm{B}]}{dt} = -2k[\mathrm{A}][\mathrm{B}]^2, \tag{2.7}$$

$$\frac{d[\mathrm{C}]}{dt} = 3k[\mathrm{A}][\mathrm{B}]^2. \tag{2.8}$$

Section 2.4 describes how to construct a system of differential equations from multiple chemical reaction equations.

The general form of a system of first-order ordinary differential equations is

$$\begin{aligned} y_1' &= f_1(t, y_1, y_2, \ldots, y_n), \\ y_2' &= f_2(t, y_1, y_2, \ldots, y_n), \\ &\vdots \\ y_n' &= f_n(t, y_1, y_2, \ldots, y_n), \end{aligned}$$

where y_1, y_2, …, y_n are variables corresponding to the functions $y_1(t)$, $y_2(t)$, …, $y_n(t)$, and y_i' is the derivative of y_i with respect to t. Vector notation encodes this system of differential equations more compactly as

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}). \tag{2.9}$$

A solution to Equation 2.9 is an n-dimensional vector function, $\mathbf{y}(t)$, such that each component of $\mathbf{y}(t)$ is differentiable

with respect to t and $\mathbf{y} = \mathbf{y}(t)$ satisfies the system of differential equations. Both of these conditions must hold along the interval $[t_0, t_{\max}]$ that the modeler considers interesting. The time t_0 is the moment when numerical initial conditions, $\mathbf{y}(t_0) = \mathbf{y}_0$, are known for the system of differential equations. The time $t_{\max}$ is the last moment that the modeler wants to examine. An initial value problem combines a system of differential equations along with numerical initial conditions. Unfortunately, a modeler usually cannot construct a solution vector function $\mathbf{y}(t)$ that solves a biologically interesting initial value problem.

Solving a System of Ordinary Differential Equations

Finding a solution for a system of differential equations generated from chemical reaction equations is often quite difficult. Chemical reaction equations frequently translate into non-linear differential equations, few of which have analytical solutions. Instead of hoping to find an analytical solution to the system of differential equations, modelers instead numerically approximate a solution to the system of differential equations.

Using the numerical solution for the system of differential equations, the modeler constructs a time series plot of the chemical species populations as a function of time. Euler's method is a simple numerical technique for solving differential equations by extending the solution vector according to a local approximation of $\mathbf{f}(t, \mathbf{y})$.

1. Start at the known point $\mathbf{y_0}$ at the time t_0,
2. construct a tangent line to the graph $(t, \mathbf{y}(t))$ with slope $\mathbf{f}(t, \mathbf{y})$,
3. follow the tangent line to the approximate point $\mathbf{y}_1$ at the time t_1, where $t_1 > t_0$,
4. and repeat this process to produce successive approximations $\mathbf{y}_2, \mathbf{y}_3, \ldots, \mathbf{y}_r$.

The overall approximation for the solution vector function $\mathbf{y}(t)$ is then

$$\mathbf{y}_{r+1} = \mathbf{y}_r + \mathbf{f}(t_r, \mathbf{y}_r)(t_{r+1} - t_r), \quad (t_0 < t_1 < \cdots < t_r < t_{r+1} < \cdots \leq t_{\max}). \tag{2.10}$$

The global error of Euler's method is proportional to the size of the differences $t_{r+1} - t_r$ [36]. Euler's method generates accurate solutions for simple systems of differential equations, such as Equations 2.6–2.8. However, applying Euler's method to large biological systems frequently leads to inaccurate results.

Solving a Stiff System of Ordinary Differential Equations

A system of differential equations that a modeler generates from many chemical reaction equations is typically stiff. A stiff system of differential equations is one that is sensitive to small perturbations in the solution value. In many cases, a stiff system of differential equations comes from a biological system where conditions change on two vastly-different time scales. Euler's method is relatively accurate for a simple system of differential equations. However, Euler's method is unstable for a stiff system of differential equations even when the sizes of the time steps are relatively small. Producing an accurate solution to a stiff system of differential equations using Euler's method is expensive.

The backward Euler's method is a variation of Euler's method that uses backward difference formulas to construct an implicit approximation for the solution vector function $\mathbf{y}(t)$. Unlike Euler's method, the backward Euler's method is stable regardless of the size of the time steps. The backward Euler's method replaces the tangent line approximation used in Euler's method with the backward difference formula,

$$\mathbf{y}'(t_{r+1}) \approx \frac{\mathbf{y}_{r+1} - \mathbf{y}_r}{t_{r+1} - t_r}. \tag{2.11}$$

Substituting Equation 2.11 to produce an equivalent to Equation 2.10 gives the backward Euler formula,

$$\mathbf{y}_{r+1} = \mathbf{y}_r + \mathbf{f}(t_{r+1}, \mathbf{y}_{r+1})(t_{r+1} - t_r). \tag{2.12}$$

Since $\mathbf{y}_{r+1}$ is defined implicitly, computing the value of $\mathbf{y}_{r+1}$ requires solving Equation 2.12.

Euler's method and the backward Euler's method are first-order methods for numerically solving differential equations. Higher-order numerical methods exist that can produce more efficiently an accurate approximation to a system of differential equations. Shampine and Gordon [127], as well as Ascher and Petzold [15], describe the theory and application of higher-order methods for solving stiff systems of ordinary differential equations.

2.3.2 Other Types of Differential Equations

This dissertation uses several other types of differential equations that also regularly appear in biological modeling. This section gives a brief overview of differential-algebraic equations, delay differential equations, partial differential equations, and stochastic differential equations.

Differential-algebraic equations

Section 2.3.1 considered systems of differential equations that have the form given by Equation 2.9. In many biological problems, algebraic equations augment the system of differential equations to produce a system of differential-algebraic equations. The general form of a system of ordinary first-order differential-algebraic equations is

$$\begin{aligned} \mathbf{y}' &= \mathbf{f}(t, \mathbf{y}, \mathbf{z}), \\ \mathbf{0} &= \mathbf{g}(t, \mathbf{y}, \mathbf{z}), \end{aligned}$$

where the $\mathbf{z}$ variables correspond to the functions $\mathbf{z}(t)$ whose time derivatives do not appear. The $\mathbf{y}$ variables are the 'differential variables' and the $\mathbf{z}$ variables are the 'algebraic variables'. Algebraic variables do not need initial conditions. Given initial conditions, $\mathbf{y}_0$, for the differential variables at time t_0, the solution for $\mathbf{z}$ of $\mathbf{g}(t_0, \mathbf{y}_0, \mathbf{z}) = \mathbf{0}$ provides the initial conditions, $\mathbf{z}_0$, for the algebraic variables. Not specifying initial conditions for the algebraic variables ensures that the joint initial conditions $(\mathbf{y}_0; \mathbf{z}_0)$ are consistent for the initial value problem of the system of differential-algebraic equations. However, the algorithm to solve for $\mathbf{z}$, commonly an iterative version of Newton's method, may need an initial guess for $\mathbf{z}_0$.

When the equation $\mathbf{g}(t, \mathbf{y}, \mathbf{z}) = \mathbf{0}$ is solvable for $\mathbf{z}$ given values of $\mathbf{y}$ and t, the system of differential-algebraic equations is not much harder to solve than a system of ordinary differential equations. Systems of differential-algebraic equations for which this condition is not true are 'high index'. High index systems of differential-algebraic equations are considerably more difficult to solve [57, 91]. This dissertation uses differential-algebraic equations in several places, primarily for conservation relations, which Section 2.4.2 discusses.

Partial differential equations

The systems of ordinary differential equations examined so far describe chemical systems that are spatially homogeneous. Spatially heterogeneous chemical systems require equations that include the spatial position in the function for the differential variables in the chemical system. Partial differential equations, equations that take a partial derivative of one or more of the differential variables, are a commonly used method for describing spatially heterogeneous chemical systems. Chemical processes that incorporate both chemical reactions and chemical species diffusion frequently use partial differential equations that combine the chemical reaction kinetics with diffusion terms,

$$\frac{\partial \mathbf{y}}{\partial t} = \mathbf{f}(t, \mathbf{y}) + \text{diffusion terms},$$

where a parameterized function, such as $k\nabla^2\mathbf{y}$, describes the diffusion process.

A famous example of a spatially heterogeneous system that uses partial differential equations is the Fourier heat equation [88]. Starting from physical first principles, Fourier showed that the temperature of a solid body, $T(t, \mathbf{p})$, at the point $\mathbf{p}$ in 3-space and time t, is given by a partial differential equation,

$$\frac{\partial T}{\partial t} = k\nabla^2 T,$$

which is the heat equation. The parameter k is a value that depends upon the conductivity of heat in the solid body.

Solving a system of partial differential equations is typically numerically intensive. The computer must sample the differential variables that represent the chemical system at many different spatial locations [98]. Modelers frequently use partial differential equations with chemical systems for which the well-stirred assumption does not hold. Section 2.5.1 describes the well-stirred assumption in detail. Important partial differential equations that this dissertation discusses include the stochastic chemical master equation (Equation 2.19), a fundamental equation that describes the population of chemical species as a function of time. Modelers usually approximate the stochastic chemical master equation rather than trying to solve the equation directly.

Delay differential equations

A delay differential equation is a differential equation that is a function of the solution to the differential equation beyond some differential time step dt. A frequent source of delay differential equations in biological modeling is transport processes. Transport processes model the movement of chemical species from one location to another in the cell. The general form of a system of ordinary first-order delay differential equations is

$$\begin{aligned} \mathbf{y}' &= \mathbf{f}(t; \tau_1, \ldots, \tau_m, \mathbf{y}), \qquad t \geq t_0, \\ \mathbf{y}(t; \tau) &= \mathbf{\Psi}(t), \qquad t_0 - \tau_m \leq t \leq t_0, \end{aligned}$$

where $\tau_1, \ldots, \tau_m$ is a list of backward time differences into the solution of the system of differential equations with $0 < \tau_1 < \tau_2 < \cdots < \tau_m$ and $\mathbf{\Psi}$ is a real-valued function on $[t_0 - \tau_m, t_0]$ that supplies a solution for the system of differential equations before the initial conditions given at time t_0. Frequently, it is convenient to define $\mathbf{\Psi}$ so that $\mathbf{\Psi}(t) = \mathbf{y}_0$ for all times $t_0 - \tau_m \leq t \leq t_0$. In this case, the system of differential equations is in a quiescent phase before the time interval $[t_0, t_{\max}]$ that the modeler is interested in studying.

Ordinary delay differential equations describe many simple diffusion and signaling processes. Delay differential equations combine with differential-algebraic equations or partial differential equations in the way that one would expect. Systems of delay differential equations are particularly difficult to solve [28, 49]. Like partial differential equations, a common use of delay differential equations is chemical systems for which the well-stirred assumption does not hold.

Stochastic differential equations

In 1908, Paul Langevin [83] presented a differential formula for the position of a particle influenced by Brownian motion that duplicated a result of Albert Einstein. The Langevin formula combines a deterministic ordinary differential equation with a noisy differential term to produce a stochastic differential equation

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}) + \mathbf{h}(t, \mathbf{y}) G(t), \tag{2.13}$$

where $\mathbf{h}(t, \mathbf{y})$ is the intensity of the noise and the $G(t)$ are independent Gaussian variables.

A complication in the solution of stochastic differential equations is that the derivative does not exist due to the presence of the noise term. Instead, Equation 2.13 is rewritten in differential form as

$$d\mathbf{y} = \mathbf{f}(t, \mathbf{y})\, dt + \mathbf{h}(t, \mathbf{y})\, dW(t),$$

where $W(t)$ is a Wiener process [76], and interpreted as the integral equation

$$\mathbf{y}(t) = \mathbf{y}_0 + \int_{t_0}^{t} \mathbf{f}(t, \mathbf{y})\, dt + \int_{t_0}^{t} \mathbf{h}(t, \mathbf{y})\, dW(t).$$

Although the first integral is an ordinary Riemann integral, the meaning of the second integral is less clear. The second integral cannot be a Riemann-Stieltjes integral because the sample paths of a Wiener process are not of bounded variation. Instead, the second integral is a stochastic integral.

A further complication in the solution of stochastic differential equations is that there are many qualitatively different, rational interpretations for what the stochastic integral means. The Itô stochastic calculus [73] and the Stratonovich stochastic calculus [79] define two of the possible interpretations for a stochastic integral. Stochastic differential equations appear in the chemical Langevin equation (Equation 2.20), which is an intermediate stop in the derivation from the stochastic chemical master equation to describing chemical systems with ordinary differential equations. However, this dissertation does not examine solutions to the chemical Langevin equation and does not consider the nuances that separate the Itô and Stratonovich stochastic calculuses.

2.4 Biochemical Reaction Networks

Models of interesting biological processes consist of many coupled chemical reactions, along with the corresponding chemical reaction kinetics. A biochemical reaction network is an aggregate of multiple chemical reactions. Biochemical reaction networks are graphs that capture the chemical products and reactants at the vertices and represent chemical reactions that create, destroy, and convert these chemical species using labeled directed edges. The treatment of biochemical reaction networks in this dissertation is similar to Aris [12] and Feinberg [53].

There are many ways to communicate the structure of a biochemical reaction network. A simple way of representing biochemical reaction networks is to list the chemical reactions and reaction kinetics that the network contains. Table 2.1 describes a biochemical reaction network that contains four chemical species, A, B, C, and D, and five chemical reactions. For simplicity, all of the chemical reaction kinetics in this example are mass action kinetics. Note that the third kinetic formula is $k[\mathrm{C}]^2$ instead of $2k[\mathrm{C}]$.

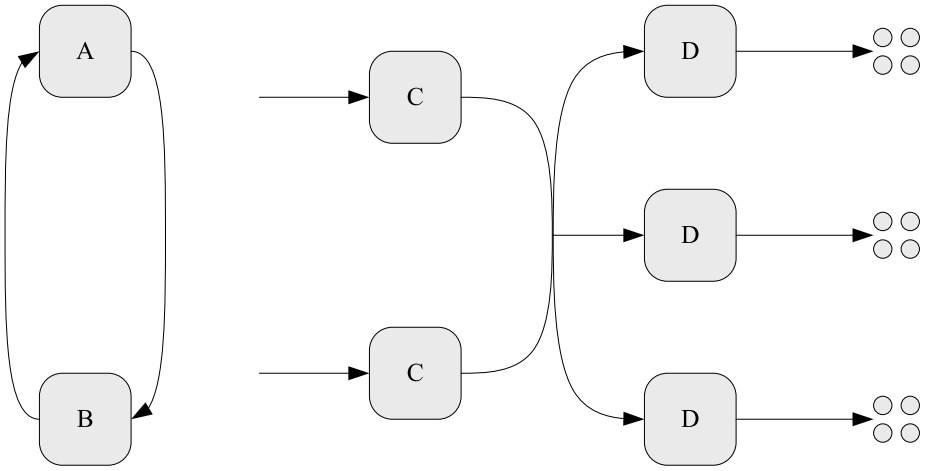
Table 2.1: Chemical reaction equations using mass-action kinetics for a simple set of chemical reactions.

Chemical reaction equation	Kinetic formula
$\mathrm{A} \underset{k_b}{\overset{k_a}{\rightleftharpoons}} \mathrm{B}$	$k_a[\mathrm{A}]$, $k_b[\mathrm{B}]$
$\bullet \xrightarrow{k_c} \mathrm{C}$	k_c
$2\mathrm{C} \xrightarrow{k} 3\mathrm{D}$	$k[\mathrm{C}]^2$
$\mathrm{D} \xrightarrow{k_d} \bullet$	$k_d[\mathrm{D}]$

The biochemical reaction network in this example is slightly unusual because it consists of several short and unlinked collections of chemical reactions. A more typical, but far larger, biochemical reaction network tends to have long series of chemical reactions joined sequentially. Chemical reactions linked by a common intermediate chemical can transfer chemical energy from one chemical reaction to the next. In the isothermal environment of the cell, chemical reactions without a common intermediate chemical cannot transfer chemical energy, and any excess energy that the chemical reaction produces dissipates as heat.

A wiring diagram is a graphical depiction of a biochemical reaction network. A wiring diagram has labeled nodes that represent the chemical species and arcs that represent the chemical reactions. Typically, modelers do not draw the chemical reaction kinetics directly on the wiring diagram but instead provide them separately. Figure 2.1 is a wiring diagram that corresponds to the biochemical reaction network given in Table 2.1. It is easy to see the structure of the biochemical reaction network when looking at the wiring diagram. However, because the chemical reaction kinetics are not present in the wiring diagram, a modeler still needs the table of chemical reaction equations and kinetic formulas to get this vital information about the biochemical reaction network. Finally, a system of differential equations can mathematically represent the biochemical reaction network. The remainder of this section covers the creation of a system of ordinary differential-algebraic equations that correspond to a biochemical reaction network.

Figure 2.1: Wiring diagram for the simple set of chemical reactions in Table 2.1.



2.4.1 Modeling a Biochemical Reaction Network with Differential Equations

The stoichiometry matrix of a biochemical reaction network is a real-valued, rectangular matrix that describes the topology of the network and the transfer of mass that results from chemical reaction events. The stoichiometries of chemical species in the reactions that make up the biochemical reaction network provide the entries of a stoichiometry matrix. The stoichiometry matrix does not include chemical reaction velocities. The stoichiometry matrix for a biochemical reaction network does not change during simulation.

A biochemical reaction network with m chemical species and n chemical reactions has an m by n stoichiometry matrix. The columns of the stoichiometry matrix correspond to chemical reactions; the rows of the stoichiometry matrix correspond to chemical species. The number at the intersection of a row and column in the stoichiometry matrix, a 'stoichiometric coefficient', gives the effect on the population of chemical species from a single occurrence of that chemical reaction event. Positive stoichiometric coefficients correspond to the chemical products of a chemical reaction. Negative stoichiometric coefficients correspond to the chemical reactants of a chemical reaction. The magnitude of the stoichiometric coefficients indicates the quantity of substance that each chemical reaction event converts. When a chemical reaction does not involve a particular chemical species, that stoichiometric coefficient is zero.

This dissertation labels a stoichiometry matrix with a script $\mathcal{S}$ and labels stoichiometric coefficients s_{ij} where $0 < i \leq m$ and $0 < j \leq n$.

$$\mathcal{S} = \begin{bmatrix} s_{11} & s_{12} & \cdots & s_{1n} \\ s_{21} & s_{22} & \cdots & s_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m1} & s_{m2} & \cdots & s_{mn} \end{bmatrix}$$

Building a system of ordinary differential equations from the stoichiometry matrix

Once the modeler computes the stoichiometry matrix of a biochemical reaction network, the process of converting that biochemical reaction network into a system of ordinary differential equations is straightforward. A stoichiometric coefficient s_{ij} gives the quantity of the chemical species with index i that an occurrence of the chemical reaction event with index j requires. Each chemical reaction has a chemical reaction velocity, v_j for the jth chemical reaction, that describes how frequently chemical reaction events occur. The chemical reaction velocity vector $\mathbf{v} = [v_1(\mathbf{C}) \quad v_2(\mathbf{C}) \quad \ldots \quad v_n(\mathbf{C})]^T$ collectively gives the chemical reaction velocities. The vector of chemical species populations $\mathbf{C} = [[\mathrm{C}_1] \quad [\mathrm{C}_2] \quad \ldots \quad [\mathrm{C}_\mathrm{m}]]^T$ describes the current state of the chemical system.

The product of a stoichiometric coefficient with a chemical reaction velocity is the rate of change of that chemical species concentration that that chemical reaction induces. For the chemical species C_i, the rate of change of the concentration $[\mathrm{C_i}]$ that the chemical reaction with index j induces is $s_{ij}v_j$. Summing up the influxes and effluxes of a

chemical species across all of the chemical reactions gives a formula for the total rate of change of the concentration of that chemical species induced by the biochemical reaction network. The chemical reactions that make up the biochemical reaction network induce a net change in the population of chemical species according to the equation

$$\frac{d\mathbf{C}}{dt} = \mathcal{S}\mathbf{v}(\mathbf{C}) = \begin{bmatrix} s_{11} & s_{12} & \cdots & s_{1n} \\ s_{21} & s_{22} & \cdots & s_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m1} & s_{m2} & \cdots & s_{mn} \end{bmatrix} \begin{bmatrix} v_1(\mathbf{C}) \\ v_2(\mathbf{C}) \\ \vdots \\ v_n(\mathbf{C}) \end{bmatrix} = \sum_{j=1}^{n} \begin{bmatrix} s_{1j}v_j(\mathbf{C}) \\ s_{2j}v_j(\mathbf{C}) \\ \vdots \\ s_{nj}v_j(\mathbf{C}) \end{bmatrix}. \tag{2.14}$$

As an example, the simple biochemical reaction network that Table 2.1 describes has four chemical species and five chemical reactions. Although the system has only four written chemical reaction equations, there are actually five chemical reactions because one of the chemical reaction equations is reversible. The stoichiometry matrix of this biochemical reaction network is

$$\mathcal{S} = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 0 \\ 0 & 0 & 0 & 3 & -1 \end{bmatrix}, \tag{2.15}$$

and the chemical reaction velocity vector is $\mathbf{v} = \begin{bmatrix} k_a[\mathrm{A}] & k_b[\mathrm{B}] & k_c & k[\mathrm{C}]^2 & k_d[\mathrm{D}] \end{bmatrix}^T$. Expanding the product $\mathcal{S}\mathbf{v}$ according to Equation 2.14 leads to the system of ordinary differential equations for the chemical reactions of the simple biochemical reaction network

$$\frac{d[\mathrm{A}]}{dt} = k_b[\mathrm{B}] - k_a[\mathrm{A}], \tag{2.16}$$

$$\frac{d[\mathrm{B}]}{dt} = k_a[\mathrm{A}] - k_b[\mathrm{B}], \tag{2.17}$$

$$\frac{d[\mathrm{C}]}{dt} = k_c - 2k[\mathrm{C}]^2,$$

$$\frac{d[\mathrm{D}]}{dt} = 3k[\mathrm{C}]^2 - k_d[\mathrm{D}].$$

2.4.2 Detecting Conservation Relations in a Biochemical Reaction Network

An interesting observation of the system of differential equations that corresponds to the biochemical reaction network in Table 2.1 is that Equations 2.16 and 2.17 sum to zero. Looking at the chemical reaction equations, it is obvious why this should occur: chemical species A converts solely to chemical species B with a stoichiometry of one, and in reverse, chemical species B converts solely to chemical species A with a stoichiometry of one. Conservation of mass states that the sum of the quantity of chemical species A and the quantity of chemical species B must remain constant. This means that the sum of the quantities of chemical species A and chemical species B is $[\mathrm{A}] + [\mathrm{B}] = T$ for some constant T. Therefore,

$$\frac{d}{dt}[\mathrm{A}] + \frac{d}{dt}[\mathrm{B}] = \frac{d}{dt}T = 0.$$

The rate of change of the quantity of chemical species A must always exactly cancel out the rate of change of the quantity of chemical species B, hence $d[\mathrm{A}]/dt = -d[\mathrm{B}]/dt$ as seen earlier.

A conserved moiety is a collection of chemical species that convert from one form to another but whose total amount never changes. The sum of the quantities of the chemical species that make up a conserved moiety is a constant. Chemical species A and B form a conserved moiety in the biochemical reaction network in Table 2.1. There are reactions in the model that convert between these two chemical species, but there are no chemical reactions that either create or destroy chemical species A or B. A chemical species that phosphorylates and dephosphorylates, but never synthesizes nor degrades, is a common biological example of a conserved moiety. Two chemical species, say species S and $\mathrm{S^P}$, that differ only due to a phosphorylation state form a conserved moiety where $[\mathrm{S}] + [\mathrm{S^P}] = S_T$.

The sum $[\mathrm{A}] + [\mathrm{B}] = T$ is called a conservation relation. Note that $[\mathrm{B}] = T - [\mathrm{A}]$ can replace Equation 2.17 to produce a system of ordinary first-order differential-algebraic equations that has the same solution as the original system of differential equations. Replacing differential equations with conservation relations is often advantageous for the analysis and simulation of a biochemical reaction network. Although conserved moieties are not the sole source of conservation relations [124], many of the conservation relations in this dissertation derive from conserved moieties, and this dissertation frequently uses the term 'conservation relation' in the sense of a moiety conservation relation.

Analysis of the stoichiometry matrix

Examining the stoichiometry matrix in Equation 2.15 reveals another interesting observation. The rows of the stoichiometry matrix for chemical species A and B are linearly dependent. In fact, rows of the stoichiometry matrix are linearly dependent if and only if they represent a conservation relation [69]. Therefore, the conservation relations for a biochemical reaction network are computable from the stoichiometry matrix. The system of differential-algebraic equations for the biochemical reaction network when every possible differential equation is replaced with a conservation relation has $\mathrm{rank}(\mathcal{S})$ differential equations and $m - \mathrm{rank}(\mathcal{S})$ conservation relations.

Several methods exist for computing the conservation relations of a biochemical reaction network. A simple and generally efficient method for computing conservation relations is to apply Gauss-Jordan elimination to row-reduce the stoichiometry matrix. Gauss-Jordan elimination separates out the linearly dependent rows of the stoichiometry matrix, replacing those rows with zeros. Examining the elementary matrices that the elimination process used determines a set of conservation relations. Note that the set of conservation relations that an elimination process produces is not unique. Applying different elementary row operations can lead to different, but equally valid, conservation relationships.

An example applying Gauss-Jordan elimination to the stoichiometry matrix given by Equation 2.15 follows.

1. Augment the stoichiometry matrix given by Equation 2.15 with the identity matrix I_m. Mentally label the rows of the augmented matrix with the chemical species that each row represents, in this case A, B, C, and D.

$$\left[\begin{array}{ccccc|cccc} -1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 3 & -1 & 0 & 0 & 0 & 1 \end{array}\right] \begin{array}{c} \mathrm{A} \\ \mathrm{B} \\ \mathrm{C} \\ \mathrm{D} \end{array}$$

2. Perform Gauss-Jordan elimination to produce the reduced matrix.

$$\left[\begin{array}{ccccc|cccc} 1 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & -1/3 & 0 & 0 & 0 & 1/3 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{array}\right] \begin{array}{c} \mathrm{A} \\ \mathrm{C} \\ \mathrm{D} \\ \mathrm{B} \end{array} \tag{2.18}$$

3. The left-hand side of the reduced matrix has a single row that contains only zero entries, indicating that the system contains one conservation relation. Multiply the corresponding row in the right-hand side of the reduced matrix by the vector of chemical species populations to form the expression for that conservation relation,

$$\begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} [\mathrm{A}] \\ [\mathrm{B}] \\ [\mathrm{C}] \\ [\mathrm{D}] \end{bmatrix} = [\mathrm{A}] + [\mathrm{B}].$$

Thus, the model expresses the conservation relation $[\mathrm{A}] + [\mathrm{B}] = T$. Note that after performing Gauss-Jordan elimination, the conservation relation row in Equation 2.18 is the one with the label 'B'. Therefore, the conservation relation replaces the differential equation for the chemical species B. Cornish-Bowden and Hofmeyr [44] give another example of using Gauss-Jordan elimination to compute the conservation relations for a simple biochemical reaction

network. Sauro and Ingalls [124] give an overview of several methods for calculating conservation relations along with consideration of related computational issues.

2.5 Approximations for Differential Equation Models

The process that Section 2.4 describes for creating a system of ordinary first-order differential equations requires several physical assumptions. The physical assumptions made in support of using a system of ordinary differential equations to describe a biochemical reaction network include assumptions about

- the thermodynamic environment in which the chemical reactions take place,
- the structure of the biochemical reaction network,
- the relative proportion of reactive and non-reactive molecular collisions,
- and the distribution of chemical species within the volume of the chemical system.

Making physical assumptions to construct an approximation of the real biochemical reaction network is a reasonable approach. The goal of modeling is to gain insight by simulating the biochemical reaction network, rather than emulating the biochemical reaction network. Physical assumptions reduce the time that simulation requires and thereby make tractable the study of large-scale biochemical reaction networks.

Two significant physical assumptions pervade all of the solution methods that this dissertation discusses. The first physical assumption deals with the environment of the cell. The second physical assumption deals with the chemical reactions that occur within the cell.

The environmental assumption

The first assumption is that chemical reactions take place inside of a fixed volume with uniform and constant temperature and pressure. Thermodynamical statistics, such as the temperature of the surrounding environment, influence the chemical reaction velocities. On a microscopic scale, physical forces and the constraints specified by thermodynamics control the chemical reactions. Simulations at the mesoscopic molecular protein level subsume the effects of physical forces into the rate constants. The modeler encodes the effect that physics and thermodynamics have on a chemical reaction by permitting only certain chemical reactions to occur and specifying the chemical reaction velocity.

Molecular statistics, such as the chemical reaction velocity, approximate the thermodynamics of the chemical reaction and the environment of the chemical system. A chemical reaction that is thermodynamically favorable either releases energy in the form of heat, or increases the entropy of the chemical system. Berg, Tymoczko, and Stryer [29] give the details of computing the free energy change of a chemical reaction to determine whether the chemical reaction is thermodynamically favorable and can therefore occur spontaneously. Chemical reaction velocities act as fixed formulas because of the assumption that the physical and thermodynamic descriptions of the chemical system are essentially constant over time.

The chemical reaction assumption

The second assumption is that all chemical reactions are instantaneous physical events. Section 2.1 defined the kinetic formula and stoichiometry for a chemical reaction liberally. There are no restrictions against having chemical reactions with non-integer stoichiometries nor are there restrictions against having chemical reactions that involve the combination of many chemical products. Using this definition of a chemical reaction, the class of permissible chemical reactions includes $1.3\mathrm{A} \longrightarrow \mathrm{B}$ and $\mathrm{A} + \mathrm{B} + \mathrm{C} \longrightarrow \mathrm{D}$. However, there is no physical analog to a chemical reaction that involves 1.3 molecules of chemical species A. Similarly, basic physics says that the simultaneous collision of a molecule of A with a molecule of B and a molecule of C is so improbable that such collisions are neglectable.

Chemical reactions that correspond to a fundamental, instantaneous physical event describe

- the synthesis of a single molecule of a chemical species,

- the conversion of a single molecule of a chemical species to another chemical species,
- and the collision of two molecules to produce chemical products.

An elementary chemical reaction is a chemical reaction that models one of these fundamental physical events.

Non-elementary chemical reactions, such as those that use Michaelis-Menten kinetics, empirically approximate a series of elementary chemical reaction steps. There is no unique way to 'unpack' an overall chemical reaction into elementary chemical reactions. In this dissertation, solution schemes that require elementary chemical reactions assume that the modeler previously unpacked the non-elementary chemical reactions in the model.

2.5.1 Discrete Approximation Schemes

Using just the two assumptions that the previous section gives about the environment of the chemical system and the structure of the chemical reactions, the only possible simulation scheme is to precisely simulate the molecular dynamics of the chemical system. A molecular dynamics simulation models the chemical system by continuously tracking the position and velocity of each molecule. The computer evolves the chemical system by updating the position of each molecule according to its velocity and treating the collision of two molecules as a discrete event.

When two molecules collide, the collision is either reactive or non-reactive. In the case of a non-reactive collision, the two molecules rebound away from one another. A non-reactive collision changes the spatial distribution of chemical species in the chemical system. In the case of a reactive collision, a chemical reaction event occurs, and one of the elementary chemical reactions modifies the population of chemical species in the chemical system.

Molecular dynamics describes the population and spatial distribution of chemical species in the chemical system as a function of time. Applying molecular dynamics precisely simulates the chemical system according to Newtonian physics, ignoring effects such as quantum mechanics. However, performing a molecular dynamics simulation is too slow to practicably apply to interesting biological problems. Such models may have many thousands of molecules of a chemical species that collide and interact between each interesting chemical reaction event.

Well-stirred chemical systems and the stochastic chemical master equation

The observation that chemical species rarely have reactive collisions is the basis of a simplifying approximation for molecular dynamics. Assume that for every reactive collision, a very large number of non-reactive collisions occur in the meantime. Non-reactive collisions change the spatial distribution of chemical species in the chemical system. After a large number of non-reactive collisions, the spatial distribution of a chemical species is essentially random.

The well-stirred assumption approximates molecular dynamics by positing that the molecules of a chemical species continuously and evenly mix and redistribute themselves throughout the volume of the chemical system. A well-stirred chemical system is spatially homogeneous. Using the well-stirred assumption, it is no longer necessary to record the spatial distribution of chemical species or to simulate any of the non-reactive collisions. Instead, the population of chemical species with discrete integer variables $|\mathrm{C}_1|$, $|\mathrm{C}_2|$, …, $|\mathrm{C}_\mathrm{m}|$ or, in vector form, $\mathbf{C}$, entirely describes the state of the chemical system. Unlike previous chemical species variables, which were concentrations, the chemical species variables for stochastic equations are usually in terms of quantity of substance. The volume of the chemical system relates the concentration of a chemical species and the quantity of substance of that chemical species.

The chemical reaction velocity vector $\mathbf{v}(\mathbf{C}) = \begin{bmatrix} v_1(\mathbf{C}) & v_2(\mathbf{C}) & \dots & v_n(\mathbf{C}) \end{bmatrix}^T$ for the biochemical reaction network gives each chemical reaction velocity as a function of the population of chemical species in the chemical system. The probability that the chemical reaction with index j occurs over the next short interval of time dt is $v_j(\mathbf{C})dt$. The probability that the chemical system has a population of chemical species $\mathbf{C}$ at time t, given that the initial conditions of the chemical system are a population of chemical species $\mathbf{C}_0 = \mathbf{C}(t_0)$ at time t_0, is written as $\Pr(\mathbf{C}; t \mid \mathbf{C}_0; t_0)$ for $t \geq t_0$, or more simply as $\Pr(\mathbf{C}; t)$.

The stochastic chemical master equation [60, 92] gives the change over time of the population of chemical species in the chemical system as the partial differential equation

$$\frac{\partial \Pr(\mathbf{C}; t)}{\partial t} = \sum_{j=1}^{n} v_j(\mathbf{C} - \mathcal{S}_j) \Pr(\mathbf{C} - \mathcal{S}_j; t) - \sum_{j=1}^{n} v_j(\mathbf{C}) \Pr(\mathbf{C}; t), \tag{2.19}$$

where $\mathcal{S}_j$ is the column of the stoichiometry matrix that corresponds to the chemical reaction with index j. Equation 2.19 contains two sums over probabilities. The first sum is the probability of the chemical system transitioning into the desired state from one of the nearby states according to the chemical reactions in the chemical system. The second sum is the probability of the chemical system transitioning out of the desired state.

Gillespie's stochastic simulation algorithm

Gillespie proposed a stochastic simulation algorithm [59] that uses the well-stirred assumption and can compute a probability given by the stochastic chemical master equation to any desired degree of accuracy. Each invocation of Gillespie's stochastic simulation algorithm produces a sample path $\mathbf{C}(t)$ that describes the evolution of the population of chemical species in the chemical system over time. Repeating Gillespie's stochastic simulation algorithm many times, and treating the obtained sample paths as observations from a sample space, approximates the statistics of the stochastic chemical reaction equation.

Gillespie's stochastic simulation algorithm is an iterative process that simulates the reactive collisions between chemical species. The steps taken by the algorithm trace out successive populations of chemical species to describe the sample path $\mathbf{C}(t)$.

1. Start at the known point $\mathbf{C}_0$ at the time t_0,
2. compute the total chemical reaction velocity, $v_T = \sum_{j=1}^{n} v_j(\mathbf{C}_0)$, among the chemical reactions in the biochemical reaction network,
3. select random variates α and β uniformly from the interval $(0, 1)$,
4. find the smallest integer j^* such that $\sum_{j=1}^{j^*} v_j(\mathbf{C}_0) > \alpha v_T$,
5. advance the population of chemical species in the chemical system to $\mathbf{C}_1 = \mathbf{C}_0 + \mathcal{S}_{j^*}$,
6. advance the chemical system time to $t_1 = t_0 + (\ln \beta^{-1})/v_T$,
7. and repeat this process to produce successive steps $\mathbf{C}_2, \mathbf{C}_3, \ldots, \mathbf{C}_r$ at times $t_2, t_3, \ldots, t_r$, respectively.

The algorithm terminates when $t_r > t_{\max}$. The state of the chemical system at time $t_{\max}$ is then definitely $\mathbf{C}_{r-1}$ as no chemical reaction event occurred between the times t_{r-1} and t_r.

Gibson and Bruck later developed a version of Gillespie's stochastic simulation algorithm that is more difficult to implement but computationally more efficient [58]. However, the stochastic simulation algorithm remains too slow for many interesting biological problem. Gillespie's stochastic simulation algorithm requires sequentially simulating every chemical reaction event. As the total size of the population of chemical species in the chemical system increases, the expected step size taken after each chemical reaction event decreases and the expected total execution time of the simulation increases.

2.5.2 Continuous Approximation Schemes

An alternative to trying to speed up computation of the discrete stochastic process that the stochastic chemical master equation defines is to construct a new approximation scheme that can jump past multiple chemical reaction events in a single time step. Suppose that the population of chemical species is sufficiently large so that the effect of chemical reaction events over a small time scale, Δt, does not appreciably change the chemical reaction velocity vector. In other words, $\mathbf{v}(\tilde{\mathbf{C}}) \approx \mathbf{v}(\mathbf{C})$ when $\tilde{\mathbf{C}}$ is a population of chemical species that can arise from the population of chemical species $\mathbf{C}$ within a short period of time Δt. Furthermore, suppose that at any time $t \in [t_0, t_{\max}]$, the modeler expects each chemical reaction in the biochemical reaction network to occur many times in the interval from t to $t + \Delta t$. This requires that every chemical species exists in at least moderate amounts.

If these two assumptions about Δt hold for the chemical system, then Gillespie concludes that the discrete stochastic process that the stochastic chemical master equation defines is approximatable using a continuous stochastic process [61]. This leads to the chemical Langevin equation, which is the stochastic differential equation

$$\frac{d\mathbf{C}}{dt} = \mathcal{S}\mathbf{v}(\mathbf{C}) + \mathcal{S}\begin{bmatrix} \sqrt{v_1(\mathbf{C})}G_1(t) \\ \sqrt{v_2(\mathbf{C})}G_2(t) \\ \vdots \\ \sqrt{v_n(\mathbf{C})}G_n(t) \end{bmatrix} \tag{2.20}$$

where the $G_1(t), G_2(t), \ldots, G_n(t)$ are all independent Gaussian variables with mean zero and standard deviation one. Note that in the chemical Langevin equation approximation, the chemical species populations $|\mathrm{C}_1|$, $|\mathrm{C}_2|$, …, $|\mathrm{C}_\mathrm{m}|$ are now continuous real variables instead of discrete integer variables. The chemical Langevin equation is a continuous stochastic process analogous to the stochastic chemical master equation (Equation 2.19).

Research continues in the attempt to speed up stochastic simulation schemes. Tau leaping [116] is a method that Gillespie and Petzold developed for use when all of the chemical species populations are of at least moderate size. At this scale size, stochastic effects still influence the evolution of the chemical system, but purely stochastic methods are much too slow. The tau-leaping method approximates the stochastic solution by advancing the chemical system solution over intervals of time, τ, in which the number and type of chemical reaction events is known. However, the tau-leaping method does not determine the order of the chemical reaction events within a τ interval.

Ordinary differential equations in the thermodynamic limit

Although a computer can evaluate the chemical Langevin equation much faster than the stochastic chemical master equation, computing the stochastic noise term is still expensive in comparison to computing the deterministic term. There are also theoretical difficulties that the modeler must avoid when using the chemical Langevin equation approximation. The first difficulty is that the conditions Gillespie places on Δt may not apply to the chemical system under study, and these conditions are difficult to check. The second difficulty is that, as Section 2.3.2 mentions, the modeler must choose an interpretation of the stochastic integral.

A new assumption further simplifies the continuous approximation to the stochastic chemical master equation. That assumption, the 'thermodynamic limit', is that the chemical system is operating in the limit where the volume containing the chemical system goes to infinity, the population of chemical species $|\mathrm{C}_1|$, $|\mathrm{C}_2|$, …, $|\mathrm{C}_\mathrm{m}|$ all go to infinity, and the concentrations of the chemical species remain constant. In the thermodynamic limit, the deterministic term of the chemical Langevin equation grows with the size of the chemical system. The stochastic noise term of the chemical Langevin equation grows with the square root of the size of the chemical system.

As the population of chemical species grows larger, the stochastic noise term becomes negligible in comparison to the deterministic term. Therefore, the chemical Langevin equation becomes in the thermodynamic limit

$$\frac{d\mathbf{C}}{dt} = \mathcal{S}\mathbf{v}(\mathbf{C}) + \mathcal{S}\begin{bmatrix} \sqrt{v_1(\mathbf{C})}G_1(t) \\ \sqrt{v_2(\mathbf{C})}G_2(t) \\ \vdots \\ \sqrt{v_n(\mathbf{C})}G_n(t) \end{bmatrix} \approx \mathcal{S}\mathbf{v}(\mathbf{C}), \qquad \text{as } |\mathrm{C}_1|, |\mathrm{C}_2|, \ldots, |\mathrm{C}_\mathrm{m}| \to \infty.$$

This is exactly Equation 2.14 for constructing a system of differential equations from a biochemical reaction network.

Models typically require hundreds of molecules of each chemical species in the chemical system before the modeler can employ the thermodynamic limit. Since this approximation discards the stochastic noise term of the chemical Langevin equation, the resulting system of ordinary differential equations is a deterministic process.

Chapter 3

Modeling Processes and Methodologies

A modeling process is a grouped, repeatable collection of operations that modelers perform during the task of building a model. A modeling process is itself a model that describes the work done by modelers. Modeling processes are a means by which modelers formalize a natural system to produce mathematical systems and a means by which modelers interpret mathematical systems to derive information about a natural system. Modeling methodologies often guide a modeling and simulation specialist during the design of a new modeling process. A modeling methodology gives a repeatable series of steps that modeling processes can incorporate and is intended to improve specific attributes for a modeling process, such as efficiency, reliability, testability, and predictability.

The objectives of building a modeling process are to

- understand the methods that biological modelers employ to build models,
- and develop ideas that allow biological modelers to build reliable models more efficiently and predictably.

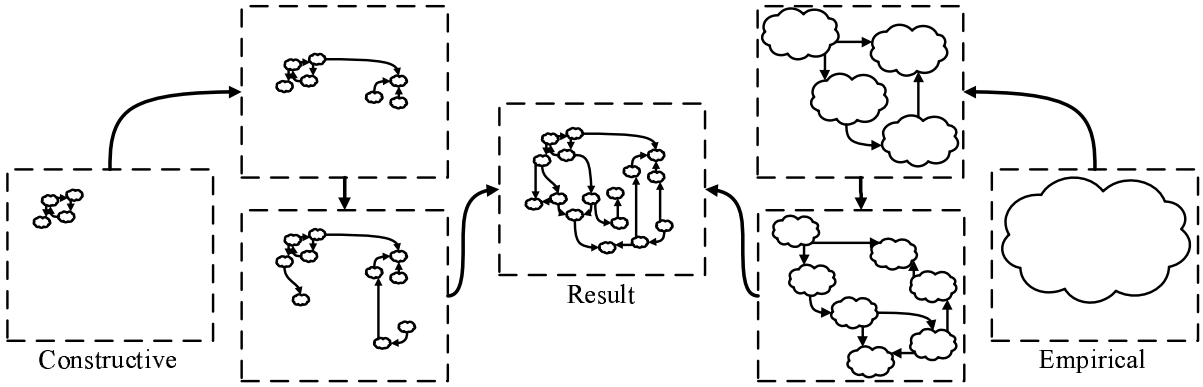
It is not necessary to laboriously detail every task that a modeler might undertake during model development. Many modelers have unique practices that they employ while working.

A modeling process is not a complete recipe for solving a problem. Building a model is a difficult task, and modelers must overcome many unforeseen obstacles before they can successfully develop a new model. A modeling process cannot substitute for experience, intelligence, or good taste for designing models. A modeling process must be parsimonious enough so that a biological modeler can understand the modeling process and integrate new techniques without expending excessive effort. However, the modeling process must also be complete enough so that the modeler can apply the modeling process to their work. This chapter describes

- a modeling process for constructing biological models that comes directly from observations of modelers building and testing models,
- a modeling methodology that supports several key attributes for performance and quality in the construction of biological models,
- and a new modeling process for constructing biological models that incorporates the modeling methodology and observed modeling process, and proposes a reorganization of certain model development tasks.

Asking biological modelers to describe how they work, many can relate themselves to using one of two kinds of modeling processes for building models of biochemical reaction networks. The ‘constructive’ style of modeling process starts with elementary chemical reaction steps, to which the modelers add increasingly larger aggregates of chemical reactions to better explain a biological process. The ‘empirical’ style of modeling process starts with a sketch of the overall biological process, and modelers then attempt to continuously refine that sketch by replacing a generalized step with more detailed components. Figure 3.1 shows the difference in model evolution between using constructive and empirical modeling processes. This dissertation focuses on constructive modeling processes.

Figure 3.1: Difference in model evolution between using constructive and empirical modeling processes.



However, modelers employ both constructive and empirical modeling processes, and no one has shown that one type of modeling process is conclusively better for biological modeling than the other.

There are two nested loops in a constructive modeling process. Table 3.1 gives a constructive modeling process that is typical of a 'bottom-up' approach. The outer loop expands the aggregate of chemical reactions. The inner loop refines and tweaks the biochemical reaction network to evaluate the performance of the model and to make subtle adjustments. Modelers can use early models from a constructive modeling process for making predictions, even though these models explain only a subset of the observations related to the biological process. A constructive modeling process terminates when the modeler is satisfied that the developed model explains all of the observations that the modeler identified as important.

Table 3.1: Outline of a constructive model building process for producing biological models.

```
Identify the distinguishing features of the overall chemical reaction.
Research the literature for experimental data on relevant chemical species and reactions.
Assemble a small, proposed model using elementary chemical reaction steps.
(Loop)
    (Loop)
        Assign rate laws and kinetic constants based on experimental measurements and intuition.
        Convert the proposed model into a computer solvable system.
        Apply numerical methods to simulate the system.
        Compare the numerical solution to the experimental data.
    (End Loop)
    Research the literature for experimental data about related chemical species and reactions.
    Add additional elementary chemical reactions and intermediate chemical species.
(End Loop)
```

An empirical modeling process also has two nested loops. Table 3.2 shows a typical 'top-down' modeling process. The outer loop defines a boundary for the search space in which the modeler attempts to locate a more refined model of the biological process. The inner loop examines candidate refinements for the model and selects between the possible refinements. Early models from an empirical modeling process explain the initial observations related to the biological process but have little predictive power. An empirical modeling process terminates when the justification for the individual component pieces of the developed model satisfies the modeler.

This chapter examines two modeling processes for building biological models. Later chapters employ these modeling processes for building biological modeling software and attempt to measure the effectiveness of the modelers that use each process. Section 3.1 is an observational account of an existing modeling process for building biological models. Then, Section 3.2 introduces a general modeling process methodology and considers what this methodol-

Table 3.2: Outline of an empirical model building process for producing biological models.

Identify the distinguishing features of the overall chemical reaction.
Research the literature for experimental data on relevant chemical species and reactions.
Assemble a proposed model covering the overall chemical reaction.
Establish experimental measurements of the inputs and outputs of the proposed model.
(Loop)
    Compile a list of plausible chemical species that may act as intermediates.
    (Loop)
        Generate a more detailed proposed model involving the intermediate chemical species.
        Assign chemical reaction kinetics using empirical laws.
        Convert the proposed model into a computer solvable system.
        Apply numerical methods to simulate the system.
        Compare the numerical solution to the experimental data.
    (End Loop)
    Research the literature for experimental data about related chemical species and reactions.
    Replace a step in the model with additional chemical reactions and intermediate chemical species.
(End Loop)

ogy can provide to assist biological modelers. Finally, Section 3.3 proposes a new modeling process for biological modeling. The remaining chapters of this dissertation study implementations of this new modeling process.

Contents

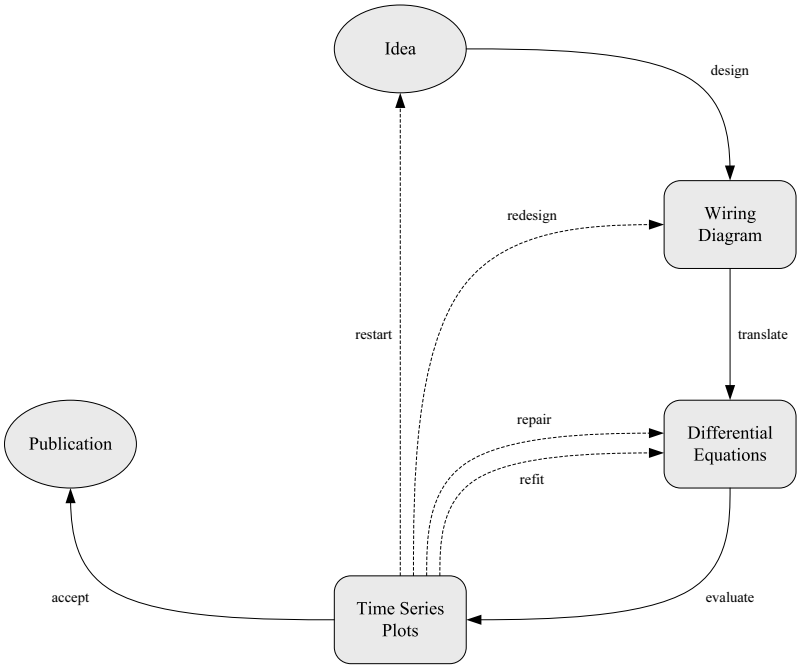
3.1 Original Modeling Process

Figure 3.2 shows a model of the process that modelers in the Tyson laboratory use to develop models of biochemical reaction networks, including Chen et al. [41], Chen et al. [42], Novak et al. [106], and Tyson and Novak [132]. This dissertation refers to the Tyson modeling process as the 'original modeling process.' The original modeling process evolved over more than ten years of practice developing models rather than using modeling process formalisms.

The original modeling process does not currently have written documentation for novice modelers. However, the original modeling process presently changes very little over time so producing such documentation is feasible. Novice modelers learn about the original modeling process primarily through demonstration and mentoring. A novice modeler typically spends several months building models before becoming proficient enough with the original modeling process to attempt original work. The original modeling process is a constructive modeling process.

In the ensuing text, the symbol $\xrightarrow{x}$ denotes the stage of the original modeling process labeled with the symbol 'x' in the diagram, and such symbols mark the text where that stages is discussed. A stage drawn with a solid line in the diagram indicates the successful completion of a process and the text refers to such a process as a 'primary' stage. A stage drawn with a dashed line in the diagram indicates an error recovery activity and the text refers to such a process as an 'error-recovery' or 'secondary' stage. Finally, the name of the **x** stage given in emphasized text distinguishes the name of a stage from the name of a generic activity or process.

Figure 3.2: Original modeling process observed in the Tyson laboratory.



The modelers in the Tyson laboratory record biochemical reaction networks using sketched diagrams and directly convert the sketch of the biochemical reaction network to a system of differential-algebraic equations by hand, using the algorithms that Section 2.4 described. This original modeling process partially standardizes the graphical elements in a sketch of a biochemical reaction network. For example, connecting two chemical species using a solid line indicates the transfer of mass via a chemical reaction. Connecting a chemical species with a chemical reaction using a dashed line indicates an influence on that chemical reaction by the chemical species, such as participating as a catalyst. However, like the original modeling process, modelers communicate the standard for using graphical elements in a sketch of a biochemical reaction network orally rather than through written documentation.

Until recently, the modelers of the Tyson laboratory primarily used off-the-shelf tools for solving and analyzing the systems of differential equations that they generated from biochemical reaction networks. Specialized tools for pathway modeling were not available. They typically constructed the differential equation specification for a model, including the parameters, initial conditions, and simulator control settings, using the ODE file format that G. Bard Ermentrout developed for the XPPAUT integrator [51]. Ermentrout documented the ODE file format but did not standardize the format. User documentation and tutorials are available in the XPPAUT manual [51] and online at [52]. This documentation does not provide enough information for someone to exactly duplicate the parsing and interpretation of ODE files that the XPPAUT program performs. Therefore, some experimentation is necessary to interpret the syntax and semantics of an ODE file and recover the originally intended model specification.

When modelers prepare a model for publication, they sketch the biochemical reaction network using a graphical figure, the differential and algebraic equations as mathematical formulas, and the numerical values of parameters and initial conditions as text in tables. In many cases, a specification of the simulator and simulator control settings does not accompany the published model. Instead, the modeler must laboriously duplicate the method by which the simulator interprets the system of differential-algebraic equations. By performing simulation runs and comparing the time series output that the simulator produces with figures of time series plots given in the publication, the modeler can infer the necessary changes to the simulator control settings.

Before modelers can begin working on a model, they first must identify the problem that the model will solve. This process, known as ‘problem formulation’, includes an analysis of requirements, an identification of a solution method, and a specification of modeling objectives [19]. Unless the modeler starts from a formulated problem, there is a risk

of inadequately solving the problem or solving the wrong problem. Preferably, the initial modelers record the output of problem formulation in some lasting form, such as a scientific publication, so that future modelers can refer to the modeling requirements and objectives while working on the model.

The original modeling process does not include problem formulation as an explicit step. During the construction of the original modeling process, there was insufficient observation of this particular group of modelers formulating completely new problems in the sense of wishing to develop a model for a new organism or to apply new solution techniques to a previously developed model. Instead, they expanded existing models by attempting to match additional experimental observations. It is unknown if this infrequent reformulation is an inherent property of the problems these modelers are attempting to solve or is a side effect of the modeling process that they are using. Observations with different modelers and modeling processes might answer this question in the future.

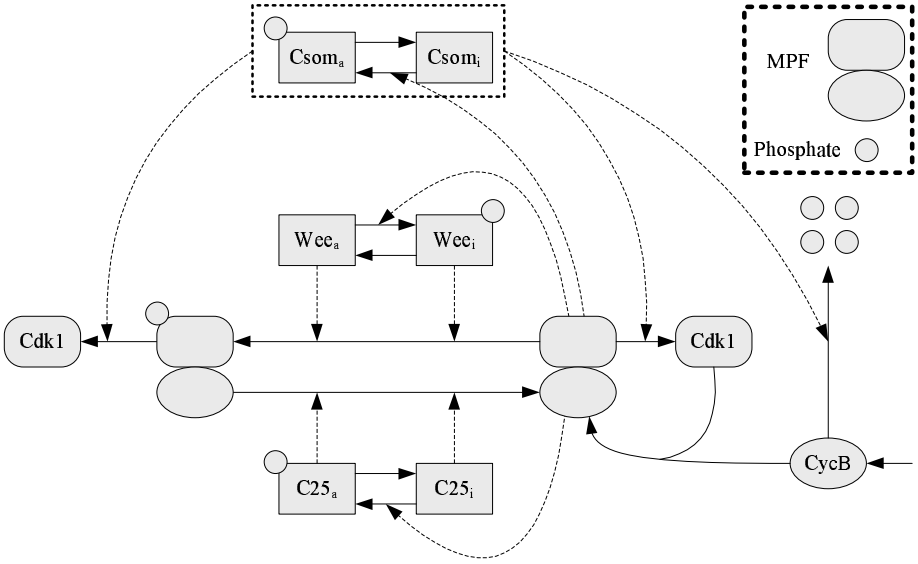
3.1.1 Primary Stages of the Original Modeling Process

The original modeling process has four primary stages: **design**, **translate**, **evaluate**, and **accept**. The modeler creates the model in the **design** stage and converts the model to a computer understandable format in the **translate** stage. During the **evaluate** stage, the modeler tests the model to determine its performance with respect to the modeling objectives. Finally, the **accept** stage results when the modelers produce a presentable model from the information that they recorded during model development. Along with these primary stages, there are additional error recovery stages that a modeler performs after detecting an error during the **evaluate** stage.

$\xrightarrow{\textbf{design}}$

The **design** stage typically begins with the modeler creating a wiring diagram from an idea of how a biological process occurs. A wiring diagram depicts the biochemical reaction network of the proposed model. As Section 2.4 described, the format of a wiring diagram is a graph that captures the chemical reactants and products at the vertices and represents chemical reactions that create, destroy, and convert these chemical species using labeled, directed edges. Additionally, the wiring diagram may note the kinetic information for a chemical reaction. Figure 3.3 is a wiring diagram that depicts the post-translational modification of cyclin, an important regulation process, in *Xenopus laevis* extracts based on the model by Marlovits, Tyson, Novak, and Tyson [87].

Figure 3.3: Wiring diagram of a regulatory process in *Xenopus laevis* extracts.



Although there are wiring diagram styles that the modelers in the Tyson laboratory share, the larger modeling community has not standardized the notation for wiring diagrams. Modelers often invent ad hoc notation to express

abstractions, replication, and unusual processes. For example, modelers frequently abstract a complicated process involving several steps that appears multiple times in the wiring diagram. The wiring diagram notation of Kohn [80] employs abstraction to represent concisely the interaction of multiple chemical species.

Removing details from the wiring diagram simplifies the presentation of the model and saves space during publication. In the publication text, the modeler describes the excised process, gives a notation that stands in place of that process in the wiring diagram, and lists the chemical species that each instance of the process uses. However, splitting the presentation of the model between the wiring diagram and text makes accurately duplicating the results of the model harder. The reader of the publication must correctly reassemble the model from the split presentation.

Figure 3.3 is simple enough that communicating the essential features of this model does not require much abstraction. Still, the figure includes a key that links particular graphical shapes with named chemical species to reinforce visually the idea that chemical reactions assemble and disassociate these molecular complexes. In addition, the wiring diagram uses the special notation of drawing four circles to indicate degradation of the chemical species CycB and the notation of drawing a single, attached circle to indicate the phosphorylation state of a chemical species.

Chemical reaction kinetics frequently have a presentation that is separate from the wiring diagram. In some cases, the publication omits the chemical reaction kinetics entirely, and the reader must infer the kinetic information from other figures. Without the full details of the chemical reaction kinetics, it is possible to structurally analyze the model but not to perform simulation. Since the wiring diagram often lacks some details of the chemical reaction kinetics, it is typical for modelers to first rewrite the model as a collection of chemical reaction equations, along with the appropriate chemical reaction kinetics. The **design** stage terminates after the modeler assigns chemical reaction kinetics to all of the chemical reaction equations. Although Figure 3.3 identifies the transfer of mass of chemical species from chemical reactions and regulatory signals, the figure gives no indication as to how or at what rate the chemical reactions occur. Table 3.3 explains the chemical reaction kinetics as an adjunct.

Table 3.3: Chemical reaction equations and rate laws for the *Xenopus laevis* extracts model of Marlovits, Tyson, Novak, and Tyson.

Chemical reaction equation	Kinetic formula
$\mathrm{Csom_a} \rightleftharpoons \mathrm{Csom_i}$	$\frac{k_{cyr}[\mathrm{Csom_a}]}{J_{cyr}+[\mathrm{Csom_a}]}, \frac{k_{cyf}([\mathrm{MPF_a}]+\epsilon_3)[\mathrm{Csom_i}]}{J_{cyf}+[\mathrm{Csom_i}]}$
$\mathrm{Wee_a} \rightleftharpoons \mathrm{Wee_i}$	$\frac{k_{Weef}([\mathrm{MPF_a}]+\epsilon_2)[\mathrm{Wee_a}]}{J_{Weef}+[\mathrm{Wee_a}]}, \frac{k_{Weer}[\mathrm{Wee_i}]}{J_{Weer}+[\mathrm{Wee_i}]}$
$\mathrm{C25_a} \rightleftharpoons \mathrm{C25_i}$	$\frac{k25_r[\mathrm{C25_a}]}{J25_r+[\mathrm{C25_a}]}, \frac{k25_f([\mathrm{MPF_a}]+\epsilon_1)[\mathrm{C25_i}]}{J25_f+[\mathrm{C25_i}]}$
$\bullet \xrightarrow{k_1} \mathrm{CycB}$	k_1
$\mathrm{CycB} \longrightarrow \bullet$	$(k_2'[\mathrm{Csom_i}]+k_2''[\mathrm{Csom_a}])[\mathrm{CycB}]$
$\mathrm{CycB}+\mathrm{Cdk1} \xrightarrow{k_3} \mathrm{MPF_a}$	$k_3[\mathrm{CycB}][\mathrm{Cdk1}]$
$\mathrm{MPF_a} \longrightarrow \mathrm{Cdk1}$	$(k_2'[\mathrm{Csom_i}]+k_2''[\mathrm{Csom_a}])[\mathrm{MPF_a}]$
$\mathrm{MPF_a} \rightleftharpoons \mathrm{MPF_i}$	$(k_w'[\mathrm{Wee_i}]+k_w''[\mathrm{Wee_a}])[\mathrm{MPF_a}], (k_c'[\mathrm{C25_i}]+k_c''[\mathrm{C25_a}])[\mathrm{MPF_i}]$
$\mathrm{MPF_i} \longrightarrow \mathrm{Cdk1}$	$(k_2'[\mathrm{Csom_i}]+k_2''[\mathrm{Csom_a}])[\mathrm{MPF_i}]$

$\xrightarrow{\textbf{translate}}$

The **translate** stage is the process of converting the chemical reaction equations written for the biochemical reaction network model to a computer understandable form, typically a system of ordinary differential-algebraic equations. For each chemical species in the model, the modeler creates an ordinary differential equation by using the algorithms that Section 2.4 described.

Modelers estimate parameter values for the rate laws using their intuition or knowledge about the chemical system. Frequently, the modeler does not know exact numeric values for these parameters. Instead, the modeler gives an

initial estimate for each of the parameter values and then continuously updates these guesses based on the feedback from model evaluation. The original modeling process includes the repeated adjustment of parameter values, which modelers often describe as 'parameter twiddling', as the **refit** error recovery stage. A modeler engages in parameter twiddling when simulation of the model fails to adequately reproduce the experimental observations.

In some cases, chemical reactions convert a chemical species between different forms but neither create nor destroy that chemical species. As Section 2.4.2 described, the total quantity of such a chemical species is conserved and an algebraic expression can replace the differential equation for one form of the chemical species.

When the model contains a conservation relation, the selection of which differential equation the conservation relation replaces is not relevant for numerical integration. Any choice of a chemical species for elimination leads to a consistent and mathematically equivalent representation for the model. Therefore, the choice of how to mathematically express the conservation relationship depends upon the aesthetic preferences of the modeler. Often, modelers prefer writing the conservation relation in a 'biologically suggestive' form, which requires writing the conservation relation so that all of the computed chemical species values and totals are non-negative.

Figure 3.3 corresponds to a system of six ordinary differential equations and four conservation relations.

$$\begin{aligned}
\frac{d[\mathrm{C25_a}]}{dt} &= \frac{k25_f([\mathrm{MPF_a}]+\epsilon_1)[\mathrm{C25_i}]}{J25_f+[\mathrm{C25_i}]} - \frac{k25_r[\mathrm{C25_a}]}{J25_r+[\mathrm{C25_a}]} \\
[\mathrm{C25_i}] &= 1.0 - [\mathrm{C25_a}] \qquad (3.1) \\
\frac{d[\mathrm{CycB}]}{dt} &= k_1 - [\mathrm{CycB}](k_2'[\mathrm{Csom_i}] + k_2''[\mathrm{Csom_a}] + k_3[\mathrm{Cdk1}]) \\
\frac{d[\mathrm{MPF_a}]}{dt} &= k_3[\mathrm{CycB}][\mathrm{Cdk1}] + (k_c'[\mathrm{C25_i}] + k_c''[\mathrm{C25_a}])[\mathrm{MPF_i}] - \\
&\quad (k_w'[\mathrm{Wee_i}] + k_w''[\mathrm{Wee_a}] + k_2'[\mathrm{Csom_i}] + k_2''[\mathrm{Csom_a}])[\mathrm{MPF_a}] \\
\frac{d[\mathrm{MPF_i}]}{dt} &= (k_w'[\mathrm{Wee_i}] + k_w''[\mathrm{Wee_a}])[\mathrm{MPF_a}] - \\
&\quad (k_c'[\mathrm{C25_i}] + k_c''[\mathrm{C25_a}] + k_2'[\mathrm{Csom_i}] + k_2''[\mathrm{Csom_a}])[\mathrm{MPF_i}] \\
[\mathrm{Cdk1}] &= 1.0 - [\mathrm{MPF_a}] - [\mathrm{MPF_i}] \qquad (3.2) \\
\frac{d[\mathrm{Wee_a}]}{dt} &= \frac{k[\mathrm{Wee}]_r[\mathrm{Wee_i}]}{J[\mathrm{Wee}]_r+[\mathrm{Wee_i}]} - \frac{k[\mathrm{Wee}]_f([\mathrm{MPF_a}]+\epsilon_2)[\mathrm{Wee_a}]}{J[\mathrm{Wee}]_f+[\mathrm{Wee_a}]} \\
[\mathrm{Wee_i}] &= 1.0 - [\mathrm{Wee_a}] \qquad (3.3) \\
\frac{d[\mathrm{Csom_a}]}{dt} &= \frac{k_{cyf}([\mathrm{MPF_a}]+\epsilon_3)[\mathrm{Csom_i}]}{J_{cyf}+[\mathrm{Csom_i}]} - \frac{k_{cyr}[\mathrm{Csom_a}]}{J_{cyr}+[\mathrm{Csom_a}]} \\
[\mathrm{Csom_i}] &= 1.0 - [\mathrm{Csom_a}] \qquad (3.4)
\end{aligned}$$

In this model, the chemical species $\mathrm{Cdc25}$ and $\mathrm{Wee1}$ convert between an active and inactive form via phosphorylation, but there are no chemical reactions for their synthesis or degradation. Similarly, the chemical species $\mathrm{Cdk1}$ is present as a free molecular species, bound as a complex in active MPF, and bound as a complex in inactive MPF, but no chemical reactions create or destroy $\mathrm{Cdk1}$. Phosphorylation and dephosphorylation convert MPF between its active and inactive forms. These chemical species have a conservation relation in the place of their differential equation (Equations 3.1, 3.2, 3.3, and 3.4).

In addition to having a continuous differential equation model, some models also include a discrete event model. A discrete event is an instantaneous change to the model that takes place in response to a trigger condition. Discrete events take place either immediately after meeting the trigger condition or after an event delay, which is some offset time into the future. Modelers describe certain cellular processes, such as cell division, with discrete events that set chemical species concentrations, change parameter values, alter chemical reaction kinetics, and switch between different systems of differential equations for the continuous model.

The addition of discrete events to a model makes the model significantly more difficult to simulate. Finding the exact time that a simulation meets the discrete event trigger condition requires the construction of an approximation

function to the differential equation solution and locating the roots of that function. Furthermore, many of the commonly used numerical integrators manipulate the integration history to predict the behavior of the differential equation solution better in the near future. The numerical integrator must destroy this recorded integration history when a discrete event takes place. The model that the numerical integrator constructs for the differential equation system is no longer valid after the chemical species concentrations or differential equations change.

$\xrightarrow{\textbf{evaluate}}$

The **evaluate** stage begins with the modeler generating time series plots of important chemical species concentrations from the model. A time series is the output of applying numerical integration to the system of differential-algebraic equations. Each step of a time series reports the concentrations of the chemical species in the model at a particular moment in time. Typically, a numerical integrator outputs time series steps at regular intervals, regardless of the actual integration steps required to simulate the model.

The modeler sets the parameters and initial conditions of the model to match the conditions under which an experimentalist conducted an experiment. For example, the *Xenopus laevis* extract model has several experimental observations that divide the initial concentrations of the chemical species MPF among its various, related forms. The modeler must compare the time series plots with the experimental observations and then judge whether the model adequately represents the biological process and fulfills the modeling objectives. In the simplest case, the experimental observations measure the change of a chemical species value over time. These experimental observations closely align with the data format that a numerical integrator uses for output, and model evaluation is straightforward curve fitting. If the observations are of phenotypic properties, then model evaluation requires a skilled modeler.

Cell cycle models typically have higher-order experimental observations that the modeler must work to interpret. An example of a higher-order observation is whether a cell representing a particular mutant strain viably reproduces. The time series plot must exhibit a checklist of requirements for the modeler to declare the mutant strain viable. These kinds of observed properties require that the modeler interpret the time series plot to see if the model matches the experimental observations. Often, it is difficult to make firm rules that guide the modeler in performing this interpretation of the model output. The modeler relies on intuition and experience when making the comparison.

$\xrightarrow{\textbf{accept}}$

Once the modeler is confident that an evaluation indicates that the model adequately reproduces the biological process and fulfills the modeling objectives, the **accept** stage begins. The **accept** stage consists of final preparations for archiving and disseminating the model. Typical means for disseminating models include web sites and scientific publications. A publication typically includes a sketch of the final biochemical reaction network, the system of differential-algebraic equations that correspond to the biochemical reaction network, and the parameters and initial conditions that the system of differential equations uses. Additionally, the modelers record the experimental observations and the procedure that they used for model evaluation. Modelers often provide the time series plots from model evaluation that depict the performance of the model against the experimental observations.

3.1.2 Secondary Stages of the Original Modeling Process

The remaining stages in the original modeling process are error-recovery stages. An error is an inadvertently incorrect model element. Errors can result from both the biology of the model and the process of building the model. Krahl [81] describes several classes of errors for general-purpose modeling, all of which can occur in biological models.

During model evaluation, the modeler detects errors in the model by examining the time series plots that the numerical integrator produces. The modeler must infer the nature of an error and its location in the model from experience. It is not always possible for the modeler to accurately identify the cause of an error. Additional laboratory experiments can test a hypothesis about a model error, but performing new experiments is extremely expensive. In general, the modeler must correct an error in the model using only the existing collection of experimental results.

Modelers often make many iterations between model refinement and evaluation before the model begins to work. However, a model missing important chemical reactions or biological mechanisms is likely to have no choice of

parameters that can adequately reproduce the desired behavior. In this case, the modeler must redesign the wiring diagram to incorporate these dynamics. Finally, the modeler may conclude that there is no hope of fixing the model, or that the cost of fixing the errors in the model is prohibitive, and discard the model in favor of another idea for how the biological process occurs.

repair $\longrightarrow$

The modeler invokes the **repair** stage when it is necessary to correct errors made while translating from the biochemical reaction network to the system of differential equations. There are two sources of such errors:

- converting the wiring diagram to chemical reaction equations,
- and converting the chemical reaction equations to a system of differential equations.

Vass and Schoenhoff [136] observed that many modelers commit errors while creating the system of differential equations. Moreover, the error rate increases when the biochemical reaction network is complex.

Although a modeler can mechanically follow the process that Section 2.4 described, manually creating a system of differential equations from a biochemical reaction network is both time-consuming and prone to error. Tedious checking between the wiring diagram and differential equations is the only means for detecting and correcting errors that a modeler introduced during model translation. Each chemical reaction equation leads to an influx and efflux term in the system of differential equations. Quite often, an error in the translated model results from the alteration or omittance from the regulating equation of one of these terms. This type of error is common among inexperienced modelers, although practice and experience can partially mitigate the problem [136].

refit $\longrightarrow$

The modeler invokes the **refit** stage when it is necessary to correct errors made while assigning numerical values to the differential equation parameters. As mentioned earlier, modelers frequently do not know exact numeric values for these parameters from laboratory experiments. When the model is not performing correctly, a modeler can make new estimates for the kinetic parameters based on past comparisons of the model against the known experimental results.

Selecting how to adjust the model parameters is highly dependent upon the current intuition that the modeler has for how the model should respond to parameter value changes. During each iteration, the modeler typically changes only a small number of parameters due to the potential interactions of the changes. The modeler often must repeat this type of error recovery activity many times before the model begins to work. Each time, the modeler successively chooses different parameters to adjust and different amounts by which to adjust those parameters. The process of parameter twiddling is the repeated adjustment of parameter values in an attempt to align the model output with the experimental observations.

redesign $\longrightarrow$

After trying parameter twiddling, the modeler may decide that an error in the model is not correctable without modifying the biochemical reaction network. The modeler invokes the **redesign** stage when making changes to the wiring diagram or rate laws. Modelers apply their biological intuition and past modeling experience to change the model by adding, removing, and modifying chemical reactions so that the model better reproduces the desired behavior. The modeler must then again translate the model into a system of differential equations and evaluate the model against the experimental observation. Since adding or removing even a single chemical reaction from the model can greatly affect the model output, an extensive period of parameter twiddling often follows the redesign of a biochemical reaction network. The modelers in the Tyson laboratory perform parameter twiddling much more often than model redesign.

restart $\longrightarrow$

Finally, the modeler may decide that correcting an error in the model is either too difficult or that correcting the error would cost more than creating an entirely new model. Not every new biological idea becomes a successful model,

and it is most efficient if the modeler weeds out the unprofitable ideas before proceeding through many iterations of the model development process. The **restart** stage is the termination of a particular biological idea and model, and proceeding through this stage marks the start of the next idea for how the biological process occurs.

3.2 Modeling Methodology

Modelers have used the original modeling process observed in the Tyson laboratory to successfully develop models that define the current state of the art. However, these modelers, and the larger modeling community, recognize that they are at the limit of the complexity that their current methodology can support, which is driving many new efforts in modeling tool development. Examples of related development efforts for modeling tools include Cell Designer [56, 78], E-CELL [130, 131], Gepasi [94, 95], Jarnac [122, 123], JigCell [6, 9], and Virtual Cell [86, 125]. Hucka et al. [71] survey the capabilities of many of these modeling tools. Furthermore, some groups of tool developers have joined forces in larger modeling efforts, such as the open source Bio-SPICE framework [30].

The proliferation of modeling applications suggests that model development has much untapped potential that a lack of computational assistance currently hinders. However, there are few opportunities for automating model development tasks in the original modeling process. Instead, it is crucial to look for steps that the original modeling process does not explicitly call out. Additional opportunities for automating model development appear after decomposing the steps of activities in the original modeling process.

The only activity in the original modeling process with a clear potential for automation is the conversion of the chemical reaction equations for the model into a system of differential equations. Although this translation process is both error-prone and time-consuming, the creation of an entirely new system of differential equations does not take place frequently and is not a bottleneck for producing well-tested biological models. Translation time is incommensurable with the time spent designing and evaluating a model. Moreover, the proportion of time spent performing translation decreases as model size increases. After the modeler initially converts the model from chemical reaction equations to a system of differential equations, the changes done to the chemical reaction equations of the model during error-recovery activities are typically small and require modifying only a small number of the differential equations.

Modeling methodologies assist in understanding the model development process and indicate requirements for supporting that process [22]. Formal methodological approaches for modeling provide well-defined and tested techniques. There is no apparent best approach for applying a modeling methodology to an existing modeling process. However, after aligning the original modeling process with the features in the modeling methodology, it is clear that there are activities that the modeling methodology contains but that the original modeling process omits. Adding in the features that the original modeling process omits seems like a natural approach to take. These discovered activities contribute to a new modeling process for constructing biological models.

Modeling methodology requirements

Based on the observed experiences with the original modeling process, it is clear that this modeling community would benefit from a modeling methodology that supports several specific capabilities. The ultimate goal of a modeler is to produce a model that validates against the modeling objectives and wins approval from the decision makers. As Section 1.1 described, demonstrating that a model is valid and acceptable requires that the modeler perform verification, validation, and testing on the model. Modelers should employ verification, validation, and testing frequently so that models that contain errors waste a minimum amount of effort [19]. If the modeler generates an incorrect system of differential equations from the chemical reaction equations, then it is likely that the model cannot pass the evaluation process with any set of parameters. Testing the model to ensure that the differential equations are correct would prevent the modeler from wasting many hours futilely twiddling parameters.

Several of the models that modelers using the original modeling process developed are extremely long lived. Modelers repeatedly adapt these models to meet changes in the modeling objectives and requirements. This is one of the reasons why the original modeling group carries out problem formulation so infrequently. Instead of starting a new model to explore a biological process, modelers take an existing model as a base and adapt the existing model to accommodate the newly proposed pathways for the biochemical reaction network. Model reuse promotes economical

model development by eliminating the duplication of work products that the two models share. However, successful model reuse requires that the modelers choose sound models.

In some cases, when the new biological process is significantly different from the biological processes that an existing model captures, modelers use a motif from the existing model instead of the entire model. Using only part of an existing model speeds up model development for the new biological process by allowing the modeler to avoid problematic interactions between the new model and existing model through the parsimonious selection of the most important aspects of the existing model. However, the modeler should assume that the ultimate goal of model development for both the new model and existing model is to eventually produce an integrated model that is self-consistent and explains both the original and new biological processes.

It is reasonable to expect that this modeling group will continue to base new models on existing models, and so these modelers require a modeling methodology and process that are capable of introducing change at an advanced stage of model development without undue cost. Furthermore, it seems apparent that computational technology will change significantly during the lifetime of a long-lived model. The previous generation of models encountered such changes. Thus, the models and the modeling process should remain independent of the runtime host and adapt to high-performance computing techniques of the next ten years or more. Using the terminology of Nance and Arthur [103], the modeling methodology and process must primarily support correctness and testability, secondarily support adaptability, maintainability, and portability, and test throughout the model lifecycle.

The primary requirements of correctness and testability from the modeling process correspond to the need for a thorough and repeatable performance of model verification, validation, and testing that ultimately produces a model that meets the model evaluation requirements. In connection with the need for the modeling process to support testability, modelers need to test their models throughout the model lifecycle to reduce the amount of effort that they waste on models that contain errors.

The secondary requirements of adaptability and maintainability from the modeling process correspond to the need for reusing existing models in the creation of new models that explain biological processes that the original modeling effort did not envision. Finally, the other secondary requirement of portability from the modeling process corresponds to the need for keeping the model representation independent of the modeling environment so that models are transportable between modeling environments. As advances in technology and high-performance computing render existing modeling environments unsuitable for continued work, new modeling environments will arise.

Conical methodology

The conical methodology [100, 101, 102] is a modeling methodology that supports the identified requirements and is sufficiently adaptable to capture both the original modeling process and a further revised modeling process that contains new modeling activities. Balci [19] describes a model lifecycle compatible with the conical methodology that includes verification, validation, and testing activities. The remainder of this section describes the conical methodology using terminology from Balci [18], Overstreet [109], and Page [112].

The domain of applicability of the conical methodology is the production of large discrete event models. Although the biological models in this dissertation are not discrete event models, biological modelers use a similar model lifecycle and construct similar kinds of model descriptions. The revised modeling process uses the conical methodology to define terminology and to generalize the model descriptions in the original modeling process.

The conical methodology prescribes a top-down model definition phase and a bottom-up model specification phase. The revised modeling process omits the model definition phase from the conical methodology. The revised modeling process has a limited domain of applicability, which allows modeling tools that implement the revised modeling process to predefine model constructs. The conical methodology ordinarily elicits a description of model objects and attributes from the modeler during the model definition phase.

The conical methodology begins with a communicated problem. A ‘communicated problem’ is the most elemental form of a problem that an individual or group wishes to solve. The communicated problem is a modeling problem from a particular domain. In this dissertation, the domain of a problem is always biology.

The modeler then performs ‘problem formulation’, which is the process of restating the communicated problem so that the problem is well-defined and amenable to specific action. The output of problem formulation is a ‘formulated problem’, which permits a decision with regard to a viable method of solution. Next, the modeler investigates possible

solution techniques with the aim of selecting a technique that has a high ratio of benefits to costs. Both the conical methodology and revised modeling process assume that the proposed solution technique is simulation.

After proposing to solve the formulated problem using simulation, the modeler then defines the requirements of the modeled system and the objectives of simulation. The remainder of the conical methodology is an iterative process for model construction and testing. A successful application of the conical methodology results in a model that decision makers find acceptable. The conical methodology does not explicitly terminate except in the case of successful completion of a model. The revised modeling process refers to the system requirements and simulation objectives as the 'problem definition'. As Section 3.1 mentioned, both the original and revised modeling processes do not include problem formulation as an explicit step. The revised modeling process assumes the existence of a problem definition before the modeler begins using that modeling process.

After establishing the system requirements and simulation objectives, the modeler formulates a conceptual model that represents the system under study. A 'conceptual model' is a representation of a model that exists only in the mind of the modeler. Page [112] describes a conceptual model in the conical methodology as likely incomplete, ambiguous, and constantly in flux. Taking a slightly different approach, a conceptual model in the revised modeling process is an instantaneous snapshot of a biological idea and hence not changing. However, a biological modeler may consider successively within a short time many different conceptual models to solve a modeling problem.

Only the conceiver of a conceptual model can work with or judge that model. Therefore, the modeler next realizes the conceptual model into a form suitable for communication to others. The 'communicative model' is a representation of the model that other modelers can understand and that exists independently from the original modeler. Other humans can compare a communicative model with the system requirements and simulation objectives, and perform verification, validation, and testing on the model.

Following the conical methodology, the modeler next engages in programming, translating the specification of the communicative model into a general-purpose or simulation programming language that a computer can compile and execute. The output of programming is a 'programmed model' that codifies a selection of programming languages, a specification of the execution environment, and an executable representation equivalent to the communicative model.

The revised modeling process diverges from the conical methodology at this point, eschewing the conventional notion of programming. The target audience of the conical methodology is modeling and simulation specialists. However, the target audiences of the original and revised modeling processes are experts in the field of biology, many of whom are not comfortable with traditional programming methodology and practice. The revised modeling process uses the more general term 'executable model' to distinguish that the executable specification of the model does not necessarily come from programming. Chapter 4 describes the process of producing an executable model.

Judging the acceptability of a model with respect to the system requirements and simulation objectives requires the design of model experiments. The modeler formulates a plan that extracts information from the model to draw inferences. An 'experimental model' is an instrumented version of the executable specification that facilitates a particular plan of investigation. In the original and revised modeling processes, biological modelers use comparisons between model output and historical laboratory data to draw inferences. The conical methodology is more general, not assuming the use of a particular technique for model analysis.

The modeler then performs the plan of the experimental model to produce experimental results. Although changes to the execution environment can affect the experimental results, the revised modeling process assumes an abstracted execution environment, with negligible differences between correct implementations. The conical methodology also does not correct for this issue. After examining the experimental results, the modeler chooses to either accept the model or continue changing the model. It is not clear from the conical methodology how the modeler makes this decision. Supposing that the model is acceptable, the modeler then interprets the results and presents conclusions to decision makers. The decision makers then propose action based on these results, although their decision may have no relation to the validity of the model or its results [112].

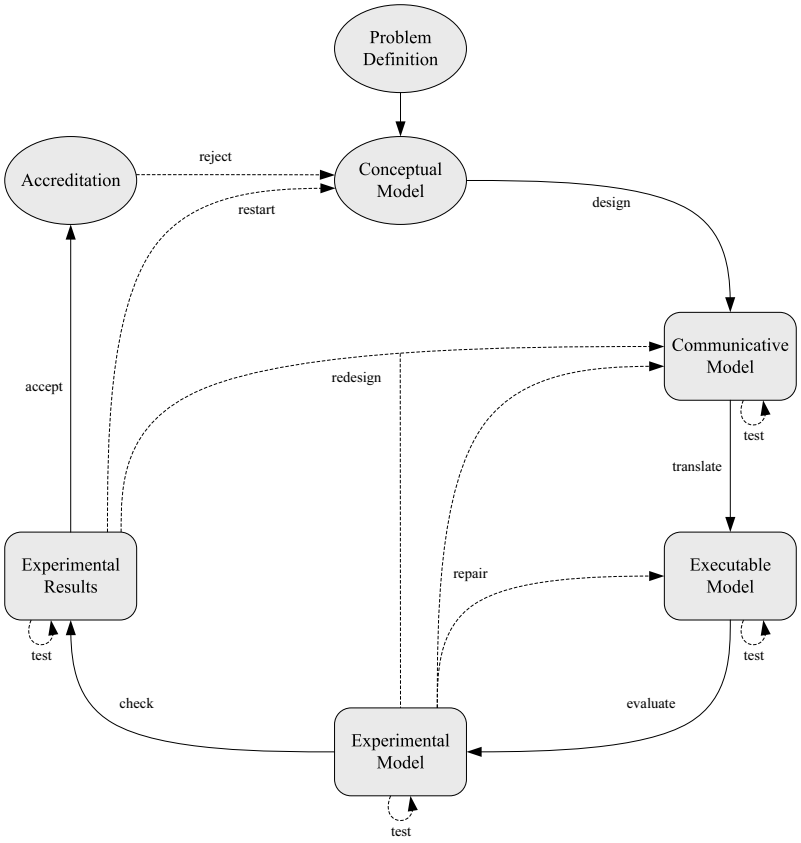
The primary objectives of the conical methodology are correctness, testability, adaptability, reusability, and maintainability [103]. These primary objectives match well with the identified primary and secondary modeling process requirements. The secondary modeling process requirement of portability is notably absent. It is necessary to instead put the burden of portability on the modeling tools to isolate the modeler from the underlying representation that the tools use for the models. Separating the model that the modeler sees in a user interface from the computer representation of the model allows modelers to move a developed model from existing to future tools. This accomplishes the

goal of protecting the modeler and developed model from changes in the computational environment.

3.3 Revised Modeling Process

Like the original modeling process, the revised modeling process assumes that the modeler previously defined the modeling problem to solve. Figure 3.4 shows the revised modeling process starting from an identified problem. This problem definition includes an analysis of modeling requirements, a plan for solving the modeling problem, and an identification of the modeling objectives that are relevant for decision makers. As with the original modeling process, the revised modeling process is constructive.

Figure 3.4: A revised modeling process that combines the original modeling process with the conical methodology.



The revised modeling process assumes that the tools that modelers employ to solve the modeling problem are adaptable to the solution technique chosen as part of the problem definition. Restricting the revised modeling process to the particular domain-specific group of problems that biological modeling encompasses justifies this assumption. A primary driver during the development of the revised modeling process and associated modeling tools was the needs of biological modelers who work on this class of problems.

The revised modeling process uses the same notation for describing stages as the original modeling process. The location, scope, and frequency of model testing activities are the most significant differences between the original and revised modeling processes. Modelers using the revised modeling process get immediate feedback about model errors after transforming the model from one form to another. For errors that the modeling tools cannot detect automatically, the modeling process can aid the identification of the source and type of error by more specifically pinpointing when the modeler introduced the error. Localizing the source of a model error reduces the amount of time that the modeler spends diagnosing the problem.

Increasing the specificity of error reporting should lead to a smaller average error-recovery time. Fault isolation is one of the most difficult and time-consuming portions of the error-recovery process. Moreover, even though it is not always possible for modeling tools to automatically detect the source and location of an error, performing modeler-defined diagnostics can determine that the modeler definitely introduced an error during the previous iteration of the modeling process. Since iterations typically reflect a small change to the model, the modeler can search within a limited set of possible sources for the error.

Terminology from the conical methodology appears in the revised modeling process. Table 3.4 summarizes the connection between terminology in the conical methodology and concepts in biological modeling.

Table 3.4: Connections between concepts in the conical methodology and biological modeling.

Modeling term	Biological modeling examples
problem definition	list of important experiments to replicate
conceptual model	biological ideas
communicative model	wiring diagram, mathematical equations, textual description
executable model	ODE file, Matlab script, program code
experimental model	time course plots, experimental observations from the literature
experimental results	list of mutant strains that the model predicts correctly
accreditation	peer review of scientific publications

3.3.1 Primary Stages of the Revised Modeling Process

Starting from the problem description, modelers begin their work by first developing model ideas that they believe will satisfy the problem requirements. The revised modeling process extends the process of realizing and testing these ideas from the original modeling process. The revised modeling process adds an additional primary stage, the **check** stage, to the four primary stages of the original modeling process. The primary stages that the original and revised modeling processes share have similar semantics between these two modeling processes. The new **check** stage uses the outputs of several model evaluations to perform further diagnostic checks on the model.

A more significant difference between the original and revised modeling processes is in the performance of error-recovery activities. In the original modeling process, the modeler undertakes error-recovery activities after determining the results of a model evaluation. Considerable time can pass between the introduction of an error and performing model evaluation. Errors that the modeler introduced into the model long ago are harder to diagnose and recover from than errors that the modeler introduced more recently. In the revised modeling process, the modeler undertakes error-detection activities, and potentially error-recovery activities, subsequent to each primary stage.

$\xrightarrow{\textbf{design}}$

After coming up with an idea for a biological process, the modeler must first produce a model that other biologists can understand. At the instant that the modeler conceives of an idea for a biological process, the model exists only in the mind of the modeler. The **design** stage is the process of taking that mentally held model, the conceptual model, and producing a model that is accessible to other modelers, the communicative model.

In the original modeling process, the output of the **design** stage is a wiring diagram. Out of necessity, which the typical lack of the required chemical reaction kinetics in the wiring diagram causes, a secondary output of the **design** stage is a collection of chemical reaction equations and kinetic formulas that corresponds to the wiring diagram. In the revised modeling process, the output of the **design** stage, the communicative model, is always the chemical reaction equations and kinetic formulas that define the model.

Modelers can still make use of wiring diagrams, but the modeling process no longer requires the production of a wiring diagram. A wiring diagram can also act as an intermediate product in the conversion from a conceptual model to a communicative model. This change to the modeling process gives modeling software the freedom to represent the model under development using chemical reaction equations directly. Modeling software might choose

to use chemical reaction equations as the model representation because chemical reaction equations are typically more amenable to computer manipulation than a graphical wiring diagram. Furthermore, modeling software can separately target different user groups of modelers by offering various levels of user interface support for chemical reaction equations versus wiring diagrams.

The choice between the textual form of chemical reaction equations and the graphical form of a wiring diagram is similar to the divide between textual and graphical input in many other application domains. Even though a modeling tool may choose to use chemical reaction equations as the underlying representation for models, it is still possible for the tool to record and display a wiring diagram to aid in the communication of the model.

The revised modeling process also includes testing activities in the **design** stage. Modeling tools can apply testing activities while the modeler constructs the communicative model. These testing activities are the first indicators of an error in model development. Modeling software at this stage can structurally test models, check models for completeness, and check user input for validity. Table 3.5 shows a selection of testing activities for the **design** stage.

Table 3.5: Questions that testing activities can try to answer during the **design** stage.

- Is there a definition for every chemical species involved in a chemical reaction?
- Is there a stoichiometric coefficient for every chemical species involved in a chemical reaction?
- Are there chemical reaction kinetics for every chemical reaction?
- Is there a definition for every parameter used in a kinetic formula?
- Are the open and closed systems in the model in correspondence with the conceptual design of the modeler?
- Do the dynamics of the model hold invariant those particular properties chosen by the modeler?
- Do the conserved moieties in the model match the understanding of the chemical system held by the modeler?

$\xrightarrow{\textbf{translate}}$

After creating a communicative model, the next step a modeler takes is to convert the model to an executable form. The executable form, an executable model, includes a system of differential equations, initial condition values, and parameter values. In the original modeling process, the modeler needed only this information to produce time series plots of the model. The simulation software and settings for the simulation routines existed independently of the model. However, the executable model incorporates information about the simulation software, simulator control settings, and runtime environment as part of the model. This ensures that the model continues to produce the same output when someone other than the original modeler tries to evaluate the model.

In the original modeling process, the modeler translates from chemical reaction equations and kinetic formulas to a system of differential equations by hand. As previous sections noted, manually performing the process in Section 2.4 is both time-consuming and error-prone. Automating this translation process is desirable. A well-described model contains all of the chemical reaction kinetics, initial condition values, and parameter values for automatic translation using modeling software. The modeler needs to tweak only the environmental information that describes the simulation software, simulator control settings, and runtime environment.

Even though the revised modeling process automates many model translation activities, modeling software still must verify that the executable model is complete, self-consistent, and tolerant of the numerical errors to which the chosen simulation process is susceptible. Table 3.6 shows a selection of testing activities for the **translate** stage.

$\xrightarrow{\textbf{evaluate}}$

The modeler next uses the executable model to perform a set of simulations for model evaluation. Model tests are recorded experiments that apply controlled inputs to the model and measure the response of the model. Model tests

Table 3.6: Questions that testing activities can try to answer during the **translate** stage.

- Is there an initial condition specified for every chemical species that the chemical system contains?
- Is there a value specified for every parameter that a kinetic formula uses?
- Can the input language of the simulation program represent the model kinetics?
- Does the model specification size exceed the capacity of the simulation program?
- Can the simulation program execute in the given runtime environment?
- Are the simulator control settings valid for the simulation program?
- Are the simulator control settings appropriate for the model?
- Do the dynamics of the model require tolerances beyond those that the simulation program can provide?
- Do the dynamics of the model exceed the storage or physical capacity of the simulation program?
- Does the model make physical assumptions that the simulation program violates?

give a pass or fail indication using the observed responses of the model. The term 'failure' for a model test indicates a problem in the model rather than the model test.

Modelers select from several types of model tests according to their needs. The problem requirements and simulation objectives from problem formulation are a significant source of model tests. Although the revised modeling process does not detail the steps of problem formulation, the discovery of model tests is another reason for modelers to thoroughly consider the biological problem before starting modeling work. The modeler needs to check each of the originally stated requirements and objectives against the proposed model. Additionally, biological modelers draw many model tests from the historically observed data that experimental laboratories produce and publications curate. Another significant source of model tests for use with model evaluation is a collection of logically-designed 'reality checks' that examine the model responses for reasonableness. A reality check subjects the model to experimental conditions for which the modeler can intuitively predict an outcome.

Collectively, the model tests, the model test plan organizing the model tests, and the information required to perform the model test plan form the experimental model. Verification, validation, and testing activities on the experimental model take up a significant portion of the model development time. Entering the problem definition once and only once, and automating the testing process from previous iterative cycles of the modeling process are important techniques for reducing model development time. Furthermore, repetitive procedures in model testing can lead to errors by conditioning the modeler to omit model tests that were frequently, or seemingly always, correct in previous trials. Modelers can miss important indicators of an error that they introduced into the model if they do not perform the test designed to detect that kind of model error. Modeling tools should automate the performance of model tests to relieve the modeler of this responsibility and reduce the number of errors that go undetected [21].

An additional factor that modeling software should consider is provision for independent execution of the model test plan to prevent modeler bias in the testing process. Independent verification and validation is the performance of verification and validation activities by someone other than the model developer. Arthur and Nance [14] emphatically conclude that independent verification and validation is an important technique for mitigating risk in model development. Additionally, it is reasonable to expect that incorporating independence into the modeling process improves model quality and operational correctness.

Verification and validation of the experimental model primarily focuses on checking that the experimental model proposes a feasible test plan, thoroughly describes the test plan, and that the model tests comprehensively evaluate the performance of the proposed model. Testing activities in the **evaluate** stage check that the modeler configured the experimental model correctly, not that the biological model itself is correct. The revised modeling process does not require a correct biological model before performing verification or validation activities. Table 3.7 shows a selection

of testing activities for the **evaluate** stage.

Table 3.7: Questions that testing activities can try to answer during the **evaluate** stage.

- Are there tests that evaluate the proposed model against all of the known historical data?
- Are there tests that check the proposed model against the originally stated problem requirements?
- Are there tests that ensure that the proposed model meets all of the originally stated simulation objectives?
- Does the executable model capture necessary model results during testing?
- Does each test have the information that a modeler needs for execution and analysis?
- Is there a defined procedure for transforming the model output during evaluation?
- Is there a defined procedure for comparing the model output to the recorded test data?
- Is there a schedule such that performing model evaluation with the test plan is feasible?
- Does the model test plan record all of the information that a modeler needs to run the tests so that someone independent of the original modeler can evaluate the model?

check $\longrightarrow$

Decision makers can still reject a model that passes the tests for the problem requirements and modeling objectives stated during problem formulation. The tests that the modeler performed on the experimental model may not have been sufficient to thoroughly evaluate the compliance of the proposed model with respect to the problem requirements and modeling objectives. Ideally, the testing process would detect this error of insufficient model testing in the **evaluate** stage so that modeler does not waste time fine-tuning a model that the modeling tool is not evaluating adequately. During the **evaluate** and **check** stages, modeling tools should flag gaps in the evaluation record so that the modeler can review the evaluation process. However, automated testing cannot detect whether the techniques that the modeler applied during model evaluation are adequate to correctly identify errors in the proposed model.

Excluding the presence of errors in the evaluation procedure, there are two common reasons in the biological domain for why decision makers reject a model that passes all of its tests:

- the proposed model is insufficiently based on established biological processes,
- or the proposed model is not significantly better than an existing, simpler model.

The **check** stage works to address both of these issues.

Comparing the proposed model against accepted models that represent similar processes is useful as a test that established biological processes underlay the proposed model. The modeler uses accepted models as a baseline for determining how the proposed model should respond to simple, controlled inputs. By generating results from the accepted models, the modeler creates a plausible substitute for historical data and can compare the results with the output of the proposed model under similar conditions. It is frequently not possible to test the full fidelity of the proposed model against existing, accepted models because it is rarely feasible to locate an existing model for each of the problem requirements and modeling objectives. The modeler cannot give the accepted models inputs that are outside the domain of their accreditation and expect to produce meaningful results.

A collection of existing models for a system is useful when testing whether the proposed model is a significant improvement. The modeler should perform a statistical analysis of ranking and selection between the proposed model and the existing models. Using the model tests for evaluation given in the model test plan, the modeler scores each of the models, and chooses a best model based upon the results of the evaluation. Ranking and selection does not always result in the identification of a single best model. When this occurs, the output is a collection of best models

with similar performance characteristics. Both the technique of ranking and selection and the technique of generating plausible data from accepted models test the proposed model against other models of biological systems [139].

Since the modeler drives the **check** stage primarily by exploring the results of previous model verification, validation, and testing activities, there are comparatively few automated tests that modeling software can perform during this stage. The automated tests focus on the past thoroughness for model validation, verification, and testing and on the materials that the modeler produced for use during model accreditation. Table 3.8 shows a selection of testing activities for the **check** stage.

Table 3.8: Questions that testing activities can try to answer during the **check** stage.

- Is there a record of the performance of the model tests from previous stages?
- Is there a record of the model transformations from previous stages?
- Has the reviewer of the model indicated that the model acceptably satisfies the originally-stated problem requirements?
- Has the reviewer of the model indicated that the model acceptably fulfills the originally-stated modeling objectives?

$\xrightarrow{\textbf{accept}}$

The **accept** stage remains relatively unchanged from the original modeling process. The preparations in the **accept** stage correspond to the process of creating documentation and presentations that show that the model is sufficiently accurate for its intended purpose [19]. This documentation plays an important role in model acceptance. The stakeholders of the model, the individuals and groups that have funded the model or have vested interest in its construction, will review the model and documentation to determine if the modeler successfully fulfilled the modeling objectives. Other modelers can use this documentation to improve their understanding of the model and make better use of the developed model in their work.

It is important to emphasize that the intended purpose of the model dictates the required accuracy of the model. Models that are not operationally correct within the tolerances set by the modeling problem description obviously cannot fully satisfy their intended purpose. However, models that are operationally correct but include details unnecessary for the modeling objectives may also fail the model acceptance process. Overly-detailed models typically are more expensive to produce, slower to execute, more fragile to changes, and more complex to explain. Parsimony in building models is a significant aid to model acceptance.

3.3.2 Secondary Stages of the Revised Modeling Process

The revised modeling process considerably augments the testing phases in comparison with the original modeling process. At every stage in the revised modeling process that creates or transforms a recorded model description, there are tests that check that the model is still valid. Additionally, the results of model testing more specifically point to causes of errors and direct the modeler to an appropriate stage for correcting the error. In many cases, modelers spot errors before they leave the stage in which they created the error. Previously, the modeler would not detect the error until performing model evaluation. This lowers the cost of correcting the error by alleviating the need to do extensive debugging to localize faults.

$\xrightarrow{\textbf{test}}$

The **test** stages located throughout the revised modeling process represent verification, validation, and testing activities that take place concurrently with model development. Each primary stage description for the revised modeling process gave examples of procedures that exercise the proposed model that modeling software can perform during test

stages. Model tests operate concurrently with modeling tool use and indicate that the modeler entered an error into the modeling tool or that the modeling tool incorrectly transformed the model. If a modeler finds an error using a model test and corrects the error before leaving that stage, then the error does not propagate to other stages of the modeling process. Immediate detection of errors reduces the time that modelers spend correcting errors and reduces unnecessary switching between modeling tools.

Continuous verification is especially important when the modeler has experimental data for model evaluation with limited quantity or quality. The modeler is less likely to detect errors in the model by automated means during model evaluation when there is a limited pool of experimental data. Instead, the modeler must spend more time evaluating the model by hand during each iteration of the modeling process. By expending some effort to perform continuous verification, the revised modeling process in return reduces the amount of work that the modeler expends performing model evaluation without increasing the chance of model errors going undetected.

$\xrightarrow{\textbf{repair}}$

The **repair** stage consists of activities that modelers perform after model verification indicates that they introduced an error while transforming the model. Model repair should occur less frequently with the revised modeling process than with the original modeling process. Automating the model transformation process eliminates many sources of model error. Model tests immediately detect some of the remaining errors that modeling tools introduce while transforming the model. Errors that a model test detects and that modelers fix before leaving the stage in which they introduced the error do not trigger the occurrence of a **repair** stage. The **repair** stage in the revised modeling process includes the activities of both the **repair** and **refit** stages in the original modeling process. In Chapter 4, an implementation of the revised modeling process once again separates these activities.

$\xrightarrow{\textbf{redesign}}$

The **redesign** stage consists of activities that parallel the **repair** stage. Model redesign consists of the correction of errors that the modeler detected during model validation. Modelers choose to redesign a model when they believe that the basic conceptual model is correct but that they introduced errors while creating the communicative model. If the conceptual model contains an error, then the modeler must restart the model development process rather than trying to tweak the implementation.

$\xrightarrow{\textbf{restart}}$

Restarting the model development process for a model that the modeler has given up on has not changed from the original modeling process. In the revised modeling process, the **restart** stage is an alternative that the modeler uses when it is inevitable that decision makers will reject the model during accreditation. Choosing to restart the model development process instead of spending the time preparing and documenting the model for accreditation is cost-effective when successful accreditation is unlikely.

$\xrightarrow{\textbf{reject}}$

The **reject** stage is a new counterpart to the **accept** stage in the original modeling process. A proposed model can pass all of the tests for model evaluation, and appear to fulfill all of the problem requirements and modeling objectives, but decision makers might still reject the model. The descriptions of the **check** and **accept** stages discussed some reasons why decision makers reject seemingly suitable biological models.

Models that decision makers reject often need careful revision to make the model acceptable. Since model development has significant cost, rejecting a valid model is a considerable expense. Therefore, decision makers tend not to reject models unless they consider the model to have fundamental flaws that need correction. The **reject** stage leads back to an examination, and potentially modification, of the conceptual model. After decision makers reject a model, the modeler must propose a new design for the biological process.

Chapter 4

JigCell Modeling Environment

JigCell is a suite of applications, programming libraries, and utility programs that forms a computational environment for biological modeling. JigCell focuses on the production, execution, and analysis of models of biochemical reaction networks. The three applications in JigCell that correspond to these modeling activities are the JigCell Model Builder, Run Manager, and Comparator. Figure 4.1 shows the user workflow in JigCell, which corresponds closely with the revised modeling process that Section 3.3 described. The revised modeling process introduced several forms of biological and mathematical models, including the communicative, executable, and experimental model forms. Each form of the model has a corresponding application in JigCell. Like the original and revised modeling processes, JigCell does not attempt to deal with problem formulation.

The focus on biochemical reaction networks in JigCell does not exclude the use of JigCell for other types of modeling. The basic units of models in JigCell are chemical reactions rather than biochemical reaction networks. A modeler working on a chemical reaction model could still make use of JigCell even though there is no biological application for the problem. However, the modeling terminology that JigCell uses might not correspond to the modeling terminology that a non-biological modeler would expect. Non-biological modelers must mentally translate the modeling terminology in JigCell to the terminology that their domain typically uses. During the development of JigCell, there was no concerted effort to generalize the modeling terminology that appears in the application user interfaces.

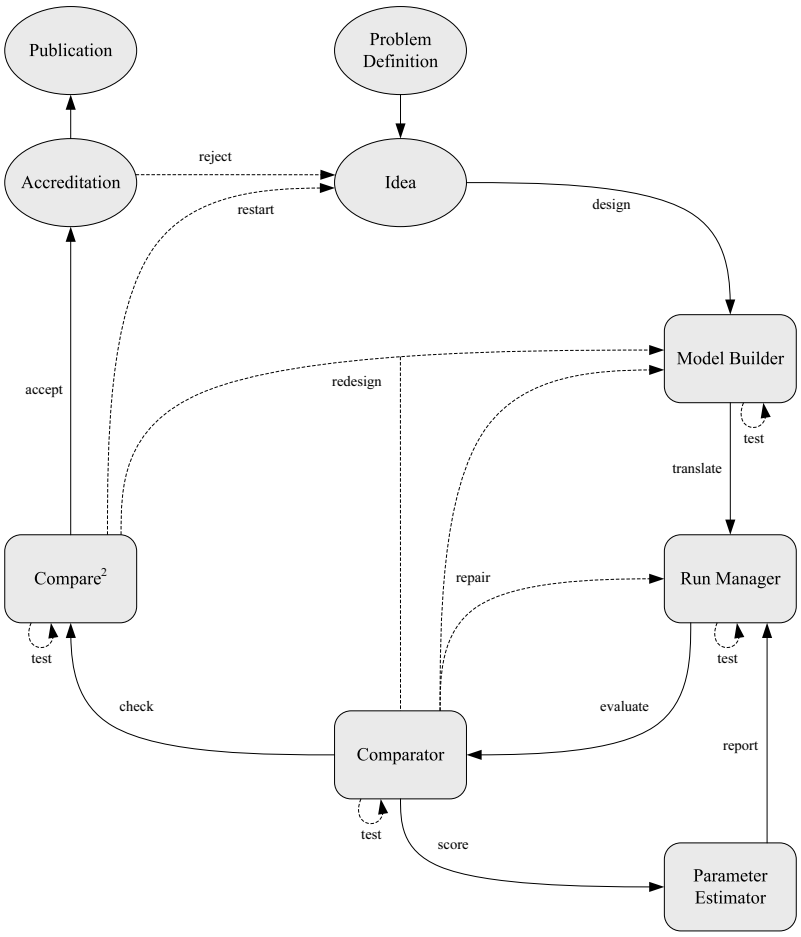
The goal of JigCell is to support users who are domain experts in biology and other related fields but who do not have significant experience in formal modeling. To this end, JigCell acts both as a computational environment in which biological modelers work and a research environment in which computer science researchers study how biological modelers build models. The design of JigCell does not come from any existing, general-purpose computational environment. During the development of JigCell, biologists and biological modelers were able to access the software for testing. In addition to reporting errors in the applications, these early users of JigCell gave feedback about new features that JigCell could implement and suggestions for development priorities.

JigCell incorporates off-the-shelf components, such as numerical libraries, visualization tools, and communications protocols, when quality implementations of such components exist. In many cases, this approach was not a significant drawback to the development or use of JigCell. The creation of domain-specific support occupied the majority of development work. The time spent building modeling infrastructure was small in comparison.

Primarily, the problems with using off-the-shelf components came from the determination of whether a particular component is a ‘quality implementation’. The use of off-the-shelf components led to significant dependencies from JigCell to these components. Failures and errors in the dependent components disrupted the development of JigCell. Additionally, the off-the-shelf components were not sufficiently transparent to the end user. Some of the components had difficult requirements for installation or use. These requirements affected the user experience of JigCell negatively. JigCell was not able to directly remediate these problems because maintenance of the components was outside the scope of the project. Eventually, JigCell discontinued the use of these problematic components to improve the user experience. Section 7.2 and the sections that describe the major JigCell applications contain more details and historical notes about these experiences with incorporating off-the-shelf components into a computational environment.

The focus of this chapter is to introduce the JigCell modeling environment and detail some of the features that the

Figure 4.1: User workflow that demonstrates the relationship between the JigCell applications and the revised modeling process.



JigCell applications provide in support of biological modeling. The remainder of the chapter describes how JigCell supports biological software developers and where JigCell is amenable to future extension. JigCell is a demonstration that the revised modeling process of Section 3.3 is possible to implement and provides a way of measuring the effectiveness and impact of the revised modeling process on real biological problems. Later chapters continue to examine this issue by actually measuring how closely JigCell aligns with the revised modeling process and how effectively JigCell supports biological modeling.

The present chapter starts with an overview of JigCell and examines how JigCell functions as a computational environment. Section 4.1 describes two types of computational environments, modeling support environments and problem-solving environments, and considers how well JigCell supports the principles of both. Next, there are separate sections that discuss the Model Builder, Run Manager, and Comparator applications in JigCell. Section 4.5 introduces the remaining pieces of JigCell, consisting primarily of the simulators, programming libraries that support the applications, and utility programs. Finally, Section 4.6 discusses several related applications for biological modeling that are not yet a part of JigCell.

Contents

4.1 Modeling Environments

Although the term 'computational environment' applies to JigCell in a generic sense, a more specific and applicable term is 'modeling support environment'. A modeling support environment is an integrated computational environment that enhances the productivity of the user for modeling and simulation tasks [27, 90]. Modeling support environments provide several key modeling tools, including tools for specifying models, translating models between different forms, executing models, and analyzing models. Note that the major applications of JigCell closely correspond to the basic tools of a modeling support environment. JigCell does not include a dedicated tool for model translation. Instead, JigCell invisibly supplies this function of a modeling support environment as part of the tool integration effort instead of making model translation an explicit tool that the user invokes. When the modeler using JigCell specifies a product that another tool in JigCell created, the current tool translates the model automatically.

The idea of building an integrated computational environment for modeling and simulation is not new. Early examples of proposed modeling simulation environments include those by Henrikson [66] and Standridge and Walker [129]. In the area of biological modeling, the widespread introduction of modeling support environments is more recent. Section 3.2 listed several modeling support environments for biological modeling, most of which are less than five years old. Many of the early tools for biological modeling were simulation tools, with little or no capability for model creation or analysis. Numerical simulation is probably the area of biological modeling that is most amenable to computer automation. Additionally, numerical simulation is also likely the area of biological modeling for which an expert modeler first requires computer automation.

An expert modeler can construct the regulating differential equations for a modestly sized model and, with difficulty, interpret the numerical output. However, finding a numerical solution to a system of differential equations rapidly becomes unfeasible even for an expert modeler with a modestly sized model. Moreover, efficiently computing a solution to a system of differential equations requires detailed mathematical and computational knowledge. It is therefore natural that the early tools for biological modeling centered on the simulation of models.

Several factors spurred the development of modeling support environments for biological modeling. Using the modeling perspectives dimensions of Balmer [26], the key factors that spurred the development of modeling support environments were the transitions

- from specialist computational modelers to end users,
- from small models to large models,
- and from simple decisions about well-defined problems to complex decisions about ill-defined problems.

An ever-increasing number of biologists build biological models. Although an expert modeler might successfully build a model with only limited tool support, novice modelers require more affordances and a better user experience. Most standalone simulation programs only accept models in the form of a system of differential-algebraic equations. Modelers often start with only a sketch of a biochemical reaction network. As Section 3.1 described, novice modelers particularly benefit from computer assistance in the translation of models. Modeling tools require integration to support the automatic translation of models between various forms.

As models became larger, the process of building and defining those models became more difficult. Large models have complex interactions in their definition and the difficulties of working with such models generally scale nonlinearly. Section 2.4 described some of these problems. Hence, the need for model creation tools increased.

Finally, increasing computational assistance changed the most-pressing problems for biological modelers. The problem that numerical integrators solve is 'What is a solution vector for this set of differential equations?'. A mathematician can construct a precise definition of this problem for many biological models. After improvements in computation reasonably solved the problem of finding a solution vector for this class of models, the next clear problem to solve is 'Does this solution vector reasonably approximate the experimental observations?'. The new problem of judging the quality of a solution is more ambiguous and leads to the need for model analysis.

Problem-solving environments

Problem-solving environments are another specialization of the concept of a computational environment [117]. This dissertation considers problem-solving environments where the identified problem is the construction, execution, and analysis of a biological model. Problem-solving environments provide integrated access to a selected collection of tools [115, 137], much like a modeling support environment. However, a problem-solving environment goes further by encompassing several computational aspects that a typical modeling support environment lacks.

Problem-solving environments often focus on problems with integrated and interdisciplinary components. For this reason, problem-solving environments must often specialize for multiple classes of domain users and provide separate tools for each of these classes. In contrast, a modeling support environment typically incorporates tools that specialize to a single, particular class of domain users. It is often easier to customize a modeling support environment by changing the selection of tools to suit the needs of an individual user than to create a modeling support environment that meets the needs of disparate classes of users.

Another difference between problem-solving environments and the typical modeling support environment is that problem-solving environments often contain greater support for remote, distributed, and supercomputing resources. An increasing number of scientific computing problems, including biological modeling, require resources in excess of those available with a typical personal computer. Problem-solving environments recognize this need for high-performance computing resources by integrating easy and transparent access to external computational resources.

JigCell is not yet a problem-solving environment. JigCell heavily specializes to the domain of biological modeling. In particular, JigCell uses terminology that biologists find comfortable but that other domain experts, such as chemists or physicists, may not recognize. Although JigCell previously attempted to provide access to remote computational resources, the current applications in JigCell generally do not access computational resources apart from those available on the local machine of the end user. Users of JigCell perform tasks that generally do not require extensive computational resources.

JigCell will require greater access to computational resources in the future. Parameter estimation, which Section 4.6 describes as an alternative to parameter twiddling, is resource-intensive. JigCell must provide access to high-performance computing resources before parameter estimation becomes feasible for many biological problems.

4.2 JigCell Model Builder

The JigCell Model Builder is responsible for creating and editing the communicative model. Marc Vass created the original JigCell Model Builder in 2001 [135]; Nicholas Allen and Ranjit Randhawa created a new program with the same name in 2005. As in the revised modeling process, the communicative model is a translation of the conceptual model that the modeler has in mind. Hereafter, this dissertation commonly uses the term 'model' synonymously with the communicative models that the Model Builder creates. The basic elements of models in JigCell are chemical reactions, which the Model Builder presents in a manner similar to the chemical reactions in Chapter 2.

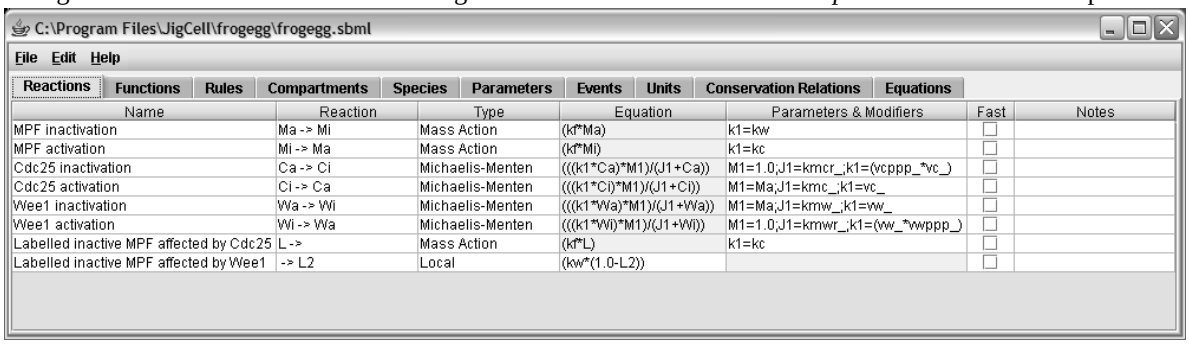
The Model Builder records models using the Systems Biology Markup Language (SBML) [71, 72]. SBML is rapidly becoming a standardized interchange language among the systems biology community. Currently, the Model Builder supports reading and writing models with SBML Level 2 Version 1 [54]. However, the Model Builder does not know about the syntactic or semantic rules of SBML. Instead, the Model Builder uses an SBML parsing library that can convert between SBML and an object-oriented representation of a model. Section 4.5 describes this SBML parsing library in more detail. The choice of SBML for representing models influences other portions of the Model Builder. Inside the Model Builder, the storage of the model reflects the structures that SBML uses to organize a model.

The Model Builder has multiple user interface screens, one screen for each major component of the model, that appear in separate tab windows. These components correspond to the model structures of SBML.

Chemical reactions

User interface screens in the Model Builder use a spreadsheet metaphor to present information. Figure 4.2 shows a collection of chemical reactions in the 'Reactions' spreadsheet of the JigCell Model Builder. The spreadsheet interface organizes information in a straightforward manner. For example, in the 'Reactions' spreadsheet, each row of the spreadsheet represents a distinct chemical reaction. These chemical reactions are all irreversible. The Model Builder requires that a modeler enter a reversible chemical reaction as two separation reactions. The other tabs in the Model Builder user interface similarly place one distinct element of the model on each row of their spreadsheet.

Figure 4.2: Chemical reactions in the JigCell Model Builder from the *Xenopus laevis* model in Chapter 3.



Each column in the 'Reactions' spreadsheet represents a piece of information about the chemical reactions. For example, the first column in the chemical reaction spreadsheet contains a descriptive name for the reaction and the second column in the spreadsheet contains the chemical reaction equation. The chemical reaction kinetic formula is split among the 'Type', 'Equation', and 'Parameters & Modifiers' columns.

To use a previously defined kinetic formula, the modeler chooses the name of that kinetic formula in the 'Type' column. An equation for the general form of that kinetic formula appears in the 'Equation' column, and the modeler can specify values for the parameters in the equation using the 'Parameters & Modifiers' column. The Model Builder automatically defines the Michaelis-Menten kinetic formula and mass action kinetic formulas for any number of reactants. The modeler can define additional kinetic formulas for use in the Model Builder by going to the 'Functions' spreadsheet, entering an equation for the kinetic formula, and giving that kinetic formula a name.

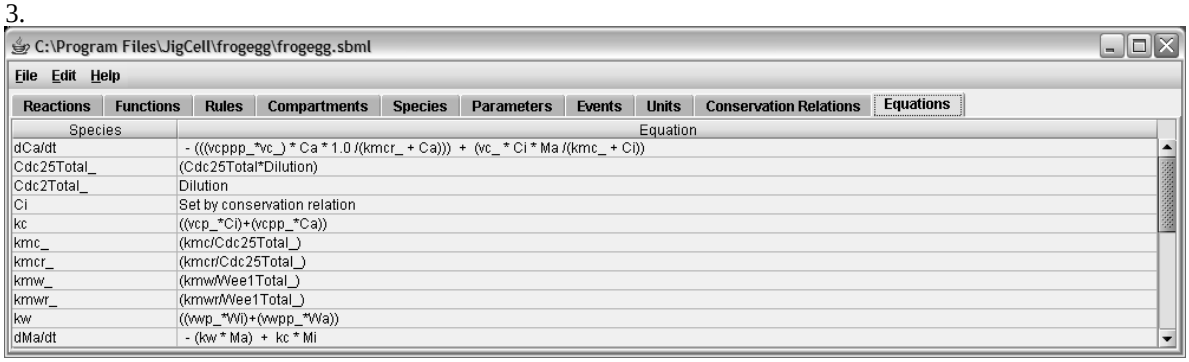
If the modeler does not want to use a previously defined kinetic formula, then the modeler can specify an arbitrary equation for the kinetic formula of a chemical reaction. To enter an arbitrary equation, the modeler chooses the local rate law option in the 'Type' column instead of specifying one of the existing kinetic formulas. After the modeler specifies that the rate law is a local rate law, the 'Equation' column becomes editable for that chemical reaction and the modeler can type an arbitrary equation for the kinetic formula. Giving a local rate law for a chemical reaction disables the 'Parameters & Modifiers' column, which this method for specifying the kinetic formula does not need.

Currently, reuse of previously defined kinetic formulas is all that the Model Builder supports in terms of model reuse. The SBML standardization effort has not yet defined any mechanism for including sub-models as either black-box or white-box components inside of a larger model. The Model Builder would have to overcome several difficult, technical issues to support a more powerful mechanism for model composition. In particular, modelers do not agree on the names to provide for elements of the model, such as chemical reactions and species. Unless there is some standardization for the names of model elements, the Model Builder would require a tedious 'identification' phase where the modeler must equate the elements of separate models by hand.

The other tabs in the Model Builder user interface similarly present a piece of the model using a spreadsheet metaphor. As the previous text mentioned, the 'Functions' spreadsheet is where the modeler defines and names equations for use in other parts of the model. The 'Units' spreadsheet allows the modeler to define units of measurement that apply to the numerical values that the modeler enters in the Model Builder. Figure 4.3 shows the 'Equations'

spreadsheet that summarizes in a single screen the differential and algebraic equations from throughout the model. The Model Builder constructs this summary system of differential-algebraic equations using the process that Section 2.4 described. Although there is no guarantee that the simulator uses the same set of differential-algebraic equations when executing the model, the modeler can expect that the simulator uses an equivalent system.

Figure 4.3: Differential and algebraic equations in the JigCell Model Builder for the *Xenopus laevis* model in Chapter 3.



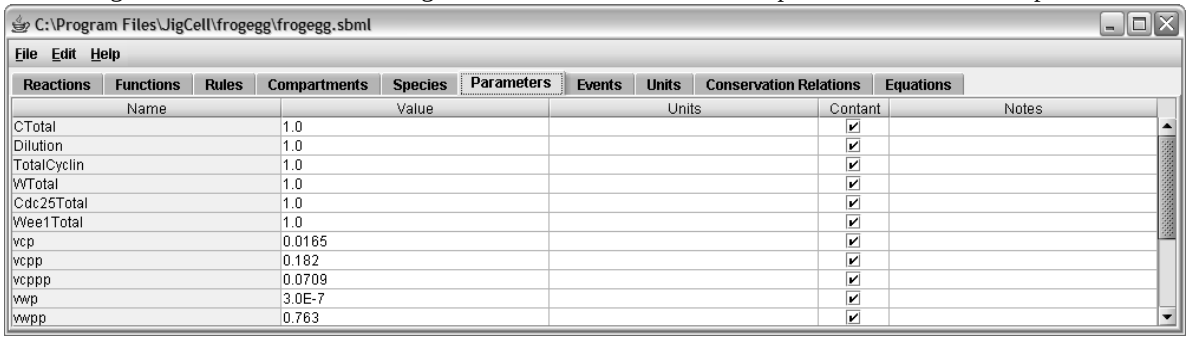
4.2.1 Specifying Model Variables

The ‘Compartments’ spreadsheet organizes the cellular compartments in the model. Each compartment has an initial size, unit of measurement, and topological properties, such as the dimension of the enclosed space and a reference to the enclosing compartment. In every new model, the Model Builder creates by default a single compartment with unit volume named ‘Cell’. Compartments are the only form of spatial organization in SBML and consequently the Model Builder. The Model Builder cannot represent models that contain spatially dependent equations. Moreover, the number, dimensionality, and topology of compartments cannot change during simulation. Although several individuals proposed schemes for the inclusion of spatial modeling, SBML has not to date accepted these proposals.

Note that entering an initial size for a compartment is optional. If the modeler chooses not to give an initial size for a compartment in the model, it is still possible to define an initial size for that compartment later using the Run Manager. The next section describes the Run Manager in detail. However, if the modeler does not give an initial size for a compartment in both the Model Builder or Run Manager, then the model is not simulatable.

The ‘Species’ and ‘Parameters’ spreadsheets allow the modeler to define chemical species and kinetic rate constants. The modeler typically does not need to add additional chemical species or parameters to the model. The Model Builder automatically creates appropriate chemical species and parameters from the context when the modeler enters a chemical reaction equation or kinetic formula. Figure 4.4 shows the layout of the ‘Parameters’ spreadsheet in the Model Builder. The tabs for editing species and compartments are similar to Figure 4.4 but have more options due to the greater detail with which SBML describes these model elements.

Figure 4.4: Parameters in the JigCell Model Builder for the *Xenopus laevis* model in Chapter 3.



SBML requires that every chemical species reside inside some compartment of the model. By default, the Model Builder places newly declared chemical species inside the 'Cell' compartment, if that compartment exists. The modeler later can reorganize the model by changing the compartment for a chemical species.

As with compartments, chemical species and parameters have an initial value, units of measurement, and topological properties. Additionally, the modeler does not have to give initial values for a chemical species or parameter in the model, but can defer that decision until using the Run Manager. When a conservation relation eliminates a chemical species, as Section 2.4.2 described, the Model Builder blocks the modeler from entering an initial value for that chemical species to prevent the construction of an inconsistent model.

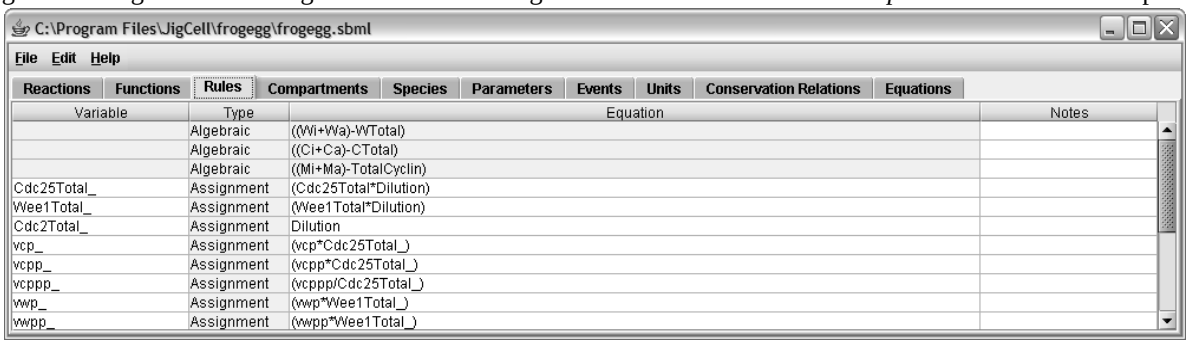
4.2.2 Controlling Model Variables

The 'Rules' spreadsheet allows the modeler to specify differential and algebraic constraints. These constraints are separate from the biochemical reaction network and the chemical reactions do not reflect the constraints. SBML supports three types of rule-based constraints. Differential rules allow the modeler to specify a differential equation for a chemical species, parameter, or compartment size. Assignment rules allow the modeler to directly specify an explicit function for the value of a chemical species, parameter, or compartment size.

The most complicated type of rule that SBML supports is the algebraic rule. Algebraic rules allow the modeler to specify an implicit function for the value of a chemical species, parameter, or compartment size. Since there is no analytical method for solving the general implicit functions that a modeler can enter, the numerical integrator must approximate a solution for the algebraic rule starting from a guessed point. Unlike a differential or assignment rule, an algebraic rule does not specify which variable the rule defines. Instead, the numerical integrator must infer an appropriate variable for the rule.

The Model Builder currently uses algebraic rules to specify the conservation relations in the model. It seems likely that this approach will require modification in the future. Although algebraic rules are more general than assignment rules, expressing conservation relations does not require this power. Numerical integrators are more likely to support the use of assignment rules than algebraic rules, which means that expressing conservation relations using assignment rules is the more portable approach. Figure 4.5 shows the 'Rules' spreadsheet of the Model Builder, including several algebraic rules that are due to conservation relations. It is not difficult to rewrite each of these conservation relations using assignment rules instead of algebraic rules.

Figure 4.5: Algebraic and assignment rules in the JigCell Model Builder for the *Xenopus laevis* model in Chapter 3.



The 'Conservation Relations' spreadsheet is where the modeler can view the conservation relations that the Model Builder detected in the biochemical reaction network. If desired, the modeler can propose a new set of conservation relations to use instead. The Model Builder uses the algorithm for calculating conservation relations that Section 2.4.2 described. Like many of the computations described in this section, the SBML library that the Model Builder rests upon contains the implementation of the algorithm that determines conservation relation. When the modeler proposes an alternative set of conservation relations, the Model Builder checks this set using another algorithm located in the SBML library. Section 4.5 describes in more detail the algorithm for validating a set of conservation relations that the modeler supplies. Additionally, the modeler can choose a different dependent species whose differential equation the conservation relation eliminates and a different name for the parameter that represents the conserved total.

Finally, the 'Events' spreadsheet allows the modeler to specify discrete events that occur during the numerical integration process. A discrete event consists of a 'trigger condition', a boolean expression, and a list of assignments that take place when the trigger condition becomes true. An assignment modifies the value of a model variable according to a formula that the modeler specifies. Additionally, a discrete event can have a time delay, which is the amount of time that the numerical integrator should wait after the discrete event occurs before applying the event assignments. The numerical integrator applies the assignments for a delayed event at the specified time regardless of the behavior of the continuous model or execution of events that occur in the meantime.

Many numerical integrators do not support discrete events because it is difficult to detect when the trigger condition occurs. A trigger condition is a function of the model variables and evaluates to a boolean expression. The event occurs at the instant when the trigger condition changes from false to true. There is no analogous event when the trigger condition changes from true to false and no event can occur at the start of simulation. A simulation program must convert the trigger condition to a numerical expression and use a root-finding procedure to detect when the trigger condition occurs. After the numerical integrator applies event assignments, it must destroy any recorded evaluation history. The application of event assignments causes a discontinuous change to the model and invalidates any local functional approximation that the numerical integrator created from previous time steps.

4.3 JigCell Run Manager

The JigCell Run Manager is responsible for creating and editing the executable model. Marc Vass created the original JigCell Run Manager in 2001 [135]; Nicholas Allen created a new program with the same name in 2005. An executable model consists of a collection of experimental configurations, or 'runs', that the Run Manager can apply to an existing model. The term experimental configuration comes from the common use of a modeler wanting to modify a model to replicate an existing laboratory experiment. However, the word 'experiment' in the term 'experimental configuration' can cause confusion as JigCell works with many other aspects of experiments, such as experimental data. Therefore, the Run Manager uses the term 'run' as a replacement.

As Section 3.3 described for an executable model, the definition of a run includes a reference to a model and information for constructing the execution environment of a numerical integrator. The Run Manager uses the communicative models that the Model Builder previously created and stored in SBML. As SBML is a standardized language, the Run Manager can also use model files that other model editing software tools create.

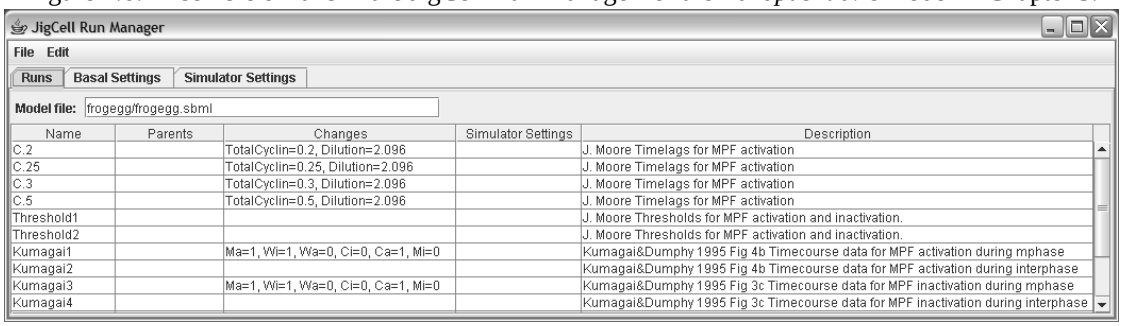
The Simulator API, a generic interface for communicating with simulation programs that generate time-course data, hides the execution environment from the numerical integrator. The Run Manager records only an identifying token for the simulation program and the control settings for the numerical integrator that the modeler desires to use with the model. Section 4.5 describes the Simulator API in more detail.

Like the Model Builder, the Run Manager is not directly responsible for reading and writing the data that the modeler edits. There is a standalone library for reading, writing, manipulating, and executing runs, which Section 4.5 describes. The Run Manager produces two data files for each logical run file. A basal file contains a collection of values for chemical species, parameters, and compartment sizes that override the corresponding values in the model. A run file defines an ensemble of runs in terms of changes that apply to the settings in the basal file. Since the run file contains most of the information for a run, it is common to reuse the term 'run file' to mean the entire collection of information that makes up an ensemble of runs.

Ensemble of runs

The Run Manager uses a spreadsheet metaphor to organize data, similar to the organization of data in the Model Builder. Figure 4.6 shows the main 'Runs' tab of the Run Manager, which organizes the ensemble of runs. There is a field at the top of the 'Runs' tab for the modeler to specify a model file. The Run Manager uses a single model for all of the runs in a run file. The run file does not duplicate the storage of a model. There is a reference to a model file inside the run file, and the Run Manager retrieves the model from the specified file each time there is a need for the model definition.

Figure 4.6: Ensemble of runs in the JigCell Run Manager for the *Xenopus laevis* model in Chapter 3.



Name	Parents	Changes	Simulator Settings	Description
C.2		TotalCyclin=0.2, Dilution=2.096		J. Moore Timelags for MPF activation
C.25		TotalCyclin=0.25, Dilution=2.096		J. Moore Timelags for MPF activation
C.3		TotalCyclin=0.3, Dilution=2.096		J. Moore Timelags for MPF activation
C.5		TotalCyclin=0.5, Dilution=2.096		J. Moore Timelags for MPF activation
Threshold1				J. Moore Thresholds for MPF activation and inactivation.
Threshold2				J. Moore Thresholds for MPF activation and inactivation.
Kumagai1		Ma=1, Wi=1, Wa=0, Ci=0, Ca=1, Mi=0		Kumagai&Dumphy 1995 Fig 4b Timecourse data for MPF activation during mphase
Kumagai2				Kumagai&Dumphy 1995 Fig 4b Timecourse data for MPF activation during interphase
Kumagai3		Ma=1, Wi=1, Wa=0, Ci=0, Ca=1, Mi=0		Kumagai&Dumphy 1995 Fig 3c Timecourse data for MPF inactivation during mphase
Kumagai4				Kumagai&Dumphy 1995 Fig 3c Timecourse data for MPF inactivation during interphase

At the bottom of the ‘Runs’ tab, there is the ensemble of runs, with one run on each line of the spreadsheet. Each run has a name, description, and a set of changes. A change is a modification to the value of one of the chemical species, parameters, or compartment sizes in the model. Unlike the Model Builder, which only supports initializing model variables with numbers, the modeler can specify an equation for the changes in the Run Manager. The equation can use any of the variables in the model, and the numerical integrator evaluates the equation with respect to the settings in the basal file before simulation.

The modeler can then organize runs into a hierarchy using the parent system. A run inherits changes from its parent runs. A run has multiple parents in an ordered list, and each parent applies its changes in that order. Thus, if the hierarchy contains multiple runs that change a particular model variable, the last change to the model variable is the one that the numerical integrator receives. The run applies its changes after all of its parent changes.

The hierarchy of runs allows the modeler to describe inheritance. There are many situations where the modeler can use inheritance to avoid redundantly inputting information about the experimental conditions for a run. For example, a common task for a run file is to describe a collection of mutant strains for a particular model. The model by itself represents the wildtype, or normal, version of the organism. A mutant strain changes model variables according to the addition, deletion, or modification of various genes. The modeler may have many runs that represent mutant strains that change a single gene. Then, the modeler wants to create a run that represents a mutant strain that changes two genes at the same time. If both of the single gene changes have a run, then the modeler can simply create a new run that inherits from the appropriate two runs.

Using inheritance reduces the likelihood of having two, differing definitions of the same experimental condition by eliminating the need for the modeler to enter a particular experimental condition more than once. Additionally, when the modeler needs to modify one of the experimental conditions, those modifications propagate down to all of the runs that inherit that change. This makes updating a run file with a new model or new experimental conditions easier. Later, this section shows how the modeler chooses the simulation program and default simulator control settings.

The final column in the ‘Runs’ spreadsheet of the Run Manager is the ‘Simulator Settings’ column. The ‘Simulator Settings’ column allows the modeler to specify changes to the control settings of the numerical integrator that apply only to that particular run. However, the modeler cannot specify different simulation programs for different runs in a run file. All of the runs in a run file must use the same numerical integrator.

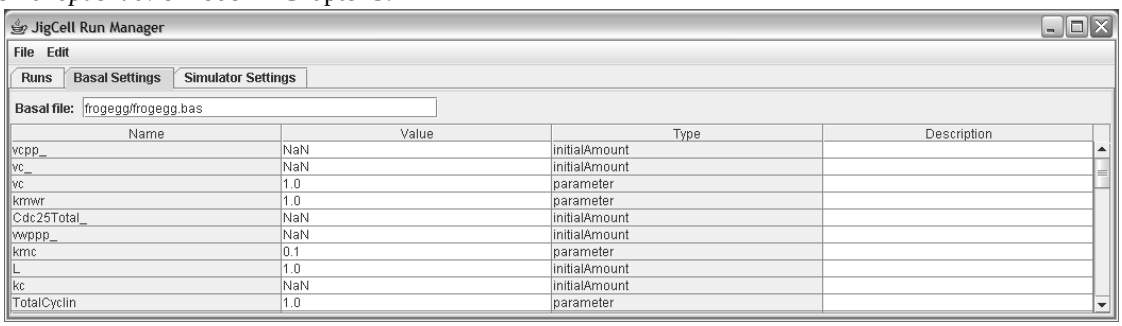
Basal settings

The ‘Basal Settings’ tab is where the modeler enters a default value for each chemical species, parameter, and compartment size in the model. Unlike the experimental condition changes in a run, a basal setting is a numeric value rather than a formula. Preventing the use of formulas in basal settings simplifies the evaluation of changes in a run by the simulation program. Moreover, restricting the basal settings to numeric values matches the restriction when initializing model variables in SBML. Thus, it is possible to substitute basal settings back into an SBML file.

Figure 4.7 shows the basal settings for the run file of Figure 4.6. At the top of the tab there is a field for the modeler to specify where the Run Manager should store the basal settings. Modelers typically have many basal files that they can associate with the model. For example, the process of parameter twiddling is likely to generate many

different basal guesses before the modeler is happy with how the model performs. If the run file directly stored the basal settings, then the modeler would have to edit the run file every time they made a new guess for the basal settings. Since it is likely that the modeler will later discover that the new guess for the model variables is not very good, the modeler will then want to revert to a previous guess for the model variables.

Figure 4.7: Basal settings for the chemical species, parameters, and compartment sizes in the JigCell Run Manager for the *Xenopus laevis* model in Chapter 3.



Reverting to a previous guess seems to require that the modeler make backup copies of the run file, duplicating the ensemble of runs in each copy. There is now a great chance that the modeler will inadvertently edit the definition of a run without making the same change to every copy of the run file. This causes a difficult-to-detect inconsistency. Having the run file reference an external basal set file solves this problem. The modeler can switch between multiple collections of basal settings without modifying the ensemble of runs.

The Run Manager devotes the remainder of the 'Basal Settings' tab to the editing of basal settings. Initially, the Run Manager does not know what variables the model contains. The modeler first must either specify an existing basal set file for the Run Manager to use or import a collection of basal settings from the model.

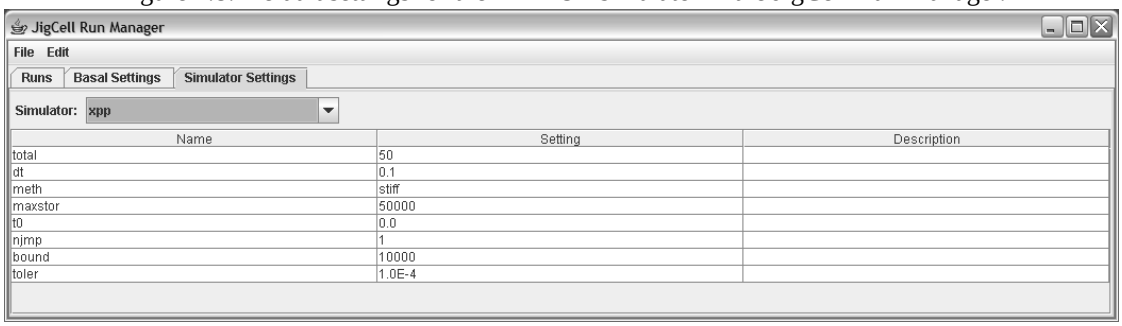
Since few programs currently exist that can create basal set files, and creating a basal set file manually is tedious, modelers typically import the basal settings from the model. Importing the basal settings from the model copies the initial conditions of chemical species, the numeric values of parameters, and the sizes of compartments from the SBML file that the Run Manager has open. There is some ambiguity when the Run Manager imports the initial conditions of chemical species from an SBML file. SBML allows the modeler to specify an initial condition for a chemical species in terms of either concentration or amount of substance. If the modeler did not include an initial condition for a chemical species in the SBML file, then the Run Manager cannot determine which specification the modeler intends to use. Therefore, the Run Manager allows the modeler to distinguish in the basal settings whether a value for a chemical species is an amount or concentration.

Simulator control settings

The 'Simulator Settings' tab, where the modeler configures the control settings of the simulation program, is the final tab in the Run Manager interface. Unlike the other parts of the run file, simulator control settings are independent of the contents of the model. The modeler can select any simulator that the Run Manager has access to through the Simulator API. Figure 4.8 shows the default simulator control settings when the modeler selects the XPPAUT simulator. Section 4.5 describes both the XPPAUT simulator and the Simulator API in detail. JigCell includes the XPPAUT simulator in every installation so the Run Manager always has access to at least one simulation program.

After the modeler selects a simulation program, the Run Manager uses the Simulator API to query the simulator about its known control settings. Each control setting has a name, type, default value, and other information. Although the current Run Manager only uses the name and default value for a simulator control setting in the user interface, future versions of the Run Manager could better make use of the information that comes back from the Simulator API. For example, the Run Manager could use the type of a simulator control setting to build a structured editor that prevents the modeler from entering impermissible values. Having a structured editor would allow the Run Manager to detect that a setting that the modeler chose is invalid before attempting to execute a run.

Figure 4.8: Default settings for the XPPAUT simulator in the JigCell Run Manager.



The Run Manager does not standardize the names of simulator control settings. XPPAUT calls the simulator control setting that indicates the starting time for simulation ‘t0’ and calls the corresponding setting for the ending time of simulation ‘total’. Another simulation program may instead want to use the names ‘tstart’ and ‘tend’ for these simulator control settings. The lack of standardized names makes the task of switching a run file from one simulation program to another more difficult. However, switching between different simulation programs is easier in the current version of the Run Manager than past versions, which required that the modeler specify the simulation program individually for each run. When each run individually chooses a simulation program, the modeler must carefully and tediously examine the run file after making a change to ensure that they configured the runs correctly.

4.4 JigCell Comparator

The most complicated JigCell application is the Comparator. The Comparator is an integrated set of tools for performing quantitative analyses on a collection of data sets. Modelers use the Comparator to perform model testing and evaluation. Tests in the Comparator are assertions about a model or comparisons between model performance and experimental data. A Comparator test performs either model validation, evaluating the operational accuracy of the model, or model verification, checking the accuracy with which other tools transform the model. The modeler must provide three types of information about the desired comparison to create a model test in the Comparator:

- a benchmark value, typically laboratory or experimental data, that the Comparator uses to compare against the model output,
- a data transformation process, the Comparator refers to this as a ‘transform’, that extracts information from the model output,
- and an objective function that measures the distance between the experimental data and the output of the transform.

The output of a transform has the same structural format as the experimental data. For example, if the experimental data is in the form of a time series, then the result of the transform must also have the form of a time series. The result of an objective function when the experimental data and transform output are in perfect agreement is a distance of zero. Successively worse matches between the experimental data and transform output have correspondingly increasing scores. There is no upper bound to an objective function score. The following section describes each of these parts of a comparison in more detail.

Like the Model Builder and Run Manager, the Comparator divides the user interface into a number of screens that correspond to the different parts of a comparison. A spreadsheet representation of that part of the comparison dominates each screen in the Comparator. However, the Comparator divides some elements of the comparison among several tabs. For example, the process of defining the transforms for a comparison uses one tab to specify the transforms and another tab to link transforms with experimental data. This approach introduces navigational issues, the modeler must engage more user interface areas to accomplish a task, but reduces the amount of information on each screen to a manageable amount.

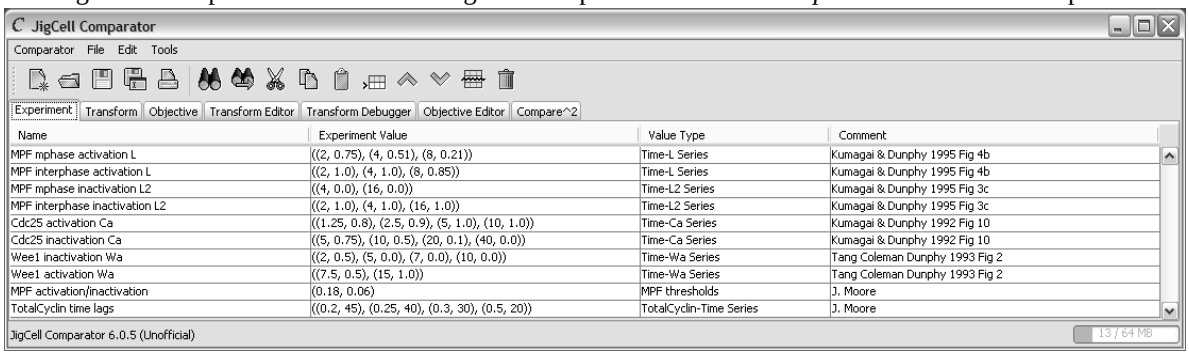
4.4.1 Configuring a Comparison

Before modelers begin working with the Comparator, they first must identify the kinds of available experimental data. JigCell does not provide an application for archiving and searching through scientific literature. This step is part of the process that the revised modeling process identifies as problem formulation. As mentioned earlier, the revised modeling process and JigCell deal with model building activities subsequent to problem formulation.

Modelers can start working with the Comparator after they identify the kinds of experimental data that the model tests use. The first tab that the modeler goes to is the 'Experiment' tab where all experimental data entry takes place. Note that experimental data entry does not require a model. The modeler can begin entering experimental data before even starting the model. Choosing the experimental data and model tests early ensures that the modeler is building a model to pass the tests rather than finding tests to justify the model.

Figure 4.9 shows the experimental data entry tab in the Comparator. Like the other JigCell applications, the Comparator organizes experimental data in a spreadsheet. Each row in the spreadsheet indicates a separate model test. The 'Name' and 'Comment' columns in the experimental data spreadsheet are self-explanatory.

Figure 4.9: Experimental data in the JigCell Comparator from the *Xenopus laevis* model in Chapter 3.



The 'Experiment Value' column is a textual representation of the experimental data for a particular model test. Experimental data in the Comparator uses a 'list of lists' format, which Section 4.5 describes in more detail. Each piece of information in the Comparator is either a scalar value or a list. Lists are indexed collections of scalar values and lists. Thus, lists can nest inside one another to any depth and duplicate most common ways of organizing data.

The experimental data in Figure 4.9 are primarily time series. A time series is a list of measurements taken at particular times. Each measurement is itself a list, with the first entry of the list the time of measurement. The remaining entries are the values of particular species in the model at that time. Time-series data is a built-in data type that the Comparator understands automatically. The Comparator can validate that a given input is a time series, and the Comparator can also display popup editors that display time-series data in a structured format.

The 'Value Type' column is how the modeler communicates the expected type of the experimental data to the Comparator. Clearly, the modeler needs an extensible set of value types. The Comparator can represent experimental data that use custom data types. The modeler can programmatically add new data types that provide the same features for editing and validation as the built-in data types. Section 6.2.1 gives an example of adding a new data type for a real biological problem.

After entering the experimental data for model tests, the modeler then needs to devise a data transformation procedure that produces the equivalent information from a model. Although this sounds like a complicated process, in many cases the modeler can use a simple data transformation procedure. For example, simulation programs typically generate time-series data as output. If the experimental data is a time series, then the data transformation procedure is often to return the simulator output either without change or after applying a filter that eliminates some model variables or time points from the output.

The data transformation procedure for the *Xenopus laevis* model is an example of a simple transform.

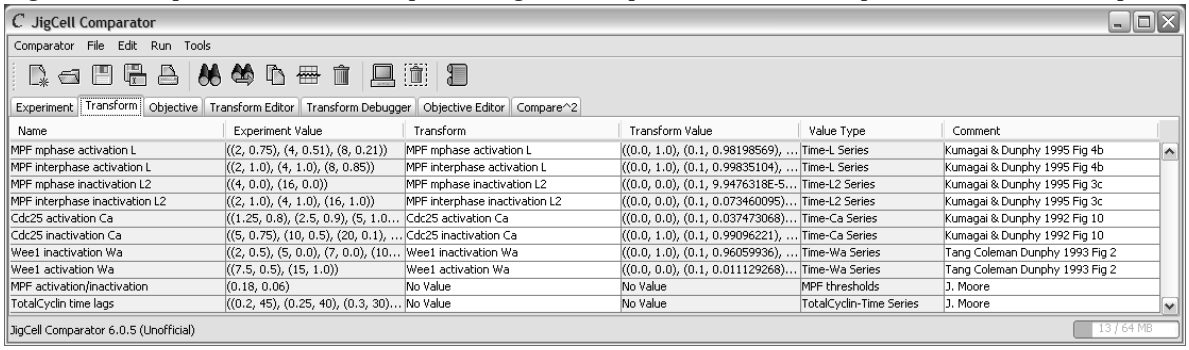
1. Execute a run that the modeler previously created with the Run Manager.

2. Filter all but one chemical species out of the time-course output according to which species the experimentalist measured in the laboratory.

There are two individual steps, 'primitive transforms', in this data transformation process. The example in Chapter 6 includes a complicated transform, built from many primitive transforms, that significantly processes the model output before producing results in a form comparable to the experimental data.

The modeler defines the individual steps of transforms in the 'Transform Editor' tab and connects transforms to model tests in the 'Transform' tab. Figure 4.10 shows the connection between transforms and the experimental data in Figure 4.9. The individual steps of the transforms in this example are of the same simple form that this section described previously. Each model test has a transform whose output is of the same form as the experimental data. Figure 4.10 shows the 'Transform' spreadsheet after the transforms completed execution. Thus, the 'Transform Value' column contains the output of each transform.

Figure 4.10: Experimental model setup in the JigCell Comparator from the *Xenopus laevis* model in Chapter 3.

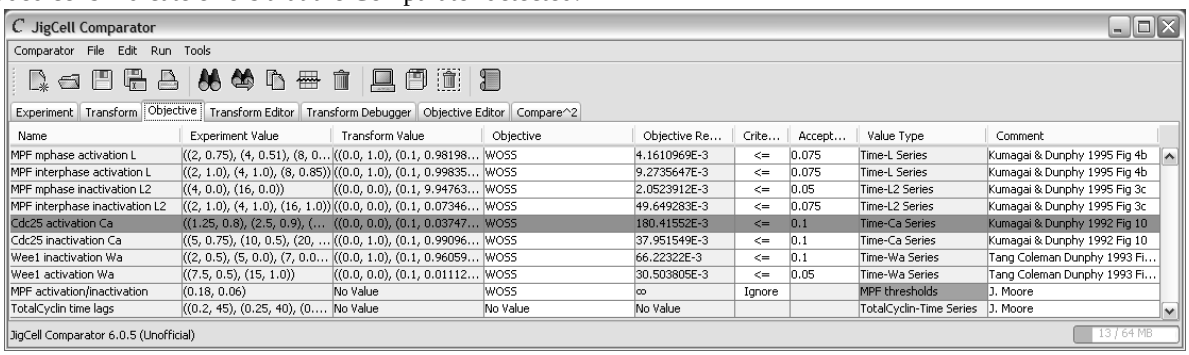


Name	Experiment Value	Transform	Transform Value	Value Type	Comment
MPF mphase activation L	((2, 0.75), (4, 0.51), (8, 0.21))	MPF mphase activation L	((0.0, 1.0), (0.1, 0.98198569), ...	Time-L Series	Kumagai & Dunphy 1995 Fig 4b
MPF interphase activation L	((2, 1.0), (4, 1.0), (8, 0.85))	MPF interphase activation L	((0.0, 1.0), (0.1, 0.99835104), ...	Time-L Series	Kumagai & Dunphy 1995 Fig 4b
MPF mphase inactivation L2	((4, 0.0), (16, 0.0))	MPF mphase inactivation L2	((0.0, 0.0), (0.1, 9.9476318E-5...	Time-L2 Series	Kumagai & Dunphy 1995 Fig 3c
MPF interphase inactivation L2	((2, 1.0), (4, 1.0), (16, 1.0))	MPF interphase inactivation L2	((0.0, 0.0), (0.1, 0.073460095)...	Time-L2 Series	Kumagai & Dunphy 1995 Fig 3c
Cdc25 activation Ca	((1.25, 0.8), (2.5, 0.9), (5, 1.0...	Cdc25 activation Ca	((0.0, 0.0), (0.1, 0.037473068)...	Time-Ca Series	Kumagai & Dunphy 1992 Fig 10
Cdc25 inactivation Ca	((5, 0.75), (10, 0.5), (20, 0.1), ...	Cdc25 inactivation Ca	((0.0, 1.0), (0.1, 0.99096221), ...	Time-Ca Series	Kumagai & Dunphy 1992 Fig 10
Wee1 inactivation Wa	((2, 0.5), (5, 0.0), (7, 0.0), (10...	Wee1 inactivation Wa	((0.0, 1.0), (0.1, 0.96059936), ...	Time-Wa Series	Tang Coleman Dunphy 1993 Fig 2
Wee1 activation Wa	((7.5, 0.5), (15, 1.0))	Wee1 activation Wa	((0.0, 0.0), (0.1, 0.01112926 8)...	Time-Wa Series	Tang Coleman Dunphy 1993 Fig 2
MPF activation/inactivation	(0.18, 0.06)	No Value	No Value	MPF thresholds	J. Moore
TotalCyclin time lags	((0.2, 45), (0.25, 40), (0.3, 30)...	No Value	No Value	TotalCyclin-Time Series	J. Moore

Note that the time-series output of the transform does not attempt to match the times in the experimental data time series. Instead, the transform provides measurement points taken at regular intervals. The objective function for this example is insensitive to the exact times of the measurements. If the objective function required that measurements only appear for the times corresponding to the experimental data, then the transform would need a second filtering step to eliminate the unnecessary points from the time series.

Finally, the modeler defines objective functions in the 'Objective Editor' tab and connects objective functions to model tests in the 'Objective' tab. Figure 4.11 shows the connection between objective functions and the experimental data in Figure 4.9. Like the transforms for the *Xenopus laevis* model, the objective function for this model requires little modeler configuration. This example uses one of the built-in objective functions in the Comparator, the weighted orthogonal sum of squares distance between the experimental data and transform time series.

Figure 4.11: Model evaluation in the JigCell Comparator from the *Xenopus laevis* model in Chapter 3. The darkly shaded cells indicate errors that the Comparator detected.



Name	Experiment Value	Transform Value	Objective	Objective Re...	Crite...	Accept...	Value Type	Comment
MPF mphase activation L	((2, 0.75), (4, 0.51), (8, 0...	((0.0, 1.0), (0.1, 0.98198...	WOSS	4.1610969E-3	<=	0.075	Time-L Series	Kumagai & Dunphy 1995 Fig 4b
MPF interphase activation L	((2, 1.0), (4, 1.0), (8, 0.85))	((0.0, 1.0), (0.1, 0.99835...	WOSS	9.2735647E-3	<=	0.075	Time-L Series	Kumagai & Dunphy 1995 Fig 4b
MPF mphase inactivation L2	((4, 0.0), (16, 0.0))	((0.0, 0.0), (0.1, 9.94763...	WOSS	2.0523912E-3	<=	0.05	Time-L2 Series	Kumagai & Dunphy 1995 Fig 3c
MPF interphase inactivation L2	((2, 1.0), (4, 1.0), (16, 1.0))	((0.0, 0.0), (0.1, 0.07346...	WOSS	49.649283E-3	<=	0.075	Time-L2 Series	Kumagai & Dunphy 1995 Fig 3c
Cdc25 activation Ca	((1.25, 0.8), (2.5, 0.9), (...	((0.0, 0.0), (0.1, 0.03747...	WOSS	180.41552E-3	<=	0.1	Time-Ca Series	Kumagai & Dunphy 1992 Fig 10
Cdc25 inactivation Ca	((5, 0.75), (10, 0.5), (20, ...	((0.0, 1.0), (0.1, 0.99096...	WOSS	37.951549E-3	<=	0.1	Time-Ca Series	Kumagai & Dunphy 1992 Fig 10
Wee1 inactivation Wa	((2, 0.5), (5, 0.0), (7, 0.0...	((0.0, 1.0), (0.1, 0.96059...	WOSS	66.22322E-3	<=	0.1	Time-Wa Series	Tang Coleman Dunphy 1993 Fi...
Wee1 activation Wa	((7.5, 0.5), (15, 1.0))	((0.0, 0.0), (0.1, 0.01112...	WOSS	30.503805E-3	<=	0.05	Time-Wa Series	Tang Coleman Dunphy 1993 Fi...
MPF activation/inactivation	(0.18, 0.06)	No Value	WOSS	∞	Ignore		MPF thresholds	J. Moore
TotalCyclin time lags	((0.2, 45), (0.25, 40), (0....	No Value	No Value	No Value			TotalCyclin-Time Series	J. Moore

The weighted orthogonal sum of squares distance measures the orthogonal distance between the experimental

data points and a curve through the transform data points. The term 'orthogonal' refers to the use of perpendicular distances from an experimental data point to the curve in this method, unlike the least-squares method that uses the vertical distance. Label the m experimental data points $\mathbf{x} = [\mathbf{x}_1 \quad \mathbf{x}_2 \quad \dots \quad \mathbf{x}_m]^T$ and the n transform data points $\mathbf{y} = [\mathbf{y}_1 \quad \mathbf{y}_2 \quad \dots \quad \mathbf{y}_n]^T$. Each data point is itself a vector with q measurements. Then, the weighted orthogonal distance $D(\mathbf{x}, \mathbf{y})$ is

$$D(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{m} e_i * \min_{j=1}^{n} \sum_{k=1}^{q} (d_k * (x_{ik} - y_{jk}))^2. \tag{4.1}$$

The e_i and d_k in Equation 4.1 are weights that the modeler can apply to adjust the relative importance of experimental observations and measurements. However, this example sets each weight to its default value, $e_i = d_k = 1$.

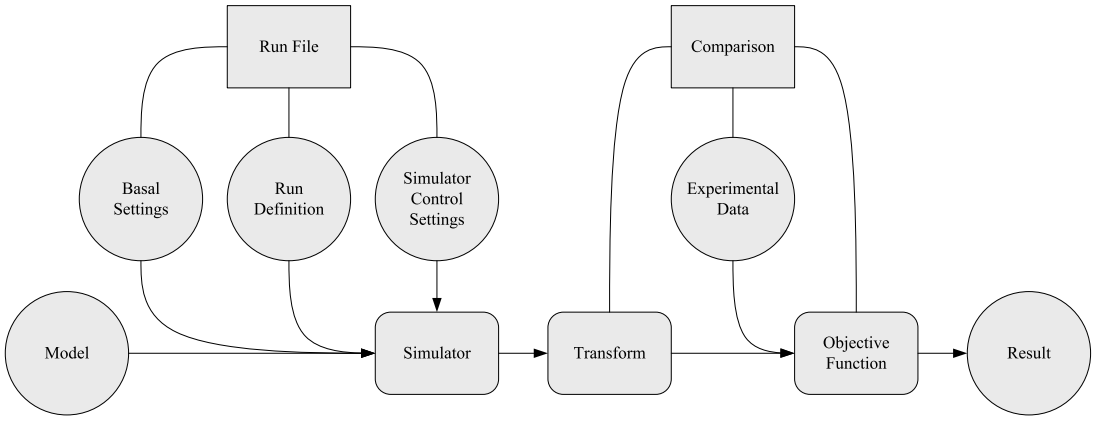
The Comparator then executes each of the model tests. The 'Objective Result' column displays the values that the objective functions compute. An objective-function value is a non-negative real number, with no upper bound. Objective-function values are unscaled. Modelers may consider a value of 1000.0 acceptable for one particular objective function, while they may want to keep the value below 1.0 for another objective function.

The 'Criteria' and 'Acceptable' columns allow the modeler to specify a tolerance for the objective-function value. When the objective-function value does not pass the criteria, then the Comparator flags that model test by highlighting the row. The Comparator also flags other errors in the model configuration. For instance, the Comparator highlights the 'Value Type' column when the model test has experimental data or a transform that does not validate for that data type. Figure 4.11 shows both of these error-reporting mechanisms.

4.4.2 Model Analysis Process

The Comparator synthesizes the information that the modeler entered into the other JigCell tools to create a repeatable analytic process for a model. Figure 4.12 shows the analysis process that many biological models, including the *Xenopus laevis* model, use. The modeler describes the model in the Model Builder or another tool that supports editing SBML. The Run Manager is where the modeler describes how to execute the model. Next, the Comparator is where the modeler describes how to score the model. Finally, the modeler must analyze the results of the comparison.

Figure 4.12: Analysis process for a typical comparison in the JigCell Comparator.



The ultimate interest of the modeler is to determine whether the model meets particular standards of acceptability. The modeler and interested stakeholders work together to set these standards. However, the model output is too vast for the modeler to examine and process the output in a timely manner. Moreover, the model evaluation process is often difficult and tedious to perform repetitively. Modelers turn to software, such as the Comparator, to automate repetitive tasks. Although the analysis process shown in Figure 4.12 automates a single model test, this does not truly solve the problem that the modeler has. Many model tests combine to indicate the acceptability of the model. Automating a single model test requires that the modeler manually collect and consider many separate tests.

The Comparator attempts to alleviate this problem. The modeler can instruct the Comparator to run any number of model tests at once and perform a comparison. The Comparator orients user interface screens so that the modeler conveniently can view and work with a large number of model tests. When the modeler runs a collection of model tests, the Comparator automatically determines the data dependencies, schedules the transforms and objective functions, and eliminates redundant computations to reduce the execution time.

If the modeler has a computer with multiple processors, then the Comparator can perform transforms and objective functions in parallel to further reduce the execution time. Reducing execution time is beneficial to the modeler and improves the user experience of the software. The modeler cannot perform model evaluation in a timely fashion if the model tests take too long to run. However, solely reducing execution time does not solve the model evaluation problem. The model test results are still too voluminous for the modeler to consider all of the model tests and decide whether the model is acceptable. Chapter 6 provides an example that shows this issue. Manual model evaluation and analysis is slow even when significant automation is possible. The modeler time that these processes require limits the number of model tests that the modeler can apply, which limits the size of models that the modeler can develop.

Visualizations are techniques that prevent information overload. As Section 4.4.1 described, the Comparator uses color and highlighting to identify questionable or problematic model test results. The modeler can scan the list of model tests quickly, skipping past the model tests that pass, and only consider the model tests that the Comparator flags as needing attention. Visualization does solve the model evaluation problem for models of a limited scale. However, this visualization approach does not work for models that have tens of thousands of model tests.

One solution for dealing with models that have large numbers of model tests is to use a heuristic approach that combines the results of many model tests into a single score. Section 4.6 describes parameter estimation, which uses this approach. Another solution for models with a large number of model tests is to change the visualization so that a larger number of model tests fit on the screen. In the 'Objective' spreadsheet of the Comparator, each model test occupies a significant portion of the usable screen area. The modeler typically can view no more than fifty model tests on the screen at once, which is insufficient for evaluating models with hundreds of components.

Compare^2 is a Comparator add-on that can fit many more model tests on the screen than the standard Comparator interface. The standard interface of Compare^2 uses the same arrangement of model tests, one per line, as does the Comparator, but eliminates much of the information about the model test except for the objective function result. This additional space allows Compare^2 to display the results of many models side-by-side, a multi-way comparison, for a collection of model tests. The modeler can select the models according to a theme, such as different models for a biological system ordered by the time that the modeler tried that particular model. Compare^2 then allows the modeler to visualize the entire collection of models and determine whether the models become better over time.

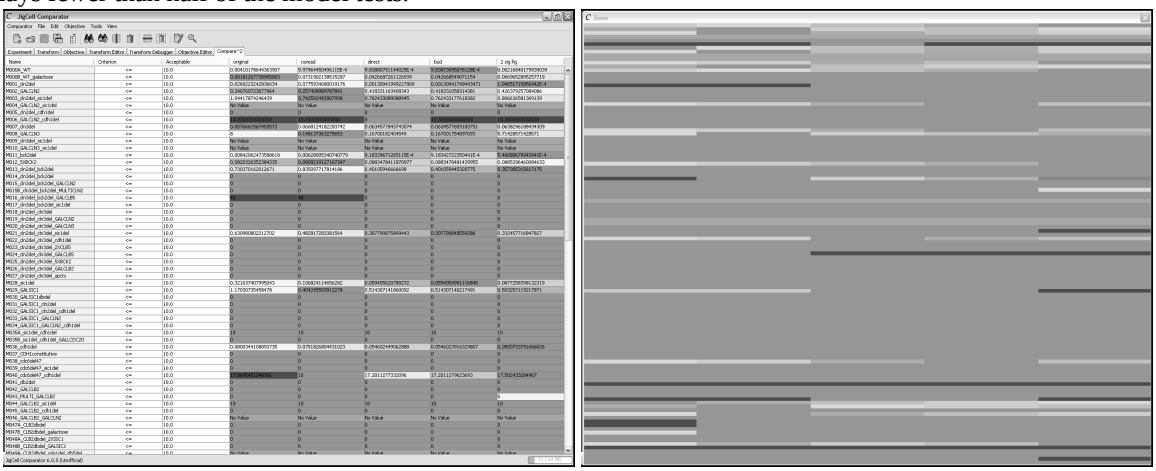
The standard interface of Compare^2 cannot display more lines of text than the regular Comparator interface. Instead, Compare^2 has a second interface that puts all of the model tests and models on the screen at the same time. This 'zoom view' uses the same visualization techniques as the standard view but eliminates all of the textual information, leaving only a graphical indication of model performance. Figure 4.13 shows the standard and zoom view interfaces of Compare^2. In the standard Compare^2 interface, the objective function results use color to indicate model performance. There are several different visualization coloring modes in Compare^2, such as absolute and relative differences from the model test threshold, binning, and historical comparisons against previous models. When the modeler uses the zoom view, the visualization hides all of the textual information about the model tests. The modeler can then spot trends in the color patterns of the model tests and quickly identify changes in model performance.

Compare^2 ties together the results of many comparisons and gives the modeler a visualization method for large numbers of model tests. It is easily feasible to visualize hundreds of model tests for dozens of models at once in the zoom view of Compare^2. As the number of model tests or models increases further, the zoom view produces blends of the colors. Once color blending begins, the modeler no longer can pick out individual objective function results but still can identify general and large-scale patterns in the collection of model tests and models.

4.5 Libraries and Utility Programs

Previous sections alluded to how the applications of JigCell rely on several libraries for reading, writing, and manipulating data. In fact, nearly all of the numerical computation and data processing in JigCell occurs in libraries. The

Figure 4.13: Compare^2 interface showing a multi-way comparison between a collection of models and the zoomed view for the same collection of models. The visualization containing textual information is unreadable even though it displays fewer than half of the model tests.



application code primarily deals with user interface issues, which take up only a small fraction of the total code base. Each JigCell application has a corresponding library for data manipulation:

- the Model Builder uses the SBML Parser,
- the Run Manager uses the Run File Library,
- and the Comparator uses the Comparison Library.

Additionally, JigCell has several standalone command-line tools, simulators, and a Simulator API, which is a library for communicating between simulators and applications.

The utility program that modelers most commonly use in JigCell is SBMLTOODE. SBMLTOODE is a conversion program from the SBML format to the ODE format that Bard Ermentrout developed [51]. As Section 3.1 described, the modelers working with the original modeling process primarily used the XPPAUT program for computational assistance. SBMLTOODE translates the models that JigCell builds so that modelers can continue to use their old tools while JigCell is under development. As SBML is a significantly richer language for expressing models than an ODE file, SBMLTOODE often must use convoluted ways of expressing model features that have no natural representation in XPPAUT. Therefore, the SBMLTOODE tool is not sufficient for using ODE files as an exchange mechanism, and there is no simple mechanism for translating ODE files back into SBML files.

SBML Parser

The SBML Parser is the library in JigCell that is responsible for reading and writing model files in SBML. Marc Vass created the original SBML Parser in 2002; Nicholas Allen created a new library with the same name in 2004. The word ‘parser’ in the name is a misnomer. In addition to parsing SBML, the SBML Parser validates model files, indexes models for searching, computes conservation relations, parses the MathML (Mathematics Markup Language) [16, 119] that SBML uses for mathematical expressions, and converts between infix presentations of a mathematical expression and the MathML presentation.

The simplest portions of the SBML parser are those that deal with reading and writing SBML files. SBML is a markup language based upon the extensible markup language, XML [140]. A large number of libraries for reading XML documents exist, which reduces the involvement of the SBML parser to identifying the semantic meaning of XML elements from their context in the document. Since SBML is a restrictive language, with only a limited number of valid ways to combine structural XML elements, identifying SBML elements from their context is relatively simple.

However, the SBML Parser has a more direct role in the writing of SBML documents. SBML prescribes default values for many of the document elements, and the SBML Parser normalizes the documents that it writes by omitting SBML elements that have their default value.

Model validation and indexing is more complicated than parsing. Although SBML is a syntactically simple language, many syntactically correct SBML documents are meaningless. SBML is a language with internal references between XML elements and restrictions on the contents of fields. The SBML specification further identifies several dozen instances of 'semantic constraints' that the syntax of the language does not reflect. A valid SBML document must satisfy all of these semantic constraints. Validation in the SBML Parser consists of checking that the model satisfies semantic constraints and that the elements that internal references identify exist in the model.

Indexing plays a crucial role both in the validation of SBML documents and directly in the Model Builder application. SBML documents contain many references between XML elements. For example, the containing compartment attribute describes the topology of a compartment. This attribute reference to the containing compartment is solely by identifier. SBML does not restrict a document from referring to a compartment before providing a definition of that compartment. Indexing allows for the rapid identification of SBML elements given an identifier and the possible structure types that might have the identifier, such as chemical species, parameters, or compartments.

The most significant computational task that the SBML Parser performs is the discovery of conservation relations in the model. Although Section 2.4.2 described the process of automatically discovering a set of conservation relations in the model, the SBML Parser also allows the modeler to propose their own set. When the modeler proposes a set of conservation relations, the SBML Parser must check that those conservation relations are valid for the model.

Validating the set of conservation relations that the modeler proposed requires the SBML Parser to perform two additional Gauss-Jordan elimination steps.

1. Check that the number of proposed conservation relations is no greater than the rank of the null space of the stoichiometry matrix,

2. apply Gauss-Jordan elimination to the proposed conservation relations to check that the set of conservation relations is linearly independent,

3. augment the original set of conservation relations with the proposed set of conservation relations,

4. and perform Gauss-Jordan elimination on the augmented matrix.

If the proposed set of conservation relations is valid, then the null space of the augmented matrix has the same rank as the number of proposed conservation relations. This is equivalent to saying that the proposed conservation relations are all linear combinations of the original conservation relations.

Run File Library

The Run File Library is the library in JigCell that is responsible for reading and writing both run and basal files. As Section 4.3 explained, a basal file overrides the parameter values, initial conditions, and compartment sizes in the SBML file, while a run file defines collections of changes to the basal settings. Although the Run File Library needs to provide the same validation and indexing services as the SBML Parser, run and basal files are significantly simpler than SBML files and hence have a correspondingly simpler implementation of these features. Instead, the bulk of the Run File Library consists of routines for creating basal files and executing runs.

When a modeler starts using the Run Manager, they first must create a basal file for their model. The Model Builder permits the modeler to enter numeric values for parameters, chemical species, and compartment sizes, and many modelers choose to do this. Therefore, the Run File Library supports creating basal settings directly from a model. During run execution, the modeler then wishes to perform the reverse process and export the basal settings and changes for a run back into the model.

Executing a run is the most difficult-to-implement service that the Run File Library provides.

1. Start the simulation program that the run file specifies,

2. transmit to the simulation program the default simulator control settings for the run file and the settings for that particular run,
3. apply the algorithm in Table 4.1 to produce an equivalent run that does not use inheritance,
4. replace the references to model elements in the formula for each change of the run with variables,
5. solve the formula for each change using the basal settings,
6. insert the computed values back into the model,
7. and send the new model to the simulation program.

The Run File Library then forwards the time course that the simulation program produces to the requestor.

Table 4.1: Algorithm for flattening a hierarchy of runs starting from the run **start**.

```
Initialize two stacks pending and current.
Initialize a map changes from model element identifiers to formulas.
pending.push start
while pending ≠ ∅
    r ← pending.top
    if r is in current then error, there is an inheritance cycle
    for each run p in r.parents taken from start to end
        pending.push p
    current.push r
    while pending ≠ ∅ and pending.top = current.top
        r ← pending.pop
        current.pop
        add r.changes to changes
```

Comparison Library

The Comparison Library supports the storage and manipulation of data in the JigCell Comparator. Additionally, the Comparison Library provides standard data formats that other applications use for exchanging experimental and time-series data. Unlike the other libraries in JigCell, the Comparison Library is amenable to user extensions. Since the library includes the transforms and objective functions of the Comparator, and modelers wish to supply new transforms and objective functions, it makes sense to provide a framework that the end-user can modify.

Transforms and objective functions make up the majority of code in the Comparison Library. The Comparator provides 18 standard transforms and objective functions for modelers to use. Additionally, the framework for transforms and objective functions in the Comparison Library significantly simplifies the process of extending the Comparator. For example, the Comparison Library automatically handles linking together transforms with other transforms, finding the experimental data that objective functions need, evaluating both transforms and objective functions, and reporting errors during the evaluation process. Furthermore, the Comparison Library organizes the in-memory and external storage of data, implements caching that reduces the number of transform and objective function evaluations during a comparison, and can persist transforms and objective functions to disk even when these objects are user-supplied and have unpredictable storage needs.

The Comparator, and hence the Comparison Library, represents data using the BioSPICE Time Series Format [4]. The name of this format is another misnomer as the format has many applications other then time-series data. An item of data in the Comparator, a 'data element', is either a scalar value or a list. Scalar values have types such as integral, real, or boolean. A list contains an arbitrary number of lists or scalar values, indexed over the positive integers. Thus, lists can nest within lists to any arbitrary depth, which allows data in the Comparator to mimic many commonly used finite and some infinite data structures, such as bags, time series, and matrices.

Efficiently implementing data elements for the Comparator is difficult. Therefore, the Comparison Library provides a number of implementations that make tradeoffs between size, access time, density of the index set, and other factors. The Comparison Library supports converting data elements to and from textual forms, and reading and writing data elements for external storage. Similarly, the Comparison Library also supports many common mathematical operations on data elements, such as taking sub-ranges or slices out of the data.

Simulators and Simulator API

Simulation programs are the remaining area of numerical computation that JigCell includes. JigCell has an in-house implementation of Gillespie's Stochastic Simulation Algorithm, which Section 2.5.1 described. However, the remaining simulation programs that JigCell comes with are third-party programs with many years of use. Obviously, the authors of these simulation programs did not have JigCell in mind when designing their programs, and these simulation programs do not support representing models with SBML. Furthermore, simulation programs often lack a simple way to connect with other applications.

Each simulation program in JigCell has a corresponding wrapper service. A wrapper service translates model files from SBML to the native language of the simulation program, executes the simulation program, and retrieves the results into the BioSPICE Time Series Format. For example, the wrapper service for the XPPAUT simulator translates an SBML model to an ODE file, runs the XPPAUT program, and then parses the output file to create a time series.

This simple description of a wrapper service hides many of the challenges. While mathematical expressions in SBML can make use of a large variety of operators and methods, XPPAUT has a limited mathematical language. Thus, the wrapper service must translate the features that XPPAUT does not supply, such as trigonometric and logical functions. Moreover, XPPAUT does not have structures in its modeling language that are exactly like the structures in SBML, such as events and rules, and has a restrictive naming convention. A wrapper service frequently must rewrite the model extensively during the translation process.

The Simulator API then provides the plumbing between the wrapper service and applications. A consumer of simulation data often relies on only a small repertoire of operations, such as loading a model, changing simulator control settings, configuring output, and generating data. The Simulator API generalizes these operations for time-course oriented simulation. From the perspective of an application, the Simulator API is a simulation program. The application can query the Simulator API about its control settings and can invoke common simulator operations. However, the behavior of the Simulator API changes depending on the currently selected wrapper service. When the modeler selects a different simulation program, the control settings of the Simulator API change to reflect the control settings of the underlying wrapper service. Moreover, the operations of the Simulator API delegate to corresponding operations in the underlying simulator. The Simulator API mechanism allows the JigCell applications to access a wide variety of simulation programs without presupposing which simulation programs the modeler wants to use.

One side benefit of creating the Simulator API was the construction of a simulator-independent test suite in JigCell. Testing a simulator is difficult because each simulation program requires a different format for the test data. Thus, adapting existing test data to a new simulation program is expensive. JigCell employs the Simulator API to use the same model format, SBML, with many different simulation programs. Therefore, JigCell can reuse a standardized suite of test models with any simulation program that the Simulator API makes available and compare the results from simulation program with another.

4.6 Future Software Projects

As Section 4.1 mentioned, JigCell is not complete in the sense of providing all of the services of a typical problem-solving environment. In particular, JigCell does not currently provide effective and transparent access to high-performance computing resources. Moreover, JigCell is not complete in the sense that the revised modeling process, which Figure 4.1 shows applied to JigCell, indicates beneficial tasks that JigCell does not support. These two measures of completeness converge in the first potential new area of application for JigCell, parameter estimation, which several sections alluded to previously.

The second potential new area of application for JigCell is project management. The applications in JigCell produce and consume large amounts of data with varied types. It is often difficult for modelers to organize this data, leading to lost or misplaced data files, or the creation of multiple copies of data that quickly lose synchronization. An application for project management in JigCell would organize the data files that constitute a particular model. The project management tool would then present to the modeler the list of available models and the tasks that JigCell can perform on those models. The JigCell project did not pursue project management extensively, and this dissertation does not consider project management in detail.

Parameter Estimation

While building a model, a modeler sometimes employs a rate constant whose value has no definitive experimental evidence. Furthermore, the rate constant may have a significant range of possible values that the modeler could try. Over time, models accumulate a greater number of these relatively unconstrained rate constraints as modelers generally build models faster than experimentalists complete the corresponding experiments. Section 3.1 described the parameter twiddling process that modelers use to cope with these unconstrained rate constants. Modelers successively experiment with many different values for the rate constants, expending great effort to evaluate the model and identify new feasible regions of the parameter space.

The JigCell Comparator automates the comparison process for evaluating a model. However, the Comparator still relies on the modeler to supply guesses for the parameter values. Without automated fitting, the modeler must calibrate the model parameters by manually searching for valid and optimal regions of the parameter space. Parameter estimation is a class of techniques that supply a guess for the next parameter vector to try with the model.

A parameter estimator explores parameter space while trying to minimize an objective function. The Comparator allows modelers to define arbitrarily complex objective functions that run as programs during model evaluation. Few general-purpose techniques accommodate such objective functions. Most parameter estimation procedures are applicable only to simple curve fitting, where the experimental data are time series and the model output needs minimal or no data transformation. If the modeler uses the transforms and objective functions from the Comparator, then the data are typically not a direct solution to the differential equations of the model but rather a complicated, nonlinear functional of the differential equation solution. Furthermore, these functionals involve both dependent and independent variables that are subject to experimental error.

Figure 4.1 included two stages, the **score** and **report** stages, that distinctly belong to parameter estimation. In the revised modeling process, these stages are a subset of the **evaluate** and **repair** stages. The **score** and **report** stages have a separate representation in JigCell because of the impact that these stages have on the model development process. The **score** stage defines an algorithm that determines whether one parameter vector produces a more acceptable fit than another for the model. The algorithm requires experimental data, an executable model, the acceptable range of parameter values for the executable model, and a user-defined objective function. This setup typically comes directly from the experimental model that the modeler builds in the Comparator. The **report** stage injects the fitted parameter values back into the modeling process for study and testing.

Local parameter estimation algorithms

A local parameter estimator takes measured steps from the current parameter vector based on objective function evaluations. Although this makes local parameter estimators very fast at homing in on nearby, good parameter vectors when the objective function is smooth, there are several drawbacks to local parameter estimation. First, local parameter estimators have a hard time finding distant, good parameter vectors. A local parameter estimator can fall into wells, or local minima, of the objective function, and escaping from these wells is difficult. Second, the user-defined evaluation procedure in the Comparator rarely leads to a smooth objective function unless the modeler takes some care when defining the evaluation process. In circumstances where these obstacles do not apply though, local parameter estimation is typically worthwhile.

ODRPACK (Orthogonal Distance Regression PACKage) [33, 34, 35] is a mathematical software system that performs local parameter estimation. ODRPACK uses a trust region Levenberg-Marquardt method. The Levenberg-Marquardt method starts with the steepest descent method and smoothly changes to Newton's method when approaching

the solution. The trust-region implementation of the Levenberg-Marquardt method determines the step size using a confidence measure of the local model of the objective-function value. At each step, ODRPACK compares the expected improvement in the objective-function value for taking the step with the actual improvement. The agreement between the expected and actual improvements in the objective-function value indicates whether ODRPACK should expand or contract the trust-region radius.

The use of a local model in ODRPACK causes slow convergence if the objective function is not differentiable near the optimum solution. The expected improvement on which ODRPACK bases parameter vector steps is essentially an estimate of the derivative. Step functions in the objective-function value can appear when matching categorical observations, such as whether a mutant strain of an organism thrives for a period. However, for a correct model of a biological system, it is unlikely that a step in the objective value is directly next to an optimum solution. If the true parameter vector had a nearby discontinuity, then the organism would show sensitivity to minute environmental changes. Extreme sensitivity to the environment is unfavorable for survival.

ODRPACK does not assume that all of the measurement errors are in the dependent variables [39]. Instead, ODRPACK seeks to minimize the weighted sum of orthogonal distances between the model output and the experimental data. The weighting factors scale the residuals of the fit and express the confidence that the modeler has in the reliability of a particular experimental observation. The output of ODRPACK gives a locally-optimal parameter vector and a measure of the goodness-of-fit of that parameter vector.

Global parameter estimation algorithms

A global parameter estimator, in contrast with local parameter estimation, eventually looks in every neighborhood in a finite-dimensional parameter space. Global parameter estimators use a systematic method for searching parameter space, sometimes backtracking even when searching a new region of parameter space leads to higher objective-function values. A parameter-estimation method that successively sampled points along a space-filling curve that winds throughout parameter space meets the criteria for global parameter estimation. However, such a scheme is not an efficient way to search parameter space. Good, global parameter-estimation methods seek to maximize the amount of information that they derive from each objective-function evaluation.

DIRECT (DIviding RECTangles) [65], is a variant of Lipschitzian methods for constrained global optimization. DIRECT searches for the minimum value of an objective function, $\min_{x \in D} f(x)$, inside a closed region $D = \{x \in E^n | \ \ell \leq x \leq \mu\}$ whose boundary consists of simple planes. There is an assumption made here that the objective function is Lipschitz continuous on D, satisfying $|f(x_1) - f(x_2)| \leq L\|x_1 - x_2\|$ for all $x_1, x_2 \in D$. This assumption does not necessarily apply to the arbitrary objective functions that the modeler can build in the Comparator. When the objective function does not satisfy this assumption, DIRECT will still eventually terminate with the global minimum value of the objective function. In this case, however, DIRECT can no longer guarantee that it searches for the global minimum efficiently.

The Lipschitz optimization method has had many practical applications in science and engineering. Unlike some other methods for global parameter estimation, the Lipschitz method requires that the modeler set few control parameters and does not rely on approximating derivatives or other more analytical information about the system. However, the Lipschitz constant of a particular objective function is often unknown and is difficult to estimate. The DIRECT method converges to the global optimum without requiring knowledge of the Lipschitz constant [75].

The DIRECT algorithm takes its name from the key step of dividing rectangles, although mathematical literature more commonly refers to these rectangles as boxes. DIRECT is a pattern-search method, which takes moves based on the objective-function values that it observes at a pattern of points. The points for sampling the objective function are the centers of the boxes. Center sampling is generally advantageous to corner sampling when the number of parameters, and hence the number of dimensions in the parameter space, is large. Most large-scale biological problems have a correspondingly large number of unconstrained parameters.

A box in the DIRECT algorithm is potentially optimal if there exists a value of the Lipschitz constant for which that box is the most likely to contain the global minimum. There is a simple algorithm for finding the potentially optimal boxes. If viewed as a scatter-plot, with the two dimensions of the plot the size and objective-function value of the boxes, then the potentially optimal boxes are the lower-right boundary of the convex hull. Each iteration of the DIRECT algorithm subdivides all of the potentially optimal boxes. DIRECT can operate in an 'exploratory mode',

which emphasizes searching untested boxes, or in an ‘exploitation mode’, which emphasizes searching boxes with better objective-function values. The modeler uses a control parameter to bias between the two modes of DIRECT.

DIRECT is relatively robust to noise in the objective-function values [65]. The amount of noise in the objective-function values limits how quickly DIRECT converges to the minimum. This insensitivity to noise makes DIRECT suitable for application to stochastic models, which intentionally introduce noise into the model and model-evaluation process. Current implementations of DIRECT cannot handle integer variables or constraints on the parameter space other than simple bound constraints. Furthermore, DIRECT is relatively inefficient for finding an accurate minimum value of the objective function. Instead, it is best to run DIRECT in the exploration mode and then use some local parameter estimation technique to find the true minimum from a collection of candidate starting points.

Chapter 5

Software Requirements for the JigCell Modeling Environment

During development of the revised modeling process in Section 3.3, modelers reported four distinct areas of concern that they have about the original modeling process. These four areas are support for model documentation, testing, standardization, and automation in the modeling process. The JigCell modeling environment that Chapter 4 introduced is an implementation of the revised modeling process, and it is possible to test JigCell for its support of these concerns.

The goal of documentation is to record critical information about the modeling process. Modelers need model documentation at each modeling process stage. The creation of model documentation is critical for model accreditation and for planning future modeling tasks. The documentation goals of the revised modeling process are to record

- the model each time the modeler transforms the description of the model,
- the procedures that the modeler used for testing, to support automated model testing and review of verification, validation, and testing methods,
- and the results of model testing, for comparison against future model tests and for presentation to decision makers during model accreditation.

The primary model testing technique that the biological modelers in Chapter 3 used for model validation is the comparison of model output against historically collected experimental data. However, there is a limited quantity and quality of available experimental data for a particular model. Moreover, conducting new laboratory experiments for model testing or expansion of a model is a significant expense. The time that modelers spend performing model validation is cheap in comparison. The JigCell modeling environment emphasizes verification during model construction to prevent the introduction of errors that strain the limited testing resources of the modeler.

The goals of the area of model testing are to introduce model verification, validation, and testing activities into the modeling process early and to continuously monitor the model for introduced errors. Balci [17] codifies several indicators of model credibility, which automated modeling tools can perform continuously during model development. The model-testing goals of the revised modeling process are to verify

- that the graph structure of the wiring diagram corresponds to the conceptual idea that the biologist has in mind,
- that the modeler is using the names of chemical species, kinetic rate laws, and parameters consistently across the model,
- and that the simulator has all of the information about the model and execution environment that it requires and can execute the model properly.

Independent model verification and validation are testing activities by someone other than the original model developer or stakeholders that improve the quality of the model [14]. Independent model testing reduces the potential

for modeler bias in evaluation, promotes the earlier detection of errors, and reduces the cost of error detection and correction. Introducing independence into the modeling process at each iterative cycle, with the goal of supporting independence for all testing activities, requires significant advances in the four areas of concern that this section outlined previously. Improving the modeling process in these areas should ultimately lead to increased rates of model accreditation and acceptance by decision makers. The JigCell modeling environment has the model-testing goal that an agent independent of the original model design or specification teams can perform model verification, validation, and testing from a plan that the model developer recorded previously.

The biological modelers in Chapter 3 use wiring diagrams to communicate their models initially. Unfortunately, no universal standards exist for the graphical language of wiring diagrams, although some representations, such as those of Kohn [80], are increasingly popular. As Chapter 4 mentioned, the pathway modeling community currently is standardizing SBML [71, 72], which is an XML-based representation of biological models at the level of chemical reactions. While SBML applies to wiring diagrams in the sense that model editing tools can convert between SBML and wiring diagram representations, modelers should not directly edit SBML files. The main purpose of SBML is to facilitate model exchange between modeling groups, who then use their own preferred modeling tools to load the model. The JigCell modeling environment currently supports SBML Level 2 Version 1 [54].

The goal of the area of standardization is to adopt

- uniform notations that make model communication easier,
- and uniform processes that make model development easier.

Furthermore, each stage of the revised modeling process has domain-specific information that modeling tools can use to construct uniform sequences of tasks for that stage. Applying uniform processes can prevent some types of errors in the planning stage of the model and can reduce the chance of modelers applying incorrect modeling techniques.

Much of the original modeling process that Section 3.1 described consists of work that modelers perform repeatedly. The goal of the area of automation is for the computer to perform some of these repetitive tasks and speed up other tasks substantially. Verification, validation, and testing activities are automatable throughout the modeling process. Using automated tools that support these activities can reduce significantly the time and effort for model testing [21].

The biological modelers that use JigCell repeatedly modify parameters and initial conditions. After each modification, these modelers then compare the revised model against experimental data. The automation goals of the revised modeling process are to perform

- regression testing as frequently as possible, to ensure that each model transformation maintains the quality of the model,
- testing activities from previous model development cycles, to check that the model is still acceptable,
- numerous and specific testing activities, so that after the modeler introduces an error, the modeler can then identify the location of the error in the revised modeling process and in the model,
- and testing activities automatically while the user is modifying the model, giving feedback on the performance of the modifications.

Additionally, biological models can repeat operations across multiple chemical reactions or incorporate other models as subcomponents. Sufficient support for both model standardization and automation would allow the use of well-tested, black-box components to implement both repeated operations and model fragments.

The focus of this chapter is to develop standards for measuring the software applications in the JigCell modeling environment. Chapter 3 described how modelers previously developed models using the original modeling process and introduced the revised modeling process. Chapter 4 introduced the JigCell modeling environment, which is an implementation of the revised modeling process. The present chapter develops standards for measuring the efficacy of the JigCell modeling environment and how well JigCell matches the revised modeling process. The standards for measuring the software applications come from user interviews (user requirements), the modeling methodology that Section 3.2 described (methodological requirements), and from domain experience (domain requirements).

This chapter starts with an overview of four major categories of requirements to support: documentation, model testing, standardization, and automation. These categories of requirements then expand into methodological requirements that correspond with ideas in the conical methodology that Section 3.2 presented. Section 5.1 and Section 5.2 explore the performance of model verification, validation, and testing from the perspective of a domain expert. These sections describe the success that the JigCell modeling environment has had with each of these techniques for supporting experts in the domain of biological modeling. Section 5.3 introduces the process of collecting user requirements and explains how this dissertation performs requirements testing. The remainder of the chapter describes the user requirements and the results of requirements testing for each of the major JigCell applications.

Contents

5.1 Building Modeling Software for Domain Experts

The successful construction of a model requires combining expertise in the problem domain with expertise in the practice of modeling and simulation. Many domain experts that build models have at least some background knowledge of modeling and simulation. Modelers gain this knowledge somewhat through formal learning but mostly from the experience of building models in the problem domain. Although domain experts are often not modeling specialists, they can successfully complete modeling projects by using the right tools. Developers build software programs from the knowledge and experiences of modeling specialists, creating modeling tools that assist with modeling and simulation tasks. A domain expert draws upon these resources by using modeling tools to solve a problem in their domain.

Domain experts often do not perform sufficient model verification, validation, and testing due to a lack of accessible support for these activities in modeling tools. An increase in model verification, validation, and testing would improve the likelihood of success for modeling efforts. The target audience for general purpose modeling tools is often the modeling specialist. A domain expert might find a general purpose modeling tool inadequate because of assumptions that the tool makes for the target user community, such as the languages that the tool uses for expressing models and modeling concepts, support for features that are not relevant to the problem domain of the expert, and a lack of modeling guidance for the tool user.

User personas

A modeling tool is unlikely to accommodate domain expert users unless the tool builders explicitly include domain experts as a user class. The modeling tools that this dissertation discusses specialize to experts in the particular domain of biological modeling. Moreover, the developers of the JigCell modeling environment had direct access to biological modelers for study. Suppose that the developers of a modeling tool did not have an appropriate domain expert available. If a software developer does not have actual user representatives, then it is helpful to construct personas that typify how a user class might use the software product [82].

Software developers can employ user personas when considering how development decisions affect users of the modeling tool. Although the developers of the JigCell modeling environment had access to biological modelers, these biological modelers did not always know how to perform a particular modeling task. User personas of domain experts performing model verification, validation, and testing can serve as substitutes in this situation.

The following text gives a small sample of scenarios that demonstrate domain experts performing verification, validation, and testing on reaction-oriented models.

Scenario 1: The domain expert wants to use a model that someone else created previously and needs to check that the model is suitable.

Abel is a biochemist who is trying to understand signaling processes in a tissue culture. Abel researched the literature for suitable models and obtained electronic versions of several models that appear interesting. The documentation for the models does not include enough details about their validation and testing procedures, so Abel wants to check that the performance of the models is acceptable for the intended application. Abel knows that integrating multiple models together is error prone. Therefore, Abel needs to verify that he is integrating the models correctly.

Scenario 2: The domain expert is acting as an independent agent to perform verification and validation activities on a model under development.

Joan is a process engineer with experience in high-rate microbial conversion. A pharmaceutical manufacturer is using a simulation study to reduce the cycle time of one of their products. The manufacturer hired Joan to help validate the models that they are creating. Joan will perform face validation (a review of simulation output for reasonableness) and participate in model reviews and walkthroughs. Joan needs to understand how the models from the manufacturer work and test the models by performing simulation experiments.

Scenario 3: The domain expert is developing an original model without the assistance of a modeling specialist.

Steven is a physicist working on a new theory of particle interaction. It is difficult to predict some of the consequences of his theory, so Steven decided to use a discrete event model to test his intuition. Steven decided to build the model himself rather than hiring a modeling specialist because this is a side project, with little budget. The outcome of the simulation is a little different than Steven expected, but now he is unsure if the error is in his model or in his intuition. Steven may have a variety of other reasons for building the model himself, including

- cost of employing specialist,
- difficulty of finding specialist,
- value of employing specialist not understood,
- model turnaround time,
- confidentiality or secrecy of information,
- unable to transfer domain knowledge or requirements,
- modeling objectives in flux,
- and expert learning from model behavior or construction process.

These three scenarios describe domain experts that are performing model verification, validation, and testing. However, the domain experts each have different modeling and simulation goals. Abel is attempting to reuse an existing model. Joan is part of a larger simulation study with a particular business objective. Steven is constructing a new model. The scenarios for using a modeling tool determine the techniques that help support the domain expert. Therefore, it is important that the developers of a modeling tool consider how domain experts will use the tool when deciding whether to incorporate a particular support technique.

Pitfalls

Although it seems clear that keeping domain experts in mind when building modeling tools is beneficial to the modeling community, including this support does have cost. The most obvious cost of building modeling tools that support domain experts is the cost of designing, building, and maintaining these tools. Including domain experts as a targeted user class may require sacrifices in quality, timeliness, efficiency, reliability, robustness, testability, and usability for other user classes. These costs and tradeoffs make it important for tool developers to consider how, and why, domain experts would want to use a modeling tool.

A domain expert that acts without sufficient modeling and simulation guidance also incurs cost in the form of the risk of failure. Modeling projects that fail have opportunity costs in addition to the expense of the project. Starting

the modeling process over again drains resources from other worthwhile activities and delays the return on investment of using modeling and simulation. Moreover, producing an incorrect model is costly as the detection of model errors through operational use is difficult and time-consuming. Meanwhile, decision makers may spend money and credibility on the results of an invalid model.

Arthur and Nance [14] emphatically conclude that independent model verification, validation, and testing is an important technique for mitigating risk in model development. Reducing the risk of failure lowers the expected costs of model development, use, and maintenance. Moreover, modelers can expect that incorporating independence into the modeling process improves model quality and operational correctness. A domain expert who is developing a model might find that employing outside help for model verification, validation, and testing is cost effective. Employing outside help is particularly effective if the model is of a "critical" nature, has a high cost of failure, or has a cost for error detection and maintenance that exceeds the cost of independent model verification and validation. However, the use of independent model verification, validation, and testing does not preclude the need for modeling tools that support domain experts. The independent agent may also be a domain expert instead of a modeling specialist.

When not employing a modeling specialist, the domain expert has an increased risk of selecting an inappropriate modeling and simulation technique. Although modeling tools provide guidance about which techniques to employ, the tool user must ultimately choose how to develop the model. When modelers use an inappropriate technique, they can introduce difficult-to-detect model errors. Correcting these errors may require discarding some of the work done on the model, which is an expense of time and money that can lead to the failure of the modeling project.

5.2 Domain Support for Model Verification, Validation, and Testing

There are many techniques for better adapting modeling tools to the needs of a domain expert. This section focuses on techniques that aid the performance of model verification, validation, and testing by domain experts. Table 5.1 summarizes the techniques that this dissertation includes and rates the degree of support and impact of the technique in the JigCell modeling environment. Table 5.1 does not attempt to rate the impact of techniques for which the degree of support is not measurable. This dissertation does not attempt to provide a comprehensive list of model verification, validation, and testing techniques for supporting domain experts. This dissertation presents the techniques that simulation studies [6, 9] previously tried for the JigCell modeling environment. Many modeling tools other than JigCell employ these techniques, with a positive effect towards supporting the domain expert.

Table 5.1: Summary of degree of support, impact, and applicability to other problem domains for techniques that support a domain expert in model verification, validation, and testing activities (L = Low, M = Medium, H = High).

Technique	Degree of support	Impact	Applicability
Using domain terms and concepts for models	H	H	H
Using domain terms and concepts for modeling	H	M	M
Structuring data and validating data entry	M	H	M
Integrating modeling tools into an environment	M	M	M
Maintaining a model test plan	M	M	H
Actively monitoring model quality	L	L	L
Diagnosing model errors with a knowledge base	–	–	L
Keeping historical records of model development and testing	L	M	H
Presenting multiple visualizations of models and model outputs	M	L	M

Table 5.1 repeats the applicability rating for each technique to problem domains other than reaction-network modeling from Allen, Shaffer, and Watson [11]. There are problem domains for which these ratings are less accurate. For example, the use of knowledge bases for diagnosing model errors has an applicability rating of 'low' because there are many problem domains that have too little history to develop reasonable classifications of past domain expert experiences. Modeling tools that support a narrowly-defined problem domain with a large and detailed historical record would find that implementing this technique is more effective.

Using domain terms and concepts for models

For decades, modelers have recognized the need for custom languages for modeling and simulation [47]. Using a language designed for modeling and simulation reduces the amount of information that a modeler needs to enter as compared to a general-purpose programming language. Moreover, raising the level of abstraction of a language removes some of the error-prone and resource-intensive mappings that modelers must employ to translate their ideas into the language. Many modeling languages are for general-purpose modeling, although some specialize to a particular modeling and simulation technique, such as discrete event simulation [138].

Introducing domain concepts into a modeling language to produce a domain-specific modeling language is a powerful technique for making models more accessible to the domain expert. A domain-specific modeling language allows domain experts to better understand, modify, develop, and test programs written in the language [134]. Testing and debugging a model requires an in-depth understanding of the construction and behavior of the model. Using a domain-specific modeling language makes the structure and function of the model more readily apparent.

Building a domain-specific modeling language requires selecting a problem domain, gathering knowledge about the domain, and reducing that knowledge to semantic objects and operations. Selecting a problem domain involves making a tradeoff between the focus and size of the language. A language that represents a large domain or scope can only weakly specialize to any particular aspect of the domain. In contrast, a language that tightly focuses on a small domain may have a limited number of interested users. Developing a domain-specific modeling language often requires several iterations between prototyping the language and having experts in the targeted domain give feedback about the applicability and ease-of-use of the language [77].

There are many forms of domain-specific modeling languages, such as textual languages, graphical languages [25], spreadsheet languages, or other forms convenient for the domain expert. Language designers construct the language by hand or by using meta-modeling tools [2]. It is difficult to construct a domain-specific modeling language that is both easy for a modeler to write and easy for a computer to process. Languages that are difficult for a modeler to write require modeling tools.

CellML [45] and SBML [72] are domain-specific modeling languages for biochemical models. Both languages are structured, textual languages that support the description of models using domain concepts, such as molecular counts, chemical reactions, and cellular compartments. The development of SBML is a good example of the tradeoff between focus and size in a domain-specific modeling language. There is continuous pressure to directly support additional types of models in SBML, such as flux-balance models. However, the committee responsible for designing SBML must struggle with the problem of adding this support without placing an undue burden on the tool builders that implement the language. There is also concern that making the language too broad will lead to communities using disjoint subsets of the language, with no real communication between these groups. As SBML becomes larger and more complicated, understanding models written in the language becomes harder.

The JigCell modeling environment uses SBML to represent models of biochemical reaction networks. As Section 3.1 described, the observed biological modelers used the ODE file format of G. Bard Ermentrout. However, the ODE file format stores only the differential equations, omitting the biological significance of these equations.

SBML stores both the mathematical and biological information about the model, giving domain experts a better understanding of why the model uses a particular differential equation. Previously, biological modelers primarily developed models in computer code and then converted their models to text, figures, and equations before publication. This conversion frequently introduces errors that impede the replication of results and further development of a published model [85]. Domain experts benefit from the use of a domain-specific modeling language by having access to the original model description in a more readily apparent form. Additionally, SBML is a standardized format, which allows other modeling tools to read and understand the models.

Using domain terms and concepts for modeling

Along with customizing the modeling language, it is also desirable to use the language of the domain expert in modeling tools. When using a domain-specific modeling language, model editing tools naturally adopt domain terms as the easiest way to express the model to the user. However, the domain-specific modeling language is unlikely to describe the inputs and actions of model testing tools. Domain-specific modeling languages typically describe

models rather than activities done with a model, although problem-solving environments are specifically intended to do both [6, 137]. Modeling tool developers may need to elicit the preferred terminology from the domain expert.

The model and modeling tools must accommodate the types of experiments that the domain expert wishes to perform on the model. Valentin and Verbraeck [133] present guidelines for creating a domain-specific modeling language that include consideration of this issue. Using domain terms and concepts in modeling tools aids the domain expert in matching the functionality of the tool to their needs.

The JigCell modeling environment uses domain terms and concepts to provide a better user experience. Users found confusing the terminology that early versions of JigCell used for modeling activities. In particular, JigCell used modeling and simulation terms, such as 'experimental model', that were too similar to terms that biological modelers used for domain activities, causing conflicts. Renaming the terms that the JigCell applications displayed in their user interfaces improved user understanding of the capabilities of the modeling tools. Additionally, the modelers using JigCell reported that they more readily found features in the applications when the sequence of steps for a particular modeling activity was analogous to the original modeling process. JigCell restructured some of its modeling activities to better resemble the original modeling process, which made learning how to use the applications faster.

Structuring data and validating data entry

Desk checking, inspections, reviews, and walkthroughs are model verification and validation methods that require careful scrutiny of the model by the domain expert. Syntactic and typographic errors are a distraction during these model-testing activities and can potentially hide serious errors. After the modelers fix the syntactic and typographic errors, they must retest the model to make sure that they corrected the identified errors without introducing new errors. The prevention or early detection of these types of errors reduces the testing burden of the domain expert. After the modeler corrects all of the superficial errors, the domain expert can test the model for more fundamental errors.

Two important techniques for preventing the introduction of syntactic and typographic errors are the use of validating and structured data editors. Validating editors check that user input is reasonable before applying it to the model. Structured editors break the task of entering data into a predefined collection of attributes, values, and relations [55, 70]. A structured editor changes the organization of data from text to a more communicative form. Schank and Hamel [126] indicate that the use of structured editors has the additional benefit of making model modification more accessible to the domain expert. Since it is difficult to validate input unless the modeling tool specifically defines the class of valid inputs, validating editors are frequently also structured editors.

The spreadsheet interfaces of the JigCell modeling environment are examples of structured editors for biological models. Each column in one of the spreadsheets represents a particular type of information about the model. A domain expert using the JigCell Model Builder does not need to learn the syntax of SBML before building a model. However, many of the fields in the user interfaces of the JigCell applications perform only minimal validation on the user input. For example, the JigCell Model Builder checks that the name of a unit of measurement is legal but does not check that a unit of measurement with that name exists in the model.

Integrating modeling tools into an environment

Integrated modeling environments supply tools that support multiple parts of the modeling and simulation lifecycle. An integrated environment also supplies a modeling methodology by selecting a particular set of tools and controlling how modelers use those tools in concert [27]. Comprehensive modeling environments reduce the need to locate a modeling tool for a particular activity. Carefully selecting the available tools in a comprehensive environment also reduces the risk of the domain expert using an inappropriate modeling and simulation technique. Errors from using an inappropriate technique are normally difficult to detect and correct. Cohesive environments have well-tested exchange of model information between tools, reducing the loss of fidelity when the model is transformed.

The integration of tools in a modeling environment is a continuum from loose to tight coupling. Loosely coupled modeling environments are easier to make comprehensive. Tightly coupled modeling environments are easier to make cohesive. The Simulation Model Development Environment [20] is an example of a modeling environment that the developers coupled loosely and designed for comprehensive coverage of the modeling and simulation lifecycle. The Simulation Model Development Environment supports the conical methodology by selecting tools appropriate for each

step of that methodology, including tools for model generation, translation, verification, analysis, and management. Modelers can specialize the environment by adding new tools for themselves or their domain of expertise. As a research environment, the Simulation Model Development Environment uses loose coupling to ease the creation and testing of prototype modeling tools.

Integrating modeling tools is a requirement for integrating the model verification and validation functions of those tools [40]. Having integrated verification and validation is important because it improves the coverage of model verification, validation, and testing activities along the model lifecycle. As modelers move from tool to tool, they never lose the ability to evaluate model quality. The ability to transfer model testing information between modeling tools also reduces the startup costs of using the model verification, validation, and testing capabilities of a tool.

The JigCell modeling environment is an integrated environment for building models of biochemical reaction networks. JigCell is comprehensive with respect to the revised modeling process of Section 3.3. However, the JigCell modeling environment does not tightly couple its applications and is not cohesive. Although one JigCell application can read the model data that another one of the applications created, the modeler has no effective way to move between the applications. Moreover, the applications of JigCell frequently discard the results of model testing, preventing other applications from examining the model tests that the application performed and the results of those tests.

Maintaining a model test plan

The core of a model test plan is a repeatable collection of scenarios for model testing. A scenario, called a test case, describes a sequence of actions to perform on the model and an expected outcome for each action. The term 'failure' for a model test indicates a problem in the model rather than the model test. If the model contains stochastic components, then the description of the expected outcome is complex. Non-deterministic models can produce many equally correct outputs from a single set of inputs. When this occurs, the model test must treat the observations of model behavior as coming from a sample space and statistically analyze the results.

Documenting that a model passes its test plan and justifying why these test cases demonstrate that the model is suitable for a particular purpose improves model credibility. By maintaining a model test plan, the next user of the model benefits from the body of evidence that the modeler developed during model accreditation [43]. Balci et al. [23] give an organization for a formal and comprehensive plan of testing and accreditation. This level of detail is not necessary for all uses of a model. However, it is instructive to review the types of information that the modeler can collect during model testing.

Maintaining a model test plan also assists the domain expert in developing the model. During the model development process, a modeler iterates between refinement and evaluation of the model. Without a repeatable test plan, model verification, validation, and testing is a scatter-shot approach that is unlikely to add significant value. Modelers can measure their progress during model development by using modeling tools that support test plans.

The JigCell modeling environment provides uneven support for building model test plans. Although the JigCell Comparator, which specializes in performing model verification, validation, and testing, almost entirely focuses on building test plans, the other JigCell applications provide minimal support for this technique. A modeler using the JigCell Model Builder or Run Manager has no means of recording the tests that they perform on the model or the results of those tests. Although the JigCell applications perform some model tests automatically, there is no way to identify the version of the applications that the modeler used or whether the model passed the tests.

Actively monitoring model quality

A modeler must perform model verification, validation, and testing throughout the model lifecycle. The early detection and correction of errors reduces the total cost of producing a correct model. When the modeler leaves errors uncorrected, the errors can cascade until it is too late to do anything but restart the modeling process. Modeling tools that continuously and actively search for model errors aid the domain expert by reducing the need to diagnose and debug the cause of an error. Ideally, a modeling tool detects and reports an error immediately after its introduction, allowing the modeler to fix the error before it spreads to other parts of the model. Balci [19] gives a comprehensive list of model verification and validation techniques and their applicability to the model lifecycle. Since the modeler can

use many of these techniques only during a specific part of the model lifecycle, it is often necessary for a modeling tool to combine several techniques to provide full coverage.

Having a computer-understandable model test plan allows for automation of model testing. Automatic test plan evaluation encourages the modeler to perform testing more frequently and the modeling tool can incorporate the test plan into the continuously run tests for model quality [9]. However, automatic evaluation of a test plan has cost in terms of the time that the modeler spends specifying the test plan and the time that the modeling tool spends executing the test plan. Overstreet [110] notes that the cost of automatic test plan evaluation can make this automation unattractive even when model correctness is crucial.

The JigCell modeling environment has little support for actively monitoring the quality of a model. Although the JigCell Comparator supports automatic test plan evaluation, the modeler must explicitly instruct the application to begin model testing. Performing a test plan for a typical biological model has significant cost. The JigCell applications cannot execute a model test plan while the user is interactively editing the model.

Diagnosing model errors with a knowledge base

Determining the reason that a model test fails and localizing the model fault is particularly difficult. Improving the skill of diagnosing or debugging the source of error in a model is hard. In general software programming, fault localization is the most difficult part of debugging and requires extensive knowledge to perform [50]. For a domain expert working on a model, this can result in an excessive expenditure of time correcting model errors.

The tenet of knowledge bases is that modelers generally do not build models to solve a particular problem and then entirely discard those models. Instead, modelers can reuse the knowledge that they generated by building a model in a domain by treating the creation of successive models as an ongoing process [48]. A knowledge base records the experience gained by a domain expert or modeling and simulation specialist when performing model diagnosis. Another modeler or domain expert can then research this information when attempting to build or modify a model.

Birta and Ozmizrak [31] describe using a knowledge base to generate new experiments and model tests. This provides similar support for model diagnosis while reducing the need for the domain expert to construct a comprehensive suite of model test cases.

The JigCell modeling environment originally planned to construct a database of past modeling activities. Modelers could search within this database for situations similar to their current problem and discover what other modelers attempted previously. However, the JigCell modeling environment never incorporated archival storage or retrieval of modeling activities. The cost of archiving the configuration data and results is similar to, and in some cases in excess of, the cost of performing those modeling activities again. Moreover, the design space of a typical biological model is vast. It is unlikely that modelers encountered enough similar circumstances previously to give rational advice about how to solve a particular modeling problem. The JigCell modeling environment does not currently have any support for diagnosing model errors with a knowledge base. Instead, the JigCell modeling environment relies on making model experimentation cheap to allow the modeler to quickly explore many different ideas for fixing a model error.

Keeping historical records of model development and testing

Historical records of model development and testing provide crucial information about the modeling process. Keeping a historical record assists with the credibility of a model and is useful for planning future modeling tasks. A modeling tool adds information to the historical record whenever the modeler transforms, modifies, or tests the model. The modeler can then use the historical record in support of automated testing, which also helps the domain expert review model verification, validation, and testing methods. Having a historical record makes the model more accessible to experimentation. A historical record lists past experiments done with a model and makes it easier to undo model changes after the modeler detects an error.

WBCSim [62] is a simulation problem-solving environment for wood-based composite models. The intended users of WBCSim are manufacturers and wood scientists. Users construct a wood composite model, perform experiments on the model, and receive visualizations of the experimental results. The addition of a historical record to WBCSim improved usability by allowing searches and comparisons of past model experiments [128]. A database stores notes from the modeler, model modifications, simulation setup, and simulation results.

The JigCell modeling environment has minimal support for keeping a historical record of model development and testing. As Section 4.4.2 described, modelers can save the results of model tests in the JigCell Comparator for later viewing and analysis. However, the JigCell Comparator only records the result of a model test, discarding all other data that the modeler may need to interpret the test results. For typical biological models, a model test produces too much data to permanently archive. Moreover, the other JigCell applications do not keep any records of model development and testing. Modelers must manually record the model history or test results that they feel are important.

Presenting multiple visualizations of models and model outputs

Visualizations are graphical representations of models and model results. Graphical model representations, such as block diagrams, are useful for communicating the high-level relationships in a model [120]. Animations and plots are common examples of visualizations of model outputs. Graphical representations often combine with textual representations. The graphical representation depicts the structure of the model, and the textual representation fills out the details. Sanchez and Langley [118] present an example of this hybrid modeling approach.

The use of visualizations improves understandability by distinctively representing patterns that the modeler considers important. Visualizations aid model verification, validation, and testing by making models and model results more understandable and by clearly showing the incorrect behavior when the model is not working correctly [32]. Visualizations reduce the need for the domain expert to master the technical and abstruse specification languages that modeling and simulation specialists often use for models. However, it is important to remember that the simplicity of visualization can come from hiding important model details. A poorly chosen visualization may give the modeler a misleading impression of the model.

One concern about providing multiple visualizations for a model is the expense of maintaining multiple model representations. Modeling tools can avoid this expense by maintaining a single model from which they generate multiple presentations. Padmanaban, Benjamin, and Mayer [111] illustrate the use of a knowledge base to store pertinent data about a model, which modeling tools can then use to create different model views. This approach prevents the introduction of inconsistencies between model representations. Otherwise, it is necessary to evaluate whether the improvement to model understandability is worth the increased cost of development and execution. Nance, Overstreet, and Page [104] report that modeling tools can eliminate the redundancy of multiple model representations automatically in some cases, significantly improving the execution time.

The JigCell modeling environment provides several means of visualizing models and model results, although modelers make use of these capabilities rarely. Section 4.4 described the visualization methods available in the JigCell Comparator, including Compare^2. The JigCell Model Builder uses the standardized SBML language to allow modelers to work with the model in a spreadsheet form while using other modeling applications that have graphical representations. Although both representations reside in the same model file, there is substantial duplication between the graphical and mathematical forms. The graphical form of the model is fragile and does not automatically update after changes to the mathematical model. The modeler must fix the resulting inconsistencies manually.

5.3 Collecting and Checking User Requirements

The remainder of this chapter discusses collecting user requirements and evaluates the JigCell modeling environment using those requirements. As the name implies, user requirements are the stated needs of the intended users. The intended users of the JigCell modeling environment are biological modelers that build large models of biochemical reaction networks. The models that these modelers build follow the approximations that Section 2.5 described, making the models suitable for description by ordinary differential equations. Although the intended users of the JigCell modeling environment are not necessarily expert biological modelers, JigCell presently does not specifically focus on supporting novice modelers. Supporting novice modelers would require additional user requirements.

The primary means of collecting user requirements is to interview users. The user requirements in this dissertation come from interviews of the same modeling group as the original modeling process in Section 3.1. Users initially participated in an unstructured interview session in which they discussed their concerns about software features. The

development of the requirements specification was iterative. After each interview session, users received for comment a requirements specification based on the needs that they expressed during the session.

This requirements specification consists of two types of requirements. Feature requirements are qualitative requirements that specify a particular action that a modeling tool should perform. Performance requirements are quantitative requirements that specify a minimum capacity or limit for a modeling tool.

Collecting user requirements is difficult because a requirements specification needs a well-defined scope and consistent level of detail. The scope of this requirements specification is the revised modeling process of Section 3.3, which the JigCell modeling environment implements. Therefore, this dissertation states user requirements in terms of the revised modeling process even though the user may have specified their requirements in terms of the JigCell applications. The level of detail of this requirements specification is "features that correspond to significant model actions". An average feature requirement should correspond to at least several weeks of developer time to implement but no more than several months. Therefore, a single feature requirement may aggregate several specific but small requests by users. Conversely, a single challenging user request can lead to multiple feature requirements.

Benchmarking

This dissertation tests support for user performance requirements through benchmarking. A benchmark is a well-defined standard of measurement that an experimentalist can repeatably apply to a software program to obtain a quantitative result. A user performance requirement consists of a benchmark methodology that explains how to perform the benchmark and a benchmark target that gives a threshold of acceptability for the benchmark result. The following benchmark methodology and targets apply to all of the benchmarks in this dissertation.

- A benchmark measures real elapsed times, with a resolution of no worse than one second.
- A benchmark program must not use more than 75% of the physical memory of the machine and must not use more than one gigabyte of physical memory regardless of the available quantity of physical memory.
- A benchmark program must not use more than one gigabyte of disk space.
- A benchmark program that produces an answer must produce the expected correct answer.
- A benchmark program must not experience a program fault during or after the benchmark run.
- A benchmark program must leave the modeling tool in a usable condition after the benchmark run.
- A benchmark program must not degrade or disable the user interface, or otherwise perform less work than performing the equivalent operations through normal user interaction.

This dissertation uses a single reference machine for performing benchmarks. Table 5.2 gives the software configuration of the reference machine and Table 5.3 gives the hardware configuration.

Table 5.2: Configuration of the test environment software for the benchmark reference machine.

Program	JigCell version 6.0.5
Compiler	IBM Jikes Java compiler version 1.22
Optimizer	Sun Java HotSpot Client VM version 1.5.0.04-b05, mixed mode
Runtime	Sun Java Runtime Environment SE version 1.5.0.04-b05
Operating System	Microsoft Windows XP Service Pack 2

The reference machine ran benchmarks with the Intel processor HyperThreading extensions enabled. There is little measurable difference between running the benchmarks with the Intel HyperThreading extensions enabled and disabled. The modeling tools use a single thread almost exclusively during each benchmark run.

Benchmarks used the high-performance system timer, which has at least microsecond resolution. The benchmarks configured the Java runtime environment to terminate the benchmark program after allocating more than 75%

Table 5.3: Configuration of the test environment hardware for the benchmark reference machine.

Processor	Intel Pentium 4 stepping D1
	3.2 GHz clock speed
	800 MHz front side bus speed
Cache	8 KB L1 data
	12 KB L1 instruction
	512 KB L2 unified
Memory	1 GB DDR2700 RAM
Disk	80 GB capacity
	100 MB/s interface bandwidth
	49.3 MB/s average transfer rate
	19.2 ms access time

(768 MB) of the physical memory of the machine. The benchmark results note the peak heap allocation size during the benchmark run, taken observationally. Note that the peak heap allocation size is not necessarily the amount of memory that the benchmark requires as the Java runtime environment can defer reclaiming freed objects. Additionally, the benchmark results estimate a lower bound on the heap allocation size by setting the heap allocation limit progressively lower until the benchmark fails due to memory exhaustion.

5.4 User Requirements for Model Building

This requirements specification defines benchmark models in terms of SBML Level 2 Version 1. However, modeling tools do not have to support SBML. Modeling tools that support a different model representation format should appropriately translate the benchmark models to their storage format.

A model consists of a name, description, list of compartments, list of chemical species, and list of chemical reactions. The size of a benchmark model is the number of chemical reactions. Each compartment has a unique identifier, name, description, size, and topology. A benchmark model has a single, three-dimensional compartment with unit size. Each chemical reaction has a unique identifier, name, kinetic formula, and parameters. The kinetic formulas for the chemical reactions are simple mass action kinetic formulas ($k_i \mathrm{S}_i$ for the ith chemical reaction). Reactions are irreversible. Each parameter, $k_i = 1/(i+2)$, has a unique identifier, name, description, and value. Each chemical species has a unique identifier, name, description, containing compartment, and initial amount. The initial amount for every chemical species is 1.0. The single compartment of the model contains all of the chemical species.

Model Building Performance

(1) The model building tool should support a benchmark model containing at least 1000 chemical reactions.

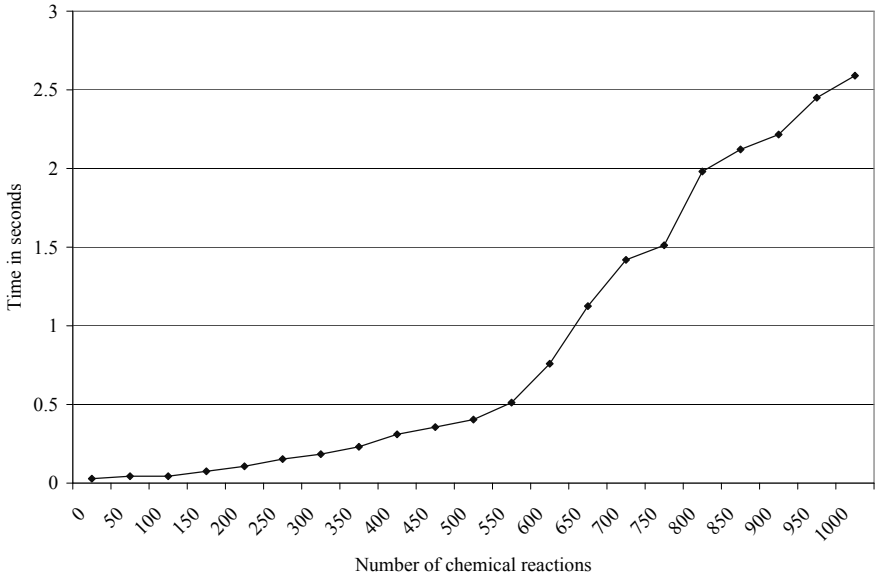
Analysis: The model loading benchmark measures whether the JigCell Model Builder supports a model. The JigCell Model Builder supports a model if the loading time is less than 300 seconds. The procedure for measuring model loading performance is:

1. Create a test model containing N chemical reactions,
2. write the test model to disk,
3. and read the test model from disk.

The benchmark time is the time to execute Step 3.

The JigCell Model Builder passes this benchmark. The time to load a model containing 1000 chemical reactions is 2.6 seconds, using between 32 MB and 66 MB of memory and 757 KB of disk space. Figure 5.1 shows a graph of the timing data interpolated between the measurement points. Although parsing the model initially dominates the benchmark time, detecting conservation relations increasingly dominates as the model becomes larger.

Figure 5.1: Time to load a model in the JigCell Model Builder against the number of chemical reactions. The benchmark target is to load a model containing 1000 chemical reactions within 300 seconds.



(2) The model building tool should support entering a chemical reaction in less than two minutes.

Analysis: The chemical reaction entry benchmark measures whether the JigCell Model Builder supports entering a new chemical reaction. The JigCell Model Builder supports entering a chemical reaction if the expended time is less than 120 seconds for a model containing 200 chemical reactions. The procedure for measuring chemical reaction entry performance is:

1. Create a test model containing N chemical reactions,
2. write the test model to disk,
3. read the test model from disk,
4. add chemical species $N + 1$,
5. add chemical reaction $N + 1$,
6. add parameter $N + 1$,
7. and write the test model to disk.

The benchmark time is the cumulative times to execute Step 3 through Step 7.

The JigCell Model Builder does not pass this benchmark. The JigCell Model Builder terminates abnormally while executing the benchmark. As the previous version of the JigCell Model Builder finished this benchmark successfully, this defect is new. The previous version of the JigCell Model Builder met this requirement.

Model Building Functionality

(3) The model building tool should support a standard interchange language for representing models. Support of a standard interchange language includes the ability to read, but not necessarily display, any valid model using the language and the ability to reject invalid models. The model building tool must use the interchange language as its native language for saving models.

Analysis: The JigCell Model Builder meets this requirement. Section 4.2 described the exclusive use of SBML Level 2 Version 1 by the JigCell Model Builder.

(4) The model building tool should support displaying a wiring diagram for the model. The model building tool does not need to display the wiring diagram unless the user requests one for a model.

Analysis: The JigCell Model Builder does not meet this requirement. The JigCell project previously planned to integrate the Model Builder with another model building tool that supports wiring diagrams. However, the JigCell Model Builder does not attempt this integration, which would require extensive development time.

(5) The model building tool should support highlighting errors in the model while the modeler is working. A model error includes definitions that the interchange language cannot persist and definitions that will prevent simulation.

Analysis: The JigCell Model Builder does not meet this requirement. The JigCell Model Builder supported this feature in a previous version. Section 4.5 described the JigCell SBML Parser, which contains routines that verify model elements. Supporting this requirement in the JigCell Model Builder requires calling the verification routines in the JigCell SBML Parser. Changing the JigCell Model Builder to call these verification routines is a minor effort.

(6) The model building tool should support verifying that the model structure is consistent. Model verification includes checking that all references to model elements refer to an element that exists in the model. The references that the model building tool needs to be check include references to compartments, chemical species, rules, and parameters in SBML. Additionally, model verification includes checking for invalidly structured models, such as a cycle between compartment containments.

Analysis: The JigCell Model Builder does not meet this requirement. The JigCell Model Builder supported this feature in a previous version. The JigCell Model Builder does not support model verification for a few types of model elements, such as units. Changing the JigCell Model Builder to verify all types of model elements is a minor effort.

(7) The model building tool should support describing models with ordinary and stochastic differential equations. Specification of ordinary differential equations requires constructing the right-hand side of the equation. The model building tool must not have unreasonable limits on the number or size of terms on the right-hand sides of the differential equations. Specification of stochastic differential equations requires an additional noise function for the equation.

Analysis: The JigCell Model Builder does not meet this requirement. Although the JigCell Model Builder supports ordinary differential equations, neither the Model Builder nor the SBML language supports stochastic differential equations. Supporting stochastic differential equations with an SBML model is a major effort.

(8) The model building tool should support creating differential equations from the chemical reactions and chemical reaction kinetics of the biochemical reaction network.

Analysis: The JigCell Model Builder meets this requirement. The JigCell Model Builder implements the conversion process in Section 2.4.1 for constructing ordinary differential equations from a biochemical reaction network.

(9) The model building tool should support displaying the model equations. Possible display formats for equations include plain-text mathematics, rich-text mathematics such as TEX, and program code.

Analysis: The JigCell Model Builder meets this requirement. Figure 4.3 illustrated the user interface in the JigCell Model Builder that displays the system of equations.

(10) The model building tool should support detecting conservation relations in the biochemical reaction network.

Analysis: The JigCell Model Builder meets this requirement. As Section 4.2.2 described, the JigCell Model Builder implements the algorithm in Section 2.4.2 for detecting the conservation relations in a biochemical reaction network.

(11) The model building tool should support importing model fragments into the current model. Model fragments are packaged models or components of models. Importation of a model fragment requires resolving the connection points between the current model and model fragment.

Analysis: The JigCell Model Builder does not meet this requirement. Supporting model composition requires considerable user interface support in the JigCell Model Builder. Moreover, model composition may require modifications to the SBML language to support merging model element definitions from separate models.

(12) The model building tool should support automating the input of similar types of chemical reactions. The model building tool must support selecting from a catalog of kinetic formulas that the user can customize. The model building tool must support basic editing functionality, such as cut, copy, and paste.

Analysis: The JigCell Model Builder meets this requirement. Section 4.2 described entering chemical reactions into the JigCell Model Builder, including reusing previous definitions of chemical reactions and kinetic formulas.

(13) The model building tool should support creating multiple model views that hide levels of detail. A model view collapses a user-defined collection of chemical species, chemical reactions, or compartments to a single point or opaque box. The model building tool must display references to model elements within a model view.

Analysis: The JigCell Model Builder does not meet this requirement. It is unlikely that the JigCell Model Builder could support this requirement without first supporting model composition.

Table 5.4: Current support for model building in the JigCell Model Builder. The JigCell modeling environment supports six of the requirements and is close to supporting three additional requirements.

#	Supported	< 4 weeks	< 4 months	> 4 months
1	•			
2		•		
3	•			
4				•
5		•		
6		•		
7			•	
8	•			
9	•			
10	•			
11				•
12	•			
13				•

5.5 User Requirements for Model Execution

A run file consists of a model file name, basal file name, simulator name, simulator control settings, and list of runs. The size of a benchmark run file is the number of runs. The benchmark model file contains 200 chemical reactions, and the basal file duplicates the parameter and initial condition values in the model file. The simulator is freely selectable, and the simulator control settings are the default values for that simulator. Each run has a unique identifier, name, description, and changes to the parameters and initial conditions. The value of the ith model parameter in the $j$the run is $1/(i * j + 2)$, starting with $i = 0$ and $j = 1$. The initial amount of the ith model chemical species in the jth run is $1/(i + j + 2)$, starting with $i = 0$ and $j = 1$.

Model Execution Performance

(14) The model execution tool should support a benchmark run file containing at least 10000 runs. The number of runs is typically larger than the number of chemical reactions in the model.

Analysis: The run file loading benchmark measures whether the JigCell Run Manager supports a run file. The JigCell Run Manager supports a run file if the loading time is less than 300 seconds. The procedure for measuring run file loading performance is:

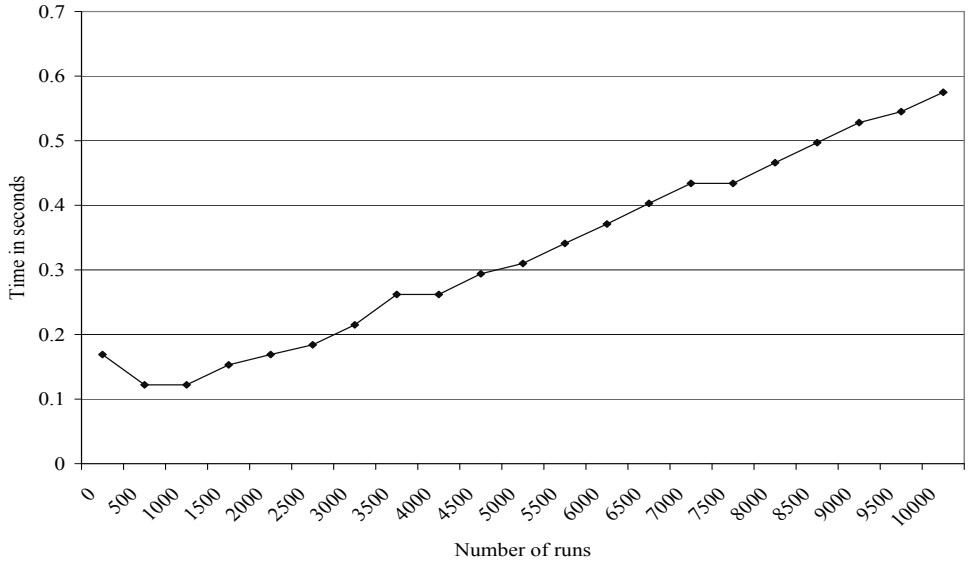
1. Create a test model containing 200 chemical reactions,
2. write the test model to disk,

3. create a test run file containing N runs,
4. write the test run file to disk,
5. and read the test run file from disk.

The benchmark time is the time to execute Step 5.

The JigCell Run Manager passes this benchmark. The time to load a run file containing 10000 runs is 0.6 seconds, using between 8 MB and 18 MB of memory and 3456 KB of disk space. Figure 5.2 shows a graph of the timing data interpolated between the measurement points.

Figure 5.2: Time to load a run file in the JigCell Run Manager against the number of runs. The benchmark target is to load a run file containing 10000 runs within 300 seconds.



(15) The model execution tool should support entering a run in less than five minutes.

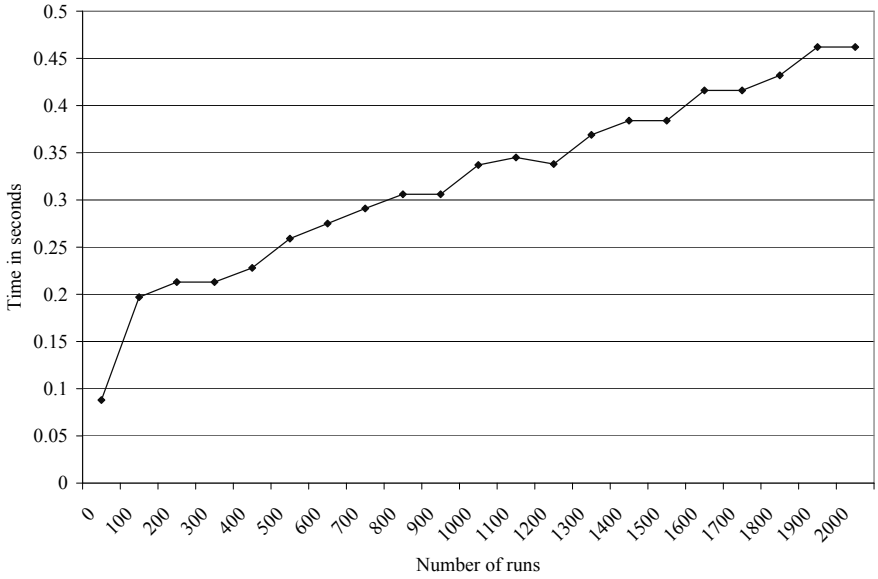
Analysis: The run entry benchmark measures whether the JigCell Run Manager supports entering a new run. The JigCell Run Manager supports entering a run if the expended time is less than 300 seconds for a run file containing 2000 runs. The procedure for measuring run entry performance is:

1. Create a test model containing 200 chemical reactions,
2. write the test model to disk,
3. create a test run file containing N runs,
4. write the test run file to disk,
5. read the test run file from disk,
6. add run $N + 1$,
7. and write the test run file to disk.

The benchmark time is the cumulative times to execute Step 5 through Step 7.

The JigCell Run Manager passes this benchmark. The time to enter a run into a run file containing 2000 runs is 0.5 seconds, using between 8 MB and 15 MB of memory and 934 KB of disk space. Figure 5.3 shows a graph of the timing data interpolated between the measurement points.

Figure 5.3: Time to enter a run into a run file in the JigCell Run Manager against the number of runs. The benchmark target is to enter a run into a run file containing 2000 runs within 300 seconds.



(16) The model execution tool should support executing a single run for a preliminary model in less than one minute.

Analysis: The run execution benchmark measures whether the JigCell Run Manager supports executing a run for a preliminary model. The JigCell Run Manager supports executing a run for a preliminary model if the expended time is less than 60 seconds for a run file containing 2000 runs. The procedure for measuring run execution performance is:

1. Create a test model containing 200 chemical reactions,
2. write the test model to disk,
3. create a test run file containing N runs,
4. write the test run file to disk,
5. read the test run file from disk,
6. and execute the first run.

The benchmark time is the cumulative times to execute Step 5 through Step 6.

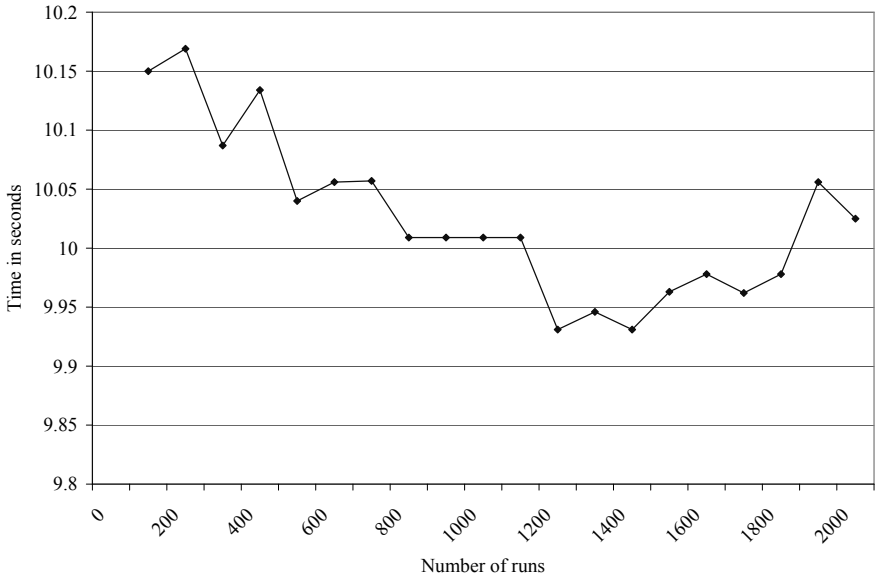
The JigCell Run Manager passes this benchmark. The time to execute a run for a run file containing 2000 runs is 10.0 seconds, using between 8 MB and 12 MB of memory and 1015 KB of disk space. Figure 5.4 shows a graph of the timing data interpolated between the measurement points. At this number of runs, the time to perform the simulation entirely dominates the benchmark time. Although the JigCell Run Manager currently supports executing runs, there is no way to access this command from the user interface. Once the JigCell Run Manager supplies a user interface for this functionality, the benchmark time will likely increase slightly.

(17) The model execution tool should support changing a single parameter or initial condition value in less than one minute. Modelers make frequent changes to parameter and initial condition values during parameter twiddling.

Analysis: The run modification benchmark measures whether the JigCell Run Manager supports changing a single parameter or initial condition value. The JigCell Run Manager supports changing a single parameter or initial condition value if the expended time is less than 60 seconds for a run file containing 2000 runs. The procedure for measuring run modification performance is:

1. Create a test model containing 200 chemical reactions,

Figure 5.4: Time to execute a run in the JigCell Run Manager against the number of runs. The benchmark target is to execute a run in a run file containing 2000 runs within 60 seconds. Invoking and running the simulation program dominates the benchmark at this number of runs, causing a small, random fluctuation in the time spent.



2. write the test model to disk,
3. create a test run file containing N runs,
4. write the test run file to disk,
5. read the test run file from disk,
6. set the parameter for the first chemical reaction in the first run to 0.0,
7. set the initial condition for the first chemical species in the first run to 0.0,
8. and write the test run file to disk.

The benchmark time is the cumulative times to execute Step 5 through Step 8.

The JigCell Run Manager passes this benchmark. The time to change a single parameter or initial condition value in a run file containing 2000 runs is 0.5 seconds, using between 8 MB and 11 MB of memory and 934 KB of disk space. Figure 5.5 shows a graph of the timing data interpolated between the measurement points.

Model Execution Functionality

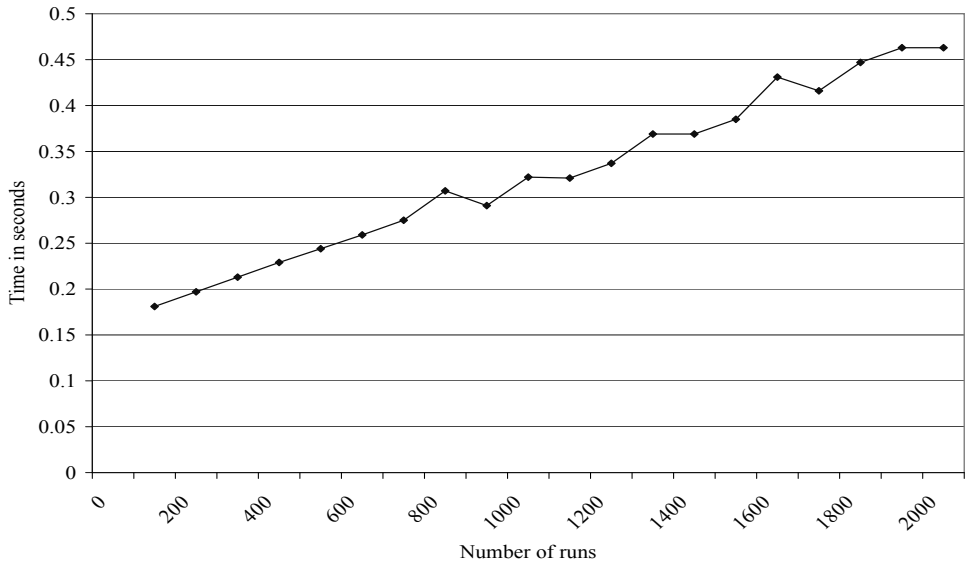
(18) The model execution tool should support multiple simulators. Supporting a simulator requires registering the simulation program with the model execution tool and querying the simulator about its control settings.

Analysis: The JigCell Run Manager meets this requirement. Section 4.5 described the Simulator API that supports registering simulators and querying simulators about their control settings, and Section 4.3 described the use of the Simulator API by the JigCell Run Manager.

(19) The model execution tool should support configuring simulator control settings. The model execution tool must display the known control settings for a simulator and persist the control setting changes for a run within the run file.

Analysis: The JigCell Run Manager meets this requirement. Figure 4.8 showed the user interface for editing simulator control settings in the JigCell Run Manager after selecting the XPPAUT simulator.

Figure 5.5: Time to change a single parameter or initial condition value in a run file against the number of runs. The benchmark target is to change a single parameter or initial condition value for a run in a run file containing 2000 runs within 60 seconds.



(20) The model execution tool should support validating the suitability of the model and simulator configuration. Simulators cannot accurately execute every model. Additionally, some simulation programs further restrict the class of suitable models according to the simulator control settings. The model execution tool must check the model for suitability prior to execution.

Analysis: The JigCell Run Manager does not meet this requirement. The SBML language does not support describing the simulation features that a model needs, and the Simulator API does not support describing the simulation features that a simulator provides. Supporting this requirement would require significant changes to the SBML language, Simulator API, and JigCell Run Manager.

(21) The model execution tool should support inheritance of parameters and initial conditions. Updates to a parameter or initial condition value propagate to all of the derived runs that do not explicitly override the value for that setting.

Analysis: The JigCell Run Manager meets this requirement. Section 4.3 described constructing ensembles of runs in the JigCell Run Manager that inherit parameter and initial condition changes between runs.

(22) The model execution tool should support highlighting errors in the configuration while working. An error includes a missing or illegal value for the model, simulator, simulator control settings, basal parameters and initial conditions, or parameter and initial condition changes of a run.

Analysis: The JigCell Run Manager does not meet this requirement. The JigCell Run Manager could easily validate the model, simulator, basal parameters and initial conditions, and parameter and initial condition changes and apply highlighting to those portions of the user interface. Although the Simulator API provides an indication of the valid range of simulator control settings, the JigCell Run Manager does not make use of this information.

(23) The model execution tool should support updating the run file when the model file changes. The model execution tool must propagate changes to the model to all runs that use that model.

Analysis: The JigCell Run Manager meets this requirement. The JigCell Run Manager rereads the model each time a run needs model information.

(24) The model execution tool should support importing parameter and initial condition sets from the model. When a model provides parameter or initial condition values, the model execution tool must permit the user to transfer the values in the model to the basal parameters and initial conditions.

Analysis: The JigCell Run Manager meets this requirement. As Section 4.3 mentioned, the JigCell Run Manager has a command that creates a basal file from the parameter and initial condition values in the model.

(25) The model execution tool should support executing a single simulation run and displaying its output. The model execution tool does not need to support executing runs that fail validation. The model execution tool must display the results of a run in a tabular or graphical form for the user to review.

Analysis: The JigCell Run Manager does not meet this requirement. The JigCell Run Manager supported this feature in a previous version. The JigCell modeling environment includes libraries for executing simulation runs and displaying plots, which Section 4.5 described in detail. Supporting this requirement in the JigCell Run Manager user interface is a minor effort.

Table 5.5: Current support for model execution in the JigCell Run Manager. The JigCell modeling environment supports nine of the requirements and is close to supporting one additional requirements.

#	Supported	< 4 weeks	< 4 months	> 4 months
14	•			
15	•			
16	•			
17	•			
18	•			
19	•			
20				•
21	•			
22			•	
23	•			
24	•			
25		•		

5.6 User Requirements for Model Analysis

An experiment set is a collection of experiments. The size of an experiment set is the number of experiments. Each experiment has a unique identifier, name, experimental observation, type, and description. The experimental observation is four observations of a two-variable time series. The type identifies the experimental observation as a time series and names the variables.

A transform set is a collection of transforms and a mapping function from experiment names to transforms. The size of a transform set is the number of mappings from experiment names to transforms. Each transform has a unique identifier, name, and procedural description. The procedural description is a program built from primitive transforms, including the control settings that the primitive transforms use and a reference to the transforms that supply input. The transform procedure for each named experiment consists of two primitive transforms and requires one parameter. The first transform receives input from the second transform; the second transform does not require input. The output of the transform procedure is a time series in the same format as the experimental observation.

An objective set is a collection of objective functions and a mapping function from experiment names to objective functions. The size of an objective set is the number of mappings from experiment names to objectives. Each objective function has a unique identifier, name, and procedural description. The procedural description is a program that specifies the objective function code and the control settings that the objective function code uses. The objective procedure for each named experiment requires one parameter.

Model Analysis Performance

(26) The model analysis tool should support an experiment set containing at least 10000 experiments. The number of experiments typically is similar to the number of runs.

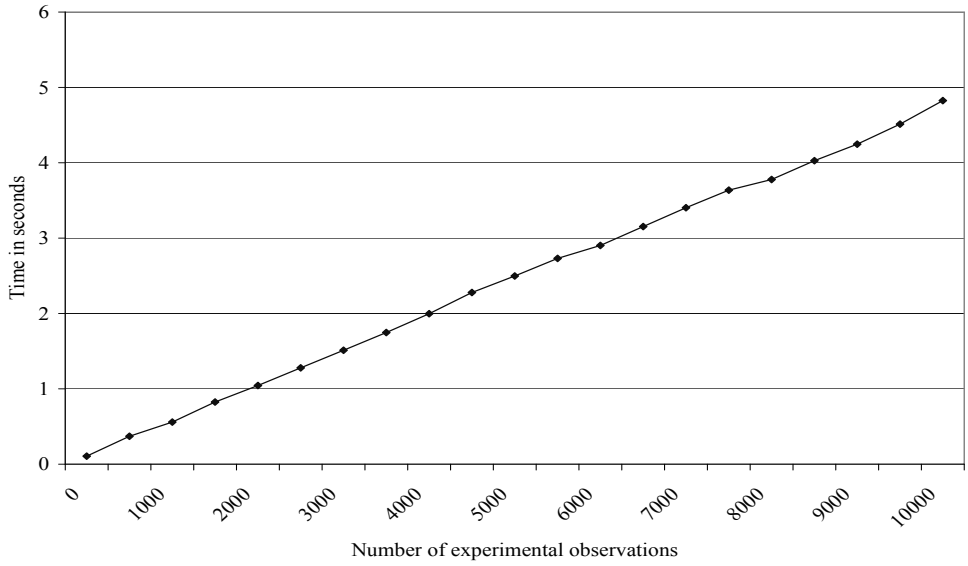
Analysis: The experiment set loading benchmark measures whether the JigCell Comparator supports an experiment set. The JigCell Comparator supports an experiment set if the loading time is less than 300 seconds. The procedure for measuring experiment set loading performance is:

1. Create a test experiment set containing N experiments,
2. write the test experiment set to disk,
3. and read the test experiment set from disk.

The benchmark time is the time to execute Step 3.

The JigCell Comparator passes this benchmark. The time to load an experiment set containing 10000 experiments is 4.8 seconds, using between 16 MB and 31 MB of memory and 5010 KB of disk space. Figure 5.6 shows a graph of the timing data interpolated between the measurement points.

Figure 5.6: Time to load an experiment set in the JigCell Comparator against the number of experiments. The benchmark target is to load an experiment set containing 10000 experiments within 300 seconds.



(27) The model analysis tool should support a transform set containing at least 2000 distinct transforms. Each different experimental data format needs its own transform. Additionally, some experimental protocols require the use of a specialized transformation procedure.

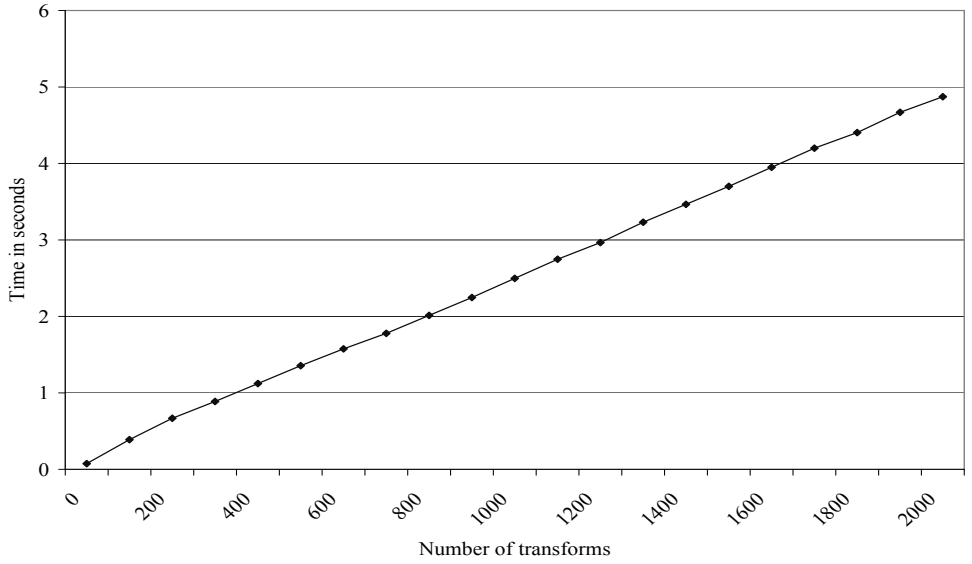
Analysis: The transform set loading benchmark measures whether the JigCell Comparator supports a transform set. The JigCell Comparator supports a transform set if the loading time is less than 300 seconds for an experiment set containing 2000 experiments. The procedure for measuring transform set loading performance is:

1. Create a test experiment set containing 2000 experiments,
2. write the test experiment set to disk,
3. create a test transform set containing N transforms,
4. write the test transform set to disk,
5. read the test experiment set from disk,
6. and read the test transform set from disk.

The benchmark time is the time to execute Step 6.

The JigCell Comparator passes this benchmark. The time to load a transform set containing 2000 distinct transforms is 4.9 seconds, using between 16 MB and 21 MB of memory and 3515 KB of disk space. Figure 5.7 shows a graph of the timing data interpolated between the measurement points.

Figure 5.7: Time to load a transform set in the JigCell Comparator against the number of transforms. The benchmark target is to load a transform set containing 2000 distinct transforms within 300 seconds.



(28) The model analysis tool should support an objective set containing at least 2000 distinct objective functions. Each different experimental data format needs its own objective function.

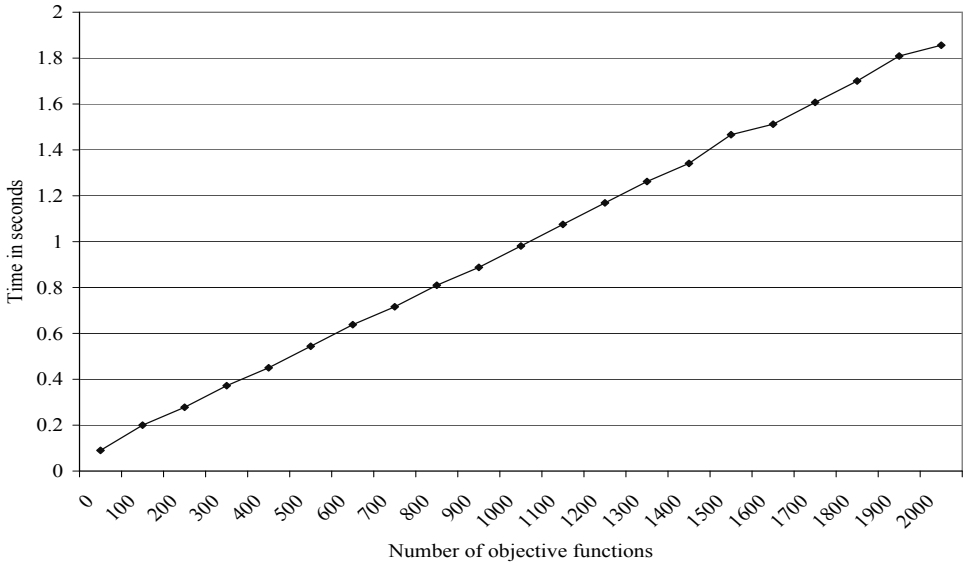
Analysis: The objective set loading benchmark measures whether the JigCell Comparator supports an objective set. The JigCell Comparator supports an objective set if the loading time is less than 300 seconds for an experiment set containing 2000 experiments. The procedure for measuring objective set loading performance is:

1. Create a test experiment set containing 2000 experiments,
2. write the test experiment set to disk,
3. create a test transform set containing N transforms,
4. write the test transform set to disk,
5. create a test objective set containing N objectives,
6. write the test objective set to disk,
7. read the test experiment set from disk,
8. read the test transform set from disk,
9. and read the test objective set from disk.

The benchmark time is the time to execute Step 9.

The JigCell Comparator passes this benchmark. The time to load an objective set containing 2000 distinct objective functions is 1.9 seconds, using between 16 MB and 25 MB of memory and 5536 KB of disk space. Figure 5.8 shows a graph of the timing data interpolated between the measurement points.

Figure 5.8: Time to load an objective set in the JigCell Comparator against the number of objective functions. The benchmark target is to load an objective set containing 2000 distinct objective functions within 300 seconds.



(29) The model analysis tool should support entering an experimental observation in less than two minutes.

Analysis: The experiment entry benchmark measures whether the JigCell Comparator supports entering an experimental observation. The JigCell Comparator supports entering an experimental observation if the expended time is less than 120 seconds for an experiment set containing 2000 experiments. The procedure for measuring experiment entry performance is:

1. Create a test experiment set containing N experiments,
2. write the test experiment set to disk,
3. read the test experiment set from disk,
4. add experiment $N + 1$,
5. and write the test experiment set to disk.

The benchmark time is the cumulative time to execute Step 3 through Step 5.

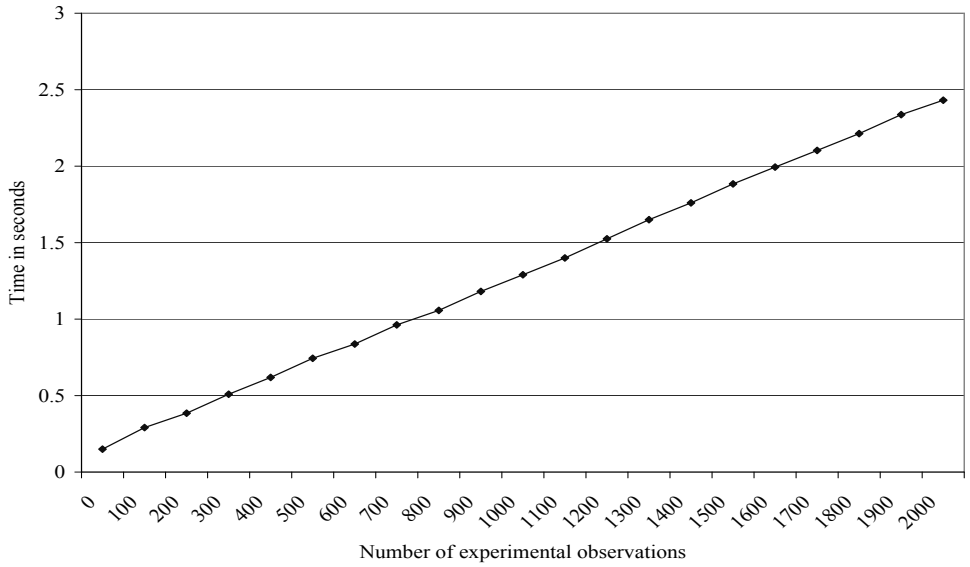
The JigCell Comparator passes this benchmark. The time to enter an experimental observation into an experiment set containing 2000 experiments is 2.4 seconds, using between 8 MB and 15 MB of memory and 997 KB of disk space. Figure 5.9 shows a graph of the timing data interpolated between the measurement points.

(30) The model analysis tool should support configuring a comparison in less than five minutes. The configuration time does not include the time spent programming transforms, objective functions, or data types.

Analysis: The experiment configuration benchmark measures whether the JigCell Comparator supports configuring a comparison. The JigCell Comparator supports configuring a comparison if the expended time is less than 300 seconds to configure a comparison with an experiment set containing 2000 experiments, a transform set containing 2000 distinct transforms, and an objective set containing 2000 distinct objective functions. The procedure for measuring experiment configuration performance is:

1. Create a test experiment set containing N experiments,
2. write the test experiment set to disk,
3. create a test transform set containing $N - 1$ transforms,

Figure 5.9: Time to enter an experimental observation into an experiment set in the JigCell Comparator against the number of experiments. The benchmark target is to enter an experimental observation into an experiment set containing 2000 experiments within 120 seconds.



4. write the test transform set to disk,
5. create a test objective set containing $N - 1$ objectives,
6. write the test objective set to disk,
7. read the test experiment set from disk,
8. read the test transform set from disk,
9. create transform N,
10. add transform N to experiment N,
11. write the test transform set to disk,
12. read the test objective set from disk,
13. create objective N,
14. add objective N to experiment N,
15. and write the test objective set to disk.

The benchmark time is the cumulative time to execute Step 7 through Step 15.

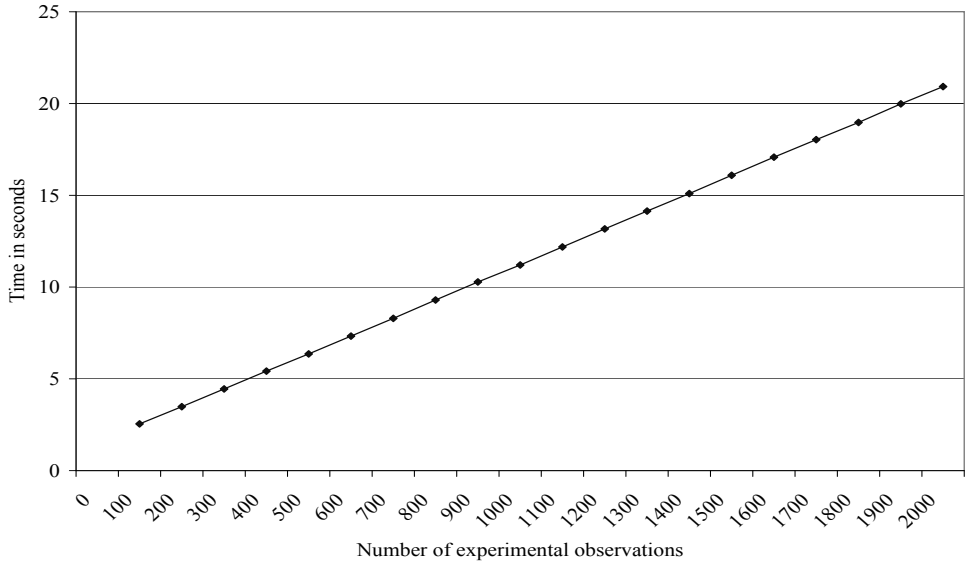
The JigCell Comparator passes this benchmark. The time to configure a comparison with an experiment set containing 2000 experiments, a transform set containing 2000 distinct transforms, and an objective set containing 2000 distinct objective functions is 20.9 seconds, using between 32 MB and 35 MB of memory and 5536 KB of disk space. Figure 5.10 shows a graph of the timing data interpolated between the measurement points.

Model Analysis Functionality

(31) The model analysis tool should support dividing experimental observations into multiple pieces. An experimental observation is frequently a collection of data, such as a time series or multiple related observations. The model analysis tool must support complex data types that allow independent addressing of each scalar element.

Analysis: The JigCell Comparator meets this requirement. Section 4.4.1 and Section 4.5 described the list-of-lists format that the JigCell Comparator uses to represent complex data types for experimental observations.

Figure 5.10: Time to configure a comparison in the JigCell Comparator against the number of experiments. The benchmark target is to configure a comparison with an experiment set containing 2000 experiments, a transform set containing 2000 distinct transforms, and an objective set containing 2000 distinct objective functions within 300 seconds.



(32) The model analysis tool should support using both owned and external experimental observations. Owned experimental observations are observations for which the model analysis tool has control of the storage. External experimental observations are observations for which some program independent of the model analysis tool has control of the storage. The model analysis tool must make the consumption of owned and external experimental observations transparent to the user.

Analysis: The JigCell Comparator meets this requirement. The JigCell Comparator supports executing code while loading an experiment set. The code for an experiment set can reference external experimental observations, making those experimental observations available as if the experiment set stored the experimental observations directly.

(33) The model analysis tool should support references in the experimental observations that refer back to source information.

Analysis: The JigCell Comparator meets this requirement. The format for experimental observations in the JigCell Comparator includes a field where the user can enter reference information. Figure 4.9 shows an example of the user using this field to record literature references.

(34) The model analysis tool should support assigning types to experimental observations, transform inputs and results, and objective function inputs. The model analysis tool must allow the user or code author to specify the data type, including complex data types if the model analysis tool supports that feature. Complex data types must support a type for each scalar value in the experimental observation.

Analysis: The JigCell Comparator meets this requirement. Experimental data, transforms, and objective functions have types that describe their format. Figure 4.9 shows the 'Value Type' column that declares the data type for a comparison. Section 6.2.1 gives an example of assigning type information to complex data types.

(35) The model analysis tool should support highlighting type errors while the user works. A type error occurs when the experimental observation, transform, or objective function does not match the declared type.

Analysis: The JigCell Comparator meets this requirement. Figure 4.11 shows the JigCell Comparator highlighting the 'Value Type' column of a comparison to indicate a type error. In that example, the 'MPF activation/inactivation' experiment has a declared type 'MPF thresholds' but the objective function 'WOSS' expects a time series.

(36) The model analysis tool should support independent definition of the experimental observations, transforms, and objective functions. The definitions of transforms and objective functions, including control settings and source code, must reside separately from experimental observations. The model analysis tool must connect transforms and objective functions to experimental observations without relying on identifiers that derive from user-controllable properties.

Analysis: The JigCell Comparator meets this requirement. The JigCell Comparator stores experimental observations, transform, objective functions, and program code separately. Each experimental observation, transform, and objective function has a randomly-generated globally unique identifier for cross-referencing.

(37) The model analysis tool should support updating transforms that perform runs when the run file changes. The model analysis tool must propagate changes to the run file to all transforms that use those runs.

Analysis: The JigCell Comparator meets this requirement. The JigCell Comparator rereads the run file each time a transform needs information about a run.

(38) The model analysis tool should support unattended execution. An error in the execution of one transform or objective function must not prevent the execution of independent transforms and objective functions. The model analysis tool must monitor progress and display progress information and an error report to the user.

Analysis: The JigCell Comparator meets this requirement. Users can select a group of comparisons that the JigCell Comparator will execute in batch mode. The JigCell Comparator displays the batch progress and provides an error log that records exceptional conditions.

(39) The model analysis tool should support defining a named group of comparisons. Biological modelers frequently rely on particular collections of comparisons to give an early test of model fitness during parameter twiddling. The model analysis tool must allow the user to identify a group of comparisons, give that group of comparisons a name, and persist storage of these named groups across program sessions.

Analysis: The JigCell Comparator does not meet this requirement. Although users can select groups of comparisons, the JigCell Comparator does not support giving names to these groups and does not record the groups for future use. The experiment set file format can store the membership of experimental observations in groups, and adding support for groups to the JigCell Comparator user interface is a minor effort.

(40) The model analysis tool should support user-defined transforms, objective functions, and data types. It is unreasonable to expect that a modeling environment can supply a comprehensive set of transforms, objective functions, and data types that will satisfy all users. New experimental techniques and observations require the creation of novel data structures and analysis procedures. The model analysis tool must support adding new data types, transforms, and objective functions without requiring recompilation.

Analysis: The JigCell Comparator meets this requirement. At run time, the JigCell Comparator loads a user-supplied list of independent program code modules, which can include data types, transforms, and objective functions. Allen [5] describes the process for creating new data types, transforms, and objective functions for the JigCell Comparator.

5.7 Other User Requirements

System Requirements

(41) The software system should support constructing a preliminary model in less than five days. A preliminary model is the initial testable implementation of a modeling hypothesis, containing 5% of the maximum supported number of chemical reactions and 1% of the maximum supported numbers of runs, experimental observations, transforms, and objective functions. A preliminary model does not support parameter estimation. Building a preliminary model does not include time spent programming transforms, objective functions, or data types.

Analysis: The JigCell modeling environment meets this requirement. If the user has an existing wiring diagram, simulation runs, and experimental observations, then entering this information into the JigCell modeling environment is straightforward. Modelers have entered models with more than 100 chemical reactions, runs, and experimental observations within one day.

Table 5.6: Current support for model analysis in the JigCell Comparator. The JigCell modeling environment supports fourteen of the requirements and is close to supporting one additional requirement.

#	Supported	< 4 weeks	< 4 months	> 4 months
26	•			
27	•			
28	•			
29	•			
30	•			
31	•			
32	•			
33	•			
34	•			
35	•			
36	•			
37	•			
38	•			
39		•		
40	•			

(42) The software system should support organizing a working model over three or more years. A working model has 20% of the maximum supported numbers of chemical reactions, runs, experimental observations, transforms, and objective functions. A working model supports parameter estimation and includes data files, historical model development information, and custom modules for transforms, objective functions, and data types.

Analysis: The JigCell modeling environment does not meet this requirement. Although Section 4.6 described an application for managing projects and project data, the JigCell modeling environment does not attempt to support project management. The JigCell modeling environment requires that users organize their own data.

Table 5.7: Current support for system requirements in the JigCell modeling environment. The JigCell modeling environment supports one of the requirements.

#	Supported	< 4 weeks	< 4 months	> 4 months
41	•			
42			•	

Model Tuning Requirements

(43) The model tuning tool should support configuring a parameter estimation run in less than one day.

Analysis: The experimental tool for parameter estimation does not meet this requirement. Converting the data files from the JigCell modeling environment for use with parameter estimation requires considerable manual processing. Moreover, the parameter estimation tool cannot use the transforms and objective functions from the JigCell Comparator, requiring the user to program new modules.

(44) The model tuning tool should support unattended execution. An error in the execution of a run must not prevent the execution of independent runs. The model tuning tool must monitor progress and display progress information and an error report to the user.

Analysis: The experimental tool for parameter estimation meets this requirement. The parameter estimator runs unattended until the program terminates or meets the stopping criteria that the user supplied. The parameter estimator logs execution results for later viewing.

(45) The model tuning tool should support returning found parameters to other tools. After a successful parameter estimation run, there are new values for each of the free parameters. The model tuning tool must support inserting parameter values found during parameter estimation into a basal file.

Analysis: The experimental tool for parameter estimation meets this requirement. Although the parameter estimator does not create basal files, the experimental tools include scripts that extract basal files from the parameter estimation execution log.

(46) The model tuning tool should support weighting the experimental observations. Modelers associate a weight with each experimental observation that scales the objective function score to indicate their confidence in the accuracy of the observation. If the model analysis tool uses a compatible objective function that defines these weights, then the model tuning tool must support importing the weighting data from the model analysis data files.

Analysis: The experimental tool for parameter estimation meets this requirement. The parameter estimator multiplies each objective function score with a scalar weight. The overall model score is the sum of these products. The parameter estimator does not import weights from the JigCell Comparator data files because the objective functions in the JigCell Comparator do not define suitable weights.

Table 5.8: Current support for model tuning in the experimental tools. The JigCell modeling environment does not currently include the experimental tools as part of the standard distribution. The experimental tools support three of the requirements.

#	Supported	< 4 weeks	< 4 months	> 4 months
43			•	
44	•			
45	•			
46	•			

Chapter 6

Case Study of a Budding Yeast Model

The cell division cycle is a central biological process that dictates how a cell grows and replicates [99, 108]. Misregulation of the cell cycle process leads to serious diseases and cell death. As Chapter 2 described, this vital process is highly conserved across eukaryotic organisms [107]. Biologists have previously elucidated the cell cycle control system in budding yeast (*Saccharomyces cerevisiae*) in great detail [93, 105]. Due to the highly-conserved nature of the process, biological modelers can apply the knowledge that they gain about budding yeast to many organisms.

The budding yeast cell cycle is a series of several phases that repeats endlessly. The cell spends a majority of its time in interphase, the long interval between occurrences of mitosis. A newly born cell starts in the G1 phase. During the G1 phase, the cell acquires resources and grows until there are suitable conditions for replication. Cells that cannot commit to replication wait in a resting state until the environmental conditions improve.

After reaching a viable size for replication, the cell transitions to the S phase. During the S phase, the cell synthesizes a new copy of its DNA. After the completion of DNA synthesis, the cell continues to grow until it reaches a mass of approximately twice its birth size. This phase, known as the G2 phase in the standard cell cycle, is generally uninteresting in budding yeast. However, the cell soon reaches the end of interphase and begins mitosis, which is the most physically complex portion of the cell cycle.

Mitosis, also called M phase, is the process of nuclear and cell division. By the end of mitosis, the cell will produce a complete new copy of itself. At the start of mitosis, the nuclear membrane breaks down and a mitotic spindle forms. Then, the cell enters metaphase, in which the duplicated chromosomes align themselves along the mitotic spindle. Next, the cell passes through anaphase and enters telophase. During this period, the pairs of chromosomes separate and move to the ends of the mitotic spindle, where new nucleuses are forming. Finally, the cell pinches until the mass of the cell divides. After division, the two cells are again in the G1 phase.

Modelers in the Tyson laboratory have built mathematical models of the budding yeast cell cycle control system that attempt to reproduce and explain the temporal evolution of cell growth and division using the modeling processes that Chapter 3 described. These modelers then want to validate the proposed biological mechanism by comparing the numerical solutions of their differential equations with the observed cellular behavior in the laboratory. Experimentalists have made more than 100 phenotypic observations of budding yeast mutant strains. Even with a computer solving the regulating equations, modelers find that evaluating a model by hand is tedious and error-prone.

Chapter 4 introduced the JigCell modeling environment that supports automated evaluation of biological models. Chapter 5 continued examining JigCell and analyzed the capabilities of the modeling environment by comparing the features that JigCell provides with the features that modelers need and request. The present chapter also aims to examine JigCell and analyze the capabilities that JigCell provides as a modeling environment. However, instead of examining JigCell from a requirements perspective, this chapter examines JigCell from an application perspective. The focus of this chapter is to apply JigCell to a real biological model and quantitatively measure the efficacy of JigCell for the task of computerized model evaluation.

This chapter presents a case study applying JigCell to a model for cell cycle control in budding yeast. Section 6.1 introduces the biology behind the budding yeast model and summarizes its mathematical structure. Section 6.2 describes the implementation of the automated model evaluation procedure in JigCell. As Section 4.4 discussed, a model

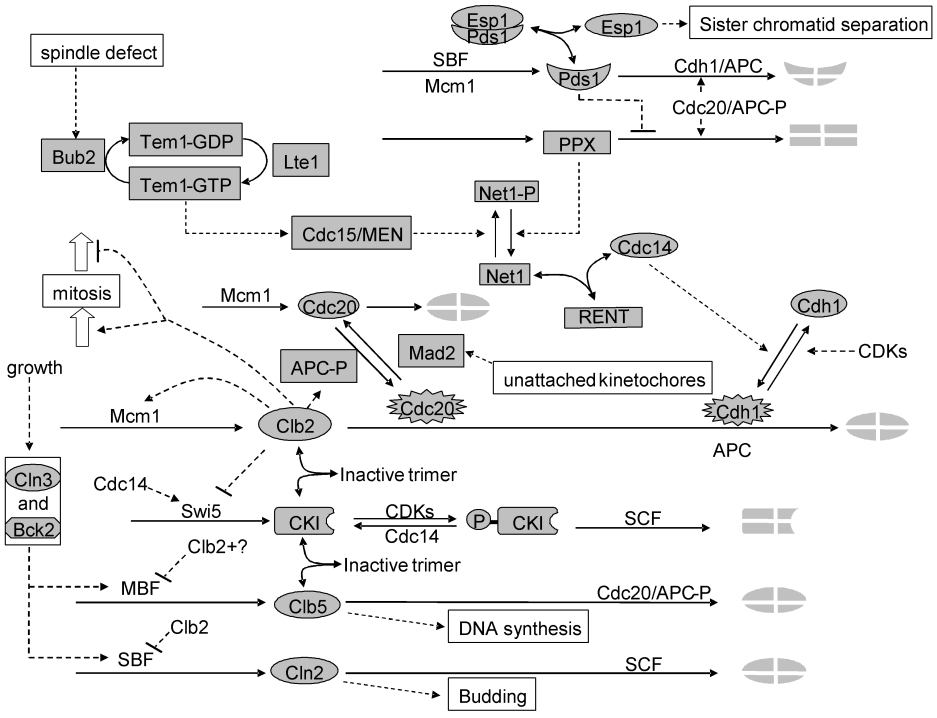
evaluation procedure requires experimental data, a data transformation procedure that extracts information equivalent to the experimental data from the model, and an objective function that measures the distance between the experimental data and the output of the data transformation. Section 6.1.2 categorizes the mutant strains that the automated model evaluation procedure uses for experimental observations and Section 6.2 provides the format for the experimental data, a data transformation procedure, and an objective function. Finally, Section 6.3 presents the results of the case study. Section 6.3 compares the results of an expert modeler using the manual model evaluation procedure that Section 3.1 described with an equivalent automated model evaluation procedure in JigCell.

Contents

6.1 Budding Yeast Model

Chen et al. [41] previously developed a model of the budding yeast cell cycle. Figure 6.1 shows their wiring diagram for the biochemical reaction network that regulates DNA synthesis, bud emergence, and mitosis in budding yeast.

Figure 6.1: Wiring diagram for the budding yeast model by Chen.

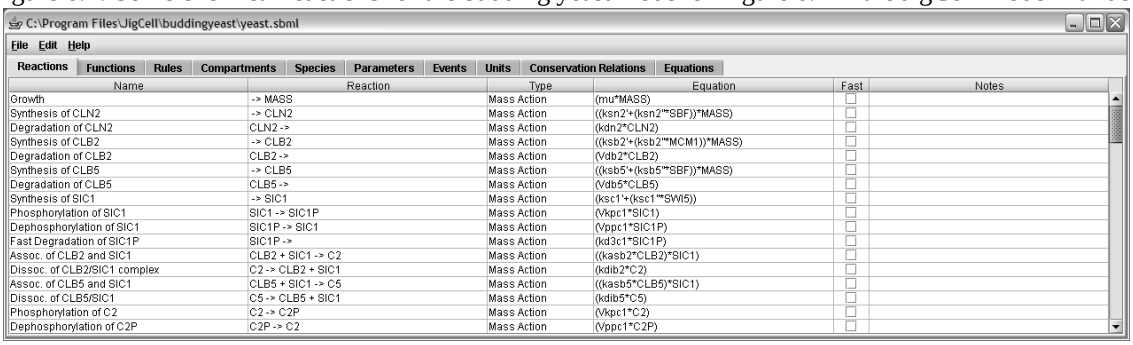


As Section 3.1 mentioned, modelers sometimes simplify the wiring diagram by omitting portions of the model and using abstractions. Figure 6.1 heavily employs such techniques. For example, Chen uses different shapes to

distinguish the forms of chemical species such as Esp1, Pds1, CKI, Cdc20, and Cdh1. Shading and silhouette indicates the dissociation of chemical species under particular conditions. In some places, a text description of a function stands in the place of that part of the model. Furthermore, Chen combined several chemical species together to simplify the model. Cln2 in the model represents both Cln1 and Cln2 in the cell. Similarly, Clb5 represents both Clb5 and Clb6, and Clb2 represents both Clb1 and Clb2. Although Chen does not formalize or describe these abstractions in detail, understanding the abstractions is essential to understanding the model.

Figure 6.2 shows part of this same budding yeast model in the JigCell Model Builder. The version of the budding yeast model in the Model Builder consists of 94 chemical reaction equations. As Section 4.2 mentioned, the Model Builder only supports irreversible chemical reactions. However, since nearly all of the pairs of chemical reactions proceed at asymmetrical rates in the budding yeast model, this restriction is rarely a problem.

Figure 6.2: Some chemical reactions for the budding yeast model of Figure 6.1 in the JigCell Model Builder.



Name	Reaction	Type	Equation	Fast	Notes
Growth	-> MASS	Mass Action	(mu*MASS)		
Synthesis of CLN2	-> CLN2	Mass Action	((ksn2'+(ksn2''*SBF))*MASS)		
Degradation of CLN2	CLN2 ->	Mass Action	(kdn2*CLN2)		
Synthesis of CLB2	-> CLB2	Mass Action	((ksb2'+(ksb2''*MCM1))*MASS)		
Degradation of CLB2	CLB2 ->	Mass Action	(Vdb2*CLB2)		
Synthesis of CLB5	-> CLB5	Mass Action	((ksb5'+(ksb5''*SBF))*MASS)		
Degradation of CLB5	CLB5 ->	Mass Action	(Vdb5*CLB5)		
Synthesis of SIC1	-> SIC1	Mass Action	(ksc1'+(ksc1''*SWI5))		
Phosphorylation of SIC1	SIC1 -> SIC1P	Mass Action	(Vkpc1*SIC1)		
Dephosphorylation of SIC1	SIC1P -> SIC1	Mass Action	(Vppc1*SIC1P)		
Fast Degradation of SIC1P	SIC1P ->	Mass Action	(kd3c1*SIC1P)		
Assoc. of CLB2 and SIC1	CLB2 + SIC1 -> C2	Mass Action	((kasb2*CLB2)*SIC1)		
Dissoc. of CLB2/SIC1 complex	C2 -> CLB2 + SIC1	Mass Action	(kdib2*C2)		
Assoc. of CLB5 and SIC1	CLB5 + SIC1 -> C5	Mass Action	((kasb5*CLB5)*SIC1)		
Dissoc. of CLB5/SIC1	C5 -> CLB5 + SIC1	Mass Action	(kdib5*C5)		
Phosphorylation of C2	C2 -> C2P	Mass Action	(Vkpc1*C2)		
Dephosphorylation of C2P	C2P -> C2	Mass Action	(Vppc1*C2P)		

6.1.1 Mathematical Model

The Model Builder translates the biochemical reaction network into a system of differential-algebraic equations with 36 differential equations. There are seven algebraic conservation relations, of which the modeler explicitly created three and the conservation relation finding algorithm in the Model Builder detected four.

$$\frac{d[\mathrm{MASS}]}{dt} = \mu[\mathrm{MASS}]$$

$$\frac{d[\text{APC-P}]}{dt} = \frac{k_{\text{a,apc}}[\mathrm{Clb2}]([\mathrm{APC}]_{\mathrm{T}} - [\text{APC-P}])}{J_{\text{a,apc}} + ([\mathrm{APC}]_{\mathrm{T}} - [\text{APC-P}])} - \frac{k_{\text{i,apc}}[\text{APC-P}]}{J_{\text{i,apc}} + [\text{APC-P}]}$$

$$\frac{d[\mathrm{BUD}]}{dt} = k_{\text{s,bud}}(\epsilon_{\text{bud,n2}}[\mathrm{Cln2}] + \epsilon_{\text{bud,n3}}[\mathrm{Cln3}] + \epsilon_{\text{bud,b5}}[\mathrm{Clb5}]) - k_{\text{d,bud}}[\mathrm{BUD}]$$

$$\frac{d[\mathrm{C2}]}{dt} = k_{\text{as,b2}}[\mathrm{Clb2}][\mathrm{Sic1}] + k_{\text{pp,c1}}[\mathrm{Cdc14}][\mathrm{C2P}] - (k_{\text{di,b2}} + V_{\text{d,b2}} + V_{\text{kp,c1}})[\mathrm{C2}]$$

$$\frac{d[\mathrm{C2P}]}{dt} = V_{\text{kp,c1}}[\mathrm{C2}] - (k_{\text{pp,c1}}[\mathrm{Cdc14}] + k_{\text{d3,c1}} + V_{\text{d,b2}})[\mathrm{C2P}]$$

$$\frac{d[\mathrm{C5}]}{dt} = k_{\text{as,b5}}[\mathrm{Clb5}][\mathrm{Sic1}] + k_{\text{pp,c1}}[\mathrm{Cdc14}][\mathrm{C5P}] - (k_{\text{di,b5}} + V_{\text{d,b5}} + V_{\text{kp,c1}})[\mathrm{C5}]$$

$$\frac{d[\mathrm{C5P}]}{dt} = V_{\text{kp,c1}}[\mathrm{C5}] - (k_{\text{pp,c1}}[\mathrm{Cdc14}] + k_{\text{d3,c1}} + V_{\text{d,b5}})[\mathrm{C5P}]$$

$$\frac{d[\mathrm{Cdc6}]}{dt} = (k'_{\text{s,f6}} + k''_{\text{s,f6}}[\mathrm{Swi5}] + k'''_{\text{s,f6}}[\mathrm{SBF}]) + (V_{\text{d,b2}} + k_{\text{di,f2}})[\mathrm{F2}] + (V_{\text{d,b5}} + k_{\text{di,f5}})[\mathrm{F5}] + k_{\text{pp,f6}}[\mathrm{Cdc14}][\mathrm{Cdc6P}] - (k_{\text{as,f2}}[\mathrm{Clb2}] + k_{\text{as,f5}}[\mathrm{Clb5}] + V_{\text{kp,f6}})[\mathrm{Cdc6}]$$

$$\frac{d[\mathrm{Cdc6P}]}{dt} = V_{\text{kp,f6}}[\mathrm{Cdc6}] - (k_{\text{pp,f6}}[\mathrm{Cdc14}] + k_{\text{d3,f6}})[\mathrm{Cdc6P}] + V_{\text{d,b2}}[\mathrm{F2P}] + V_{\text{d,b5}}[\mathrm{F5P}]$$

$$\frac{d[\text{Cdc14}]}{dt} = k_{\text{s,14}} - k_{\text{d,14}}[\text{Cdc14}] + (k_{\text{d,net}} + k_{\text{di,rent}})[\text{RENT}] + (k_{\text{d,net}} + k_{\text{di,rentp}})[\text{RENTP}] - (k_{\text{as,rent}}[\text{NET1}] + k_{\text{as,rentp}}[\text{Net1P}])[\text{Cdc14}]$$

$$\frac{d[\text{Cdc14}]_{\text{T}}}{dt} = k_{\text{s,14}} - k_{\text{d,14}}[\text{Cdc14}]_{\text{T}}$$

$$\frac{d[\text{Cdc15}]}{dt} = (k'_{\text{a,15}}[\text{Tem1}]_{\text{T}} + (k''_{\text{a,15}} - k'_{\text{a,15}})[\text{Tem1}] + k'''_{\text{a,15}}[\text{Cdc14}])([\text{Cdc15}]_{\text{T}} - [\text{Cdc15}]) - k_{\text{i,15}}[\text{Cdc15}]$$

$$\frac{d[\text{Cdc20}]}{dt} = (k'_{\text{a,20}} + k''_{\text{a,20}}[\text{APC-P}])([\text{Cdc20}]_{\text{T}} - [\text{Cdc20}]) - (k_{\text{mad2}} + k_{\text{d,20}})[\text{Cdc20}]$$

$$\frac{d[\text{Cdc20}]_{\text{T}}}{dt} = k'_{\text{s,20}} + k''_{\text{s,20}}[\text{Mcm1}] - k_{\text{d,20}}[\text{Cdc20}]_{\text{T}}$$

$$\frac{d[\text{Cdh1}]}{dt} = k_{\text{s,cdh}} - k_{\text{d,cdh}}[\text{Cdh1}] + \frac{V_{\text{a,cdh}}([\text{Cdh1}]_{\text{T}} - [\text{Cdh1}])}{J_{\text{a,cdh}} + ([\text{Cdh1}]_{\text{T}} - [\text{Cdh1}])} - \frac{V_{\text{i,cdh}}[\text{Cdh1}]}{J_{\text{i,cdh}} + [\text{Cdh1}]}$$

$$\frac{d[\text{Cdh1}]_{\text{T}}}{dt} = k_{\text{s,cdh}} - k_{\text{d,cdh}}[\text{Cdh1}]_{\text{T}}$$

$$\frac{d[\text{Clb2}]}{dt} = (k'_{\text{s,b2}} + k''_{\text{s,b2}}[\text{Mcm1}])[\text{MASS}] + k_{\text{di,b2}}[\text{C2}] + k_{\text{d3,c1}}[\text{C2P}] + k_{\text{di,f2}}[\text{F2}] + k_{\text{d3,f6}}[\text{F2P}] - (V_{\text{d,b2}} + k_{\text{as,b2}}[\text{Sic1}] + k_{\text{as,f2}}[\text{Cdc6}])[\text{Clb2}]$$

$$\frac{d[\text{Clb5}]}{dt} = (k'_{\text{s,b5}} + k''_{\text{s,b5}}[\text{SBF}])[\text{MASS}] + k_{\text{di,b5}}[\text{C5}] + k_{\text{d3,c1}}[\text{C5P}] + k_{\text{di,f5}}[\text{F5}] + k_{\text{d3,f6}}[\text{F5P}] - (V_{\text{d,b5}} + k_{\text{as,b5}}[\text{Sic1}] + k_{\text{as,f5}}[\text{Cdc6}])[\text{Clb5}]$$

$$\frac{d[\text{Cln2}]}{dt} = (k'_{\text{s,n2}} + k''_{\text{s,n2}}[\text{SBF}])[\text{MASS}] - k_{\text{d,n2}}[\text{Cln2}]$$

$$\frac{d[\text{Esp1}]}{dt} = (k_{\text{di,esp}} + V_{\text{d,pds}})([\text{Esp1}]_{\text{T}} - [\text{Esp1}]) - k_{\text{as,esp}}[\text{Pds1}][\text{Esp1}]$$

$$\frac{d[\text{F2}]}{dt} = k_{\text{as,f2}}[\text{Clb2}][\text{Cdc6}] + k_{\text{pp,f6}}[\text{Cdc14}][\text{F2P}] - (k_{\text{di,f2}} + V_{\text{d,b2}} + V_{\text{kp,f6}})[\text{F2}]$$

$$\frac{d[\text{F2P}]}{dt} = V_{\text{kp,f6}}[\text{F2}] - (k_{\text{pp,f6}}[\text{Cdc14}] + k_{\text{d3,f6}} + V_{\text{d,b2}})[\text{F2P}]$$

$$\frac{d[\text{F5}]}{dt} = k_{\text{as,f5}}[\text{Clb5}][\text{Cdc6}] + k_{\text{pp,f6}}[\text{Cdc14}][\text{F5P}] - (k_{\text{di,f5}} + V_{\text{d,b5}} + V_{\text{kp,f6}})[\text{F5}]$$

$$\frac{d[\text{F5P}]}{dt} = V_{\text{kp,f6}}[\text{F5}] - (k_{\text{pp,f6}}[\text{Cdc14}] + k_{\text{d3,f6}} + V_{\text{d,b5}})[\text{F5P}]$$

$$\frac{d[\text{Net1}]}{dt} = k_{\text{s,net}} - (k_{\text{d,net}} + k_{\text{as,rent}}[\text{Cdc14}] + V_{\text{kp,net}})[\text{Net1}] + (k_{\text{d,14}} + k_{\text{di,rent}})[\text{RENT}] + V_{\text{pp,net}}[\text{Net1P}]$$

$$\frac{d[\text{Net1}]_{\text{T}}}{dt} = k_{\text{s,net}} - k_{\text{d,net}}[\text{Net1}]_{\text{T}}$$

$$\frac{d[\text{ORI}]}{dt} = k_{\text{s,ori}}(\epsilon_{\text{ori,b2}}[\text{Clb2}] + \epsilon_{\text{ori,b5}}[\text{Clb5}]) - k_{\text{d,ori}}[\text{ORI}]$$

$$\frac{d[\text{Pds1}]}{dt} = k'_{\text{s,pds}} + k''_{\text{s1,pds}}[\text{SBF}] + k''_{\text{s2,pds}}[\text{Mcm1}] + k_{\text{di,esp}}([\text{Esp1}]_{\text{T}} - [\text{Esp1}]) - (V_{\text{d,pds}} + k_{\text{as,esp}}[\text{Esp1}])[\text{Pds1}]$$

$$\frac{d[\text{PPX}]}{dt} = k_{\text{s,ppx}} - V_{\text{d,ppx}}[\text{PPX}]$$

$$\frac{d[\text{RENT}]}{dt} = k_{\text{as,rent}}[\text{Cdc14}][\text{Net1}] + V_{\text{pp,net}}[\text{RENTP}] - (k_{\text{d,14}} + k_{\text{d,net}} + k_{\text{di,rent}} + V_{\text{kp,net}})[\text{RENT}]$$

$$\frac{d[\text{Sic1}]}{dt} = (k'_{\text{s,c1}} + k''_{\text{s,c1}}[\text{Swi5}]) + (V_{\text{d,b2}} + k_{\text{di,b2}})[\text{C2}] + (V_{\text{d,b5}} + k_{\text{di,b5}})[\text{C5}] + k_{\text{pp,c1}}[\text{Cdc14}][\text{Sic1P}] - (k_{\text{as,b2}}[\text{Clb2}] + k_{\text{as,b5}}[\text{Clb5}] + V_{\text{kp,c1}})[\text{Sic1}]$$

$$\frac{d[\text{Sic1P}]}{dt} = V_{\text{kp,c1}}[\text{Sic1}] + V_{\text{d,b2}}[\text{C2P}] + V_{\text{d,b5}}[\text{C5P}] - (k_{\text{pp,c1}}[\text{Cdc14}] + k_{\text{d3,c1}})[\text{Sic1P}]$$
$$\frac{d[\text{SPN}]}{dt} = \frac{k_{\text{s,spn}}[\text{Clb2}]}{J_{\text{spn}} + [\text{Clb2}]} - k_{\text{d,spn}}[\text{SPN}]$$
$$\frac{d[\text{Swi5}]}{dt} = k'_{\text{s,swi}} + k''_{\text{s,swi}}[\text{Mcm1}] + k_{\text{a,swi}}[\text{Cdc14}]([\text{Swi5}]_{\text{T}} - [\text{Swi5}]) - (k_{\text{d,swi}} + k_{\text{i,swi}}[\text{Clb2}])[\text{Swi5}]$$
$$\frac{d[\text{Swi5}]_{\text{T}}}{dt} = k'_{\text{s,swi}} + k''_{\text{s,swi}}[\text{Mcm1}] - k_{\text{d,swi}}[\text{Swi5}]_{\text{T}}$$
$$\frac{d[\text{Tem1}]}{dt} = \frac{k_{\text{lte1}}([\text{Tem1}]_{\text{T}} - [\text{Tem1}])}{J_{\text{a,tem}} + ([\text{Tem1}]_{\text{T}} - [\text{Tem1}])} - \frac{k_{\text{bub2}}[\text{Tem1}]}{J_{\text{i,tem}} + [\text{Tem1}]}$$
$$[\text{Bck2}] = B_0[\text{MASS}]$$
$$[\text{Cln3}] = \frac{C_0 D_{\text{n3}}[\text{MASS}]}{J_{\text{n3}} + D_{\text{n3}}[\text{MASS}]}$$
$$[\text{Mcm1}] = G(k_{\text{a,mcm}}[\text{Clb2}], k_{\text{i,mcm}}, J_{\text{a,mcm}}, J_{\text{i,mcm}})$$
$$[\text{SBF}] = G(V_{\text{a,sbf}}, V_{\text{i,sbf}}, J_{\text{a,sbf}}, J_{\text{i,sbf}})$$
$$[\text{Cdc6}]_{\text{T}} = [\text{Cdc6}] + [\text{Cdc6P}] + [\text{F2}] + [\text{F2P}] + [\text{F5}] + [\text{F5P}]$$
$$[\text{CKI}]_{\text{T}} = [\text{Sic1}]_{\text{T}} + [\text{Cdc6}]_{\text{T}}$$
$$[\text{Clb2}]_{\text{T}} = [\text{Clb2}] + [\text{C2}] + [\text{C2P}] + [\text{F2}] + [\text{F2P}]$$
$$[\text{Clb5}]_{\text{T}} = [\text{Clb5}] + [\text{C5}] + [\text{C5P}] + [\text{F5}] + [\text{F5P}]$$
$$[\text{Net1P}] = [\text{Net1}]_{\text{T}} - [\text{Net1}] + [\text{Cdc14}] - [\text{Cdc14}]_{\text{T}}$$
$$[\text{RENTP}] = [\text{Cdc14}]_{\text{T}} - [\text{RENT}] - [\text{Cdc14}]$$
$$[\text{Sic1}]_{\text{T}} = [\text{Sic}] + [\text{Sic1P}] + [\text{C2}] + [\text{C2P}] + [\text{C5}] + [\text{C5P}]$$

The remaining algebraic equations define the Goldbeter-Koshland function [63] and reusable rate terms that appear in the differential equations.

$$G(V_{\text{a}}, V_{\text{i}}, J_{\text{a}}, J_{\text{i}}) = \frac{2J_{\text{i}}V_{\text{a}}}{V_{\text{i}} - V_{\text{a}} + J_{\text{a}}V_{\text{i}} + J_{\text{i}}V_{\text{a}} + \sqrt{(V_{\text{i}} - V_{\text{a}} + J_{\text{a}}V_{\text{i}} + J_{\text{i}}V_{\text{a}})^2 - 4(V_{\text{i}} - V_{\text{a}})J_{\text{i}}V_{\text{a}}}}$$
$$V_{\text{a,cdh}} = k'_{\text{a,cdh}} + k''_{\text{a,cdh}}[\text{Cdc14}]$$
$$V_{\text{a,sbf}} = k_{\text{a,sbf}}(\epsilon_{\text{sbf,n2}}[\text{Cln2}] + \epsilon_{\text{sbf,n3}}([\text{Cln3}] + [\text{Bck2}]) + \epsilon_{\text{sbf,b5}}[\text{Clb5}])$$
$$V_{\text{d,b2}} = k'_{\text{d,b2}} + k''_{\text{d,b2}}[\text{Cdh1}] + k_{\text{d,b2p}}[\text{Cdc20}]$$
$$V_{\text{d,b5}} = k'_{\text{d,b5}} + k''_{\text{d,b5}}[\text{Cdc20}]$$
$$V_{\text{d,pds}} = k'_{\text{d1,pds}} + k''_{\text{d2,pds}}[\text{Cdc20}] + k''_{\text{d3,pds}}[\text{Cdh1}]$$
$$V_{\text{d,ppx}} = k'_{\text{d,ppx}} + \frac{k''_{\text{d,ppx}}(J_{\text{20,ppx}} + [\text{Cdc20}])J_{\text{pds}}}{[\text{Pds1}] + J_{\text{pds}}}$$
$$V_{\text{i,cdh}} = k'_{\text{i,cdh}} + k''_{\text{i,cdh}}(\epsilon_{\text{cdh,n3}}[\text{Cln3}] + \epsilon_{\text{cdh,n2}}[\text{Cln2}] + \epsilon_{\text{cdh,b2}}[\text{Clb2}] + \epsilon_{\text{cdh,b5}}[\text{Clb5}])$$
$$V_{\text{i,sbf}} = k'_{\text{i,sbf}} + k''_{\text{i,sbf}}[\text{Clb2}]$$
$$V_{\text{kp,c1}} = k_{\text{d1,c1}} + \frac{k_{\text{d2,c1}}(\epsilon_{\text{c1,n3}}[\text{Cln3}] + \epsilon_{\text{c1,k2}}[\text{Bck2}] + \epsilon_{\text{c1,n2}}[\text{Cln2}] + \epsilon_{\text{c1,b5}}[\text{Clb5}] + \epsilon_{\text{c1,b2}}[\text{Clb2}])}{J_{\text{d2,c1}} + [\text{Sic1}]_{\text{T}}}$$
$$V_{\text{kp,f6}} = k_{\text{d1,f6}} + \frac{k_{\text{d2,f6}}(\epsilon_{\text{f6,n3}}[\text{Cln3}] + \epsilon_{\text{f6,k2}}[\text{Bck2}] + \epsilon_{\text{f6,n2}}[\text{Cln2}] + \epsilon_{\text{f6,b5}}[\text{Clb5}] + \epsilon_{\text{f6,b2}}[\text{Clb2}])}{J_{\text{d2,f6}} + [\text{Cdc6}]_{\text{T}}}$$
$$V_{\text{kp,net}} = (k'_{\text{kp,net}} + k''_{\text{kp,net}}[\text{Cdc15}])[\text{MASS}]$$
$$V_{\text{pp,net}} = k'_{\text{pp,net}} + k''_{\text{pp,net}}[\text{PPX}]$$

The budding yeast model includes auxiliary variables BUD, ORI, and SPN in addition to the equations that control the chemical species in Figure 6.1. Auxiliary variables are model variables that exist solely to facilitate working with the model. These auxiliary variables link the evolution of protein concentrations to cell cycle events. The SBML model elements that support the traditional concept of a variable are chemical species and parameters. The modeler entered the auxiliary variables in the Model Builder using chemical species. Many simulation systems translate an SBML parameter into an unchanging model value regardless of how the model uses that parameter. This limitation makes the use of chemical species instead of parameters to represent auxiliary variables a more portable approach.

The concentration $[\mathrm{BUD}]$ signals the formation of a bud. In the model, the differential equation for BUD tracks the proteins that $\mathrm{Cdc28/cyclin}$ dimers phosphorylate. The rate of synthesis of BUD reflects the importance of these driving proteins known to affect bud formation [46]. The concentration $[\mathrm{ORI}]$ signals the onset of DNA synthesis and the invocation of the DNA replication and spindle assembly checkpoint. Finally, the concentration $[\mathrm{SPN}]$ signals the alignment of chromosomes along the metaphase plate and the lifting of the checkpoint. There is a known connection between the lifting of this checkpoint and a drop in Clb2-dependent kinase activity [141]. The drop in the concentration $[\mathrm{Clb2}]$ signals cellular division.

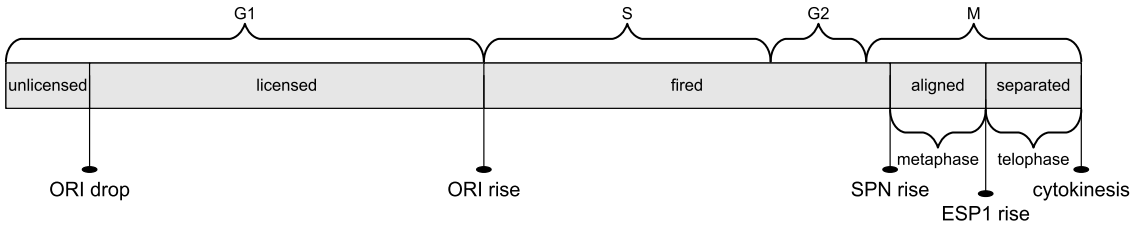
The modelers fitted the rate constants for the auxiliary variables against mutant strains that lack certain combinations of the involved cyclins. Furthermore, the modelers chose scales for the auxiliary variables so that events occur when the concentrations $[\mathrm{BUD}]$, $[\mathrm{ORI}]$, or $[\mathrm{SPN}]$ reach the threshold value of 1.0. However, the modelers later discovered that several viable mutant strains have a budding response that is too weak for the concentration $[\mathrm{BUD}]$ to surpass 1.0 in the model with the chosen rate constants. The automated model evaluation procedure in this dissertation uses a threshold value of 0.8 for the concentration $[\mathrm{BUD}]$. Since the scales of the auxiliary variables are arbitrary, the modeler must choose appropriate threshold values using experience about the model.

The budding yeast model cycles through five conceptual stages during execution. These conceptual stages of the model do not correspond directly to the phases in the traditional cell cycle. The model uses discrete events to signal transitions between the conceptual stages. Section 4.2 described the mechanism for implementing discrete events in JigCell. A model transition occurs when a function of time-dependent variables in the model crosses a threshold.

Initially, the model is in the **unlicensed** stage. When the concentration $[\mathrm{ORI}]$ drops below its threshold, the origins of replication on the DNA of the budding yeast cell become licensed and the model enters the **licensed** stage. Eventually, the concentration $[\mathrm{ORI}]$ rises back up above its threshold. At this point, the budding yeast cell completes the G1 phase of the traditional cell cycle and the model transitions to the **fired** stage. While the model is in the **fired** stage, the budding yeast cell replicates its DNA and passes through the S and G2 phases of the traditional cell cycle.

Finally, the budding yeast cell begins mitosis. When the concentration $[\mathrm{SPN}]$ rises above its threshold, the chromosomes align along the metaphase plate, and the model enters the **aligned** stage. The final event before division is a rise in the concentration $[\mathrm{Esp1}]$. This event occurs when the cell passes through anaphase to telophase, and the model transitions to the **separated** stage. All preparations for division in the budding yeast cell are now complete. When division occurs, the mass splits into two copies of the budding yeast cell, and the model returns to the **unlicensed** stage. Figure 6.3 shows the connection between the conceptual stages of the model, the discrete events that occur, and the traditional phases of the budding yeast cell cycle.

Figure 6.3: Connections between the conceptual stages in the model, discrete events in the simulation, and the traditional budding yeast cell cycle.



In the mathematical model, division occurs when the concentration $[\mathrm{Clb2}]$ drops below the parameter K_{ez}. The mother and daughter cells divide the mass unequally. The daughter cell receives $(1-2^{-\mu/D})[\mathrm{MASS}]$ mass, where μ is

the mass doubling time of the growth medium and D is the observed daughter cell cycle time. The mother cell receives the remaining mass. Typical values for μ are 90 minutes in glucose and 150 minutes in galactose. The concentrations $[\mathrm{BUD}]$ and $[\mathrm{SPN}]$ reset immediately after cellular division. The concentration $[\mathrm{ORI}]$ resets early in the cell cycle after the relicensing of the origins of replication of the cell [84]. A drop in the total concentration $[\mathrm{Clb2}] + [\mathrm{Clb5}]$ below the parameter K_{ez2} signals this event.

6.1.2 Mutant Strains

The modelers fitted the parameters of the budding yeast model by applying parameter twiddling and attempting to replicate the phenotypes of the wildtype and the 131 mutant strains in Table 6.1. The automated model evaluation procedure in this dissertation can handle the wildtype and 122 of these mutant strains. Experiments that expose the mutant strain to nocodazole require a different evaluation procedure. These experiments produce experimental data that is in a qualitatively different form from the standard experiments.

Table 6.1: Mutant strains used to fit the model. (∗) indicates a mutant strain Chen previously found not to agree with the experimental observations. (†) indicates an additional problematic mutant strain that the model evaluation procedure identified. (‡) indicates a mutant strain that the automated model evaluation procedure does not support.

	Wildtype	
in glucose	in galactose	
	Cln mutants	
cln1Δ cln2Δ	*GAL-CLN2 cln1Δ cln2Δ*	*cln1Δ cln2Δ sic1Δ*
cln1Δ cln2Δ cdh1Δ	*GAL-CLN2 cln1Δ cln2Δ cdh1Δ* (†)	*cln3Δ*
GAL-CLN3		
	Bck2 mutants	
bck2Δ	multi-copy *BCK2*	*cln1Δ cln2Δ bck2Δ*
cln3Δ bck2Δ	*cln3Δ bck2Δ GAL-CLN2 cln1Δ cln2Δ*	*cln3Δ bck2Δ* multi-copy *CLN2*
cln3Δ bck2Δ sic1Δ		
	Triple cln mutants	
cln1Δ cln2Δ cln3Δ	*cln1Δ cln2Δ cln3Δ GAL-CLN2*	*cln1Δ cln2Δ cln3Δ GAL-CLN3*
cln1Δ cln2Δ cln3Δ sic1Δ	*cln1Δ cln2Δ cln3Δ cdh1Δ*	*cln1Δ cln2Δ cln3Δ* multi-copy *CLB5*
cln1Δ cln2Δ cln3Δ GAL-CLB5	*cln1Δ cln2Δ cln3Δ* multi-copy *BCK2*	*cln1Δ cln2Δ cln3Δ GAL-CLB2*
cln1Δ cln2Δ cln3Δ apc^{ts}		
	Cdh1, Sic1, and Cdc6 mutants	
sic1Δ	*GAL-SIC1*	*GAL-SIC1-dbΔ*
GAL-SIC1 cln1Δ cln2Δ	*GAL-SIC1 GAL-CLN2 cln1Δ cln2Δ*	*GAL-SIC1 cln1Δ cln2Δ cdh1Δ*
GAL-SIC1 GAL-CLN2 cln1Δ cln2Δ cdh1Δ	*cdh1Δ* (∗)	Cdh1 constitutively active
sic1Δ cdh1Δ (∗)	*sic1Δ cdh1Δ GALL-CDC20*	*cdc6Δ2-49*
cdc6Δ2-49 sic1Δ	*cdc6Δ2-49 cdh1Δ* (∗)	*cdc6Δ2-49 sic1Δ cdh1Δ* (∗)
cdc6Δ2-49 sic1Δ cdh1Δ GALL-CDC20	*swi5Δ*	*swi5Δ GAL-CLB2*
swi5Δ cdh1Δ (∗)	*swi5Δ cdh1Δ GAL-SIC1*	
	Clb1 Clb2 mutants	
clb1Δ clb2Δ	*clb2Δ CLB1*(∗)	*GAL-CLB2*
Multi-copy *GAL-CLB2*	*clb2Δ CLB1 cdh1Δ* (∗)	*clb2Δ CLB1 pds1Δ* (∗)
GAL-CLB2 sic1Δ (∗)	*GAL-CLB2 cdh1Δ*	*CLB2-dbΔ*
CLB2-dbΔ in galactose	*CLB2-dbΔ* multi-copy *SIC1*	*CLB2-dbΔ GAL-SIC1*
CLB2-dbΔ multi-copy *CDC6*	*CLB2-dbΔ clb5Δ*	*CLB2-dbΔ clb5Δ* in galactose
GAL-CLB2-dbΔ		
	Clb5 Clb6 mutants	
clb5Δ clb6Δ	*clb5Δ clb6Δ cln1Δ cln2Δ*	*GAL-CLB5*

(Continued on next page)

GAL-CLB5 sic1Δ	*GAL-CLB5 cdh1Δ*	*CLB5-dbΔ*
CLB5-dbΔ sic1Δ	*CLB5-dbΔ pds1Δ*	*CLB5-dbΔ pds1Δ cdc20Δ*
GAL-CLB5-dbΔ		
	Cdc20 mutants	
cdc20^{ts}	*cdc20Δ clb5Δ*	*cdc20Δ pds1Δ*
cdc20Δ pds1Δ clb5Δ	*GAL-CDC20*	*cdc20^{ts} mad2Δ*
cdc20^{ts} bub2Δ		
	Pds1/Esp1 interaction mutants	
pds1Δ (∗)	*esp1^{ts}*	*PDS1-dbΔ*
GAL-PDS1-dbΔ	*GAL-PDS1-dbΔ esp1^{ts}*	*GAL-ESP1 cdc20^{ts}*
	MEN pathway mutants	
tem1Δ	*GAL-TEM1*	*tem1^{ts}* multi-copy *CDC15*
tem1^{ts} GAL-CDC15	*tem1Δ net1^{ts}*	*tem1Δ* multi-copy *CDC14*
cdc15Δ	Multi-copy *CDC15*(†)	*cdc15^{ts}* multi-copy *TEM1*
cdc15Δ net1^{ts}	*cdc15^{ts}* multi-copy *CDC14*	
	Exit-of-mitosis mutants	
net1^{ts}	*GAL-NET1*	*cdc14^{ts}*
GAL-CDC14	*GAL-CDC14 GAL-NET1*	*net1^{ts} cdc20^{ts}*
cdc14^{ts} GAL-SIC1	*cdc14^{ts}* then *GAL-SIC1*	*cdc14^{ts} sic1Δ* at permissive temp.
cdc14^{ts} cdh1Δ at permissive temp.	*cdc14^{ts} GAL-CLN2* at permissive temp.	*TAB6-1*(†)
TAB6-1 cdc15^{ts}	*TAB6-1 clb5Δ*	*TAB6-1 clb2Δ CLB1*(†)
	Checkpoint mutants	
mad2Δ	*bub2Δ*	*mad2Δ bub2Δ*
wildtype in nocodazole*(‡)*	*mad2Δ* in nocodazole*(‡)*	*mad2Δ GAL-TEM1* in nocodazole*(‡)*
mad2Δ pds1Δ in nocodazole*(‡)*	*bub2Δ* in nocodazole(∗)(‡)	*bub2Δ pds1Δ* in nocodazole*(‡)*
bub2Δ mad2Δ in nocodazole*(‡)*	*pds1Δ* in nocodazole*(‡)*	*net1^{ts}* in nocodazole*(‡)*
	APC mutants	
APC-A	*APC-A cdh1Δ*	*APC-A cdh1Δ* in galactose
APC-A cdh1Δ multi-copy *SIC1*(†)	*APC-A cdh1Δ GAL-SIC1*	*APC-A cdh1Δ* multi-copy *CDC6*
APC-A cdh1Δ GAL-CDC6	*APC-A cdh1Δ* multi-copy *CDC20*	*APC-A sic1Δ*
APC-A GAL-CLB2		

Each mutant strain uses the same system of differential equations, initial conditions, and parameters as the wild-type, changing only those values that the nature of the mutation governs. A mutant strain that deletes a gene sets the rate of synthesis for the corresponding protein to zero. A mutant strain that overexpresses a gene adjusts the rate of synthesis for the corresponding protein according to the method of overexpression. If the gene has multiple integrated copies under control of the natural promoter, then the mutant strain multiplies the rate of synthesis to account for the extra copies. If a foreign promoter constitutively overexpresses the gene, then the mutant strain increases the rate of synthesis and changes the specific growth rate of the cell to match the new medium.

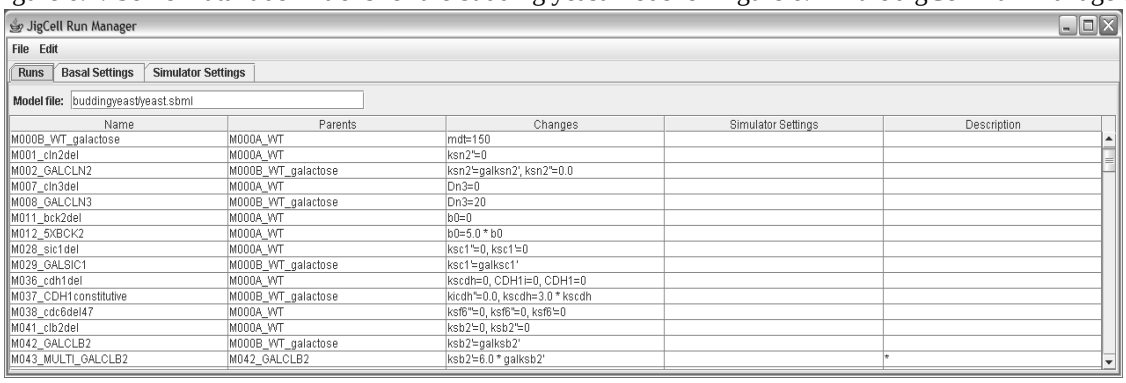
The modeler generally does not know the true rate of synthesis for an overexpressed gene. Instead, the modeler must fit the rate of synthesis against the known mutant strains with that method of overexpression. Every mutant that uses a particular overexpression construct shares a single value for the rate of synthesis.

6.2 Evaluation Procedure

Section 4.4 discussed the component pieces of a comparison in the JigCell Comparator. Specifically, the automated model evaluation procedure for the budding yeast model needs executable descriptions and experimental data for the mutant strains, a data transformation procedure, and an objective function. Section 6.1 described the budding yeast model and mutant strains. This section explains first how the executable description derives from the mathematical budding yeast model and mutant strain definitions and then covers the remaining components of a comparison in turn.

Previously, Chen entered the budding yeast model into the JigCell Model Builder, shown in Figure 6.2. Each mutant strain changes parameters and initial conditions in the model to describe the nature of the mutation. The JigCell Run Manager stores these changes without requiring that the modeler produce multiple copies of the model. Figure 6.4 depicts a collection of runs for the budding yeast model that represents the mutant strains from Table 6.1.

Figure 6.4: Some mutant definitions for the budding yeast model of Figure 6.1 in the JigCell Run Manager.



Name	Parents	Changes	Simulator Settings	Description
M000B_WT_galactose	M000A_WT	mdt=150		
M001_cln2del	M000A_WT	ksn2'=0		
M002_GALCLN2	M000B_WT_galactose	ksn2'=galksn2', ksn2"=0.0		
M007_cln3del	M000A_WT	Dn3=0		
M008_GALCLN3	M000B_WT_galactose	Dn3=20		
M011_bck2del	M000A_WT	b0=0		
M012_5XBCK2	M000A_WT	b0=5.0 * b0		
M028_sic1del	M000A_WT	ksc1'=0, ksc1'=0		
M029_GALSIC1	M000B_WT_galactose	ksc1'=galksc1'		
M036_cdh1del	M000A_WT	kscdh=0, CDH1i=0, CDH1=0		
M037_CDH1constitutive	M000B_WT_galactose	kicdh"=0.0, kscdh=3.0 * kscdh		
M038_cdc6del47	M000A_WT	ksf6"=0, ksf6'=0, ksf6'=0		
M041_clb2del	M000A_WT	ksb2'=0, ksb2"=0		
M042_GALCLB2	M000B_WT_galactose	ksb2'=galksb2'		
M043_MULTI_GALCLB2	M042_GALCLB2	ksb2'=6.0 * galksb2'		*

The output of each run consists of measurements for all of the chemical species in the budding yeast model taken at a regular interval from the start time, which is time zero. The unit of time for the budding yeast model is minutes. One complete pass through the budding yeast cell cycle generally takes between 100 and 200 time steps. Since the initial cycles exhibit transitory behavior, modelers generally wish to examine only the later cycles. The automated model evaluation procedure in this dissertation uses 2000 time steps, approximately ten to twenty cycles, as the maximum amount of time to run the model. From an efficiency perspective, the automated model evaluation procedure should only request the minimum needed number of time-course measurements. However, the automated model evaluation procedure currently must request all of the time-course measurements in advance, and it is difficult to predict how many time-course measurements the automated model evaluation procedure needs.

In [41], Chen performed simulations using the WinPP simulator from G. Bard Ermentrout of the University of Pittsburgh Mathematics Department. This dissertation uses the similar XPP simulator [51], another simulation program that Ermentrout developed, for comparison. The simulator control settings in the Run Manager specify that the simulation program is XPP, that the integration method is the stiff solver, and that the end of simulation time is at 2000 time units. The other simulator control settings retain their default values from the JigCell wrapper service for XPP. All of the mutant strains use the same simulation program and simulator control settings.

Finally, the Run Manager requires the basal parameters and initial conditions for the budding yeast model, to which it applies the changes of the mutant strains before execution. Table 6.2 lists the basal initial conditions for the budding yeast model and Table 6.3 lists the basal parameters. The remainder of this chapter assumes that the modeler is using the basal parameters and initial conditions from these tables.

Table 6.2: Basal initial conditions for the wildtype budding yeast cell.

$[\text{MASS}] = 1.206019$	$[\text{APC-P}] = 0.1015$	$[\text{BUD}] = 0.008473$	$[\text{C2}] = 0.238404$
$[\text{C2P}] = 0.024034$	$[\text{C5}] = 0.070081$	$[\text{C5P}] = 0.006878$	$[\text{Cdc6}] = 0.10758$
$[\text{Cdc6P}] = 0.015486$	$[\text{Cdc14}] = 0.468344$	$[\text{Cdc14}]_{\text{T}} = 2.0$	$[\text{Cdc15}] = 0.656533$
$[\text{Cdc20}] = 0.444296$	$[\text{Cdc20}]_{\text{T}} = 1.91634$	$[\text{Cdh1}] = 0.930499$	$[\text{Cdh1}]_{\text{T}} = 1.0$
$[\text{Clb2}] = 0.1469227$	$[\text{Clb5}] = 0.0518014$	$[\text{Cln2}] = 0.0652511$	$[\text{Esp1}] = 0.301313$
$[\text{F2}] = 0.236058$	$[\text{F2P}] = 0.0273938$	$[\text{F5}] = 0.0000724$	$[\text{F5P}] = 0.0000791$
$[\text{Net1}] = 0.018645$	$[\text{Net1}]_{\text{T}} = 2.8$	$[\text{ORI}] = 0.000909$	$[\text{Pds1}] = 0.025612$
$[\text{PPX}] = 0.123179$	$[\text{RENT}] = 1.04954$	$[\text{Sic1}] = 0.0228776$	$[\text{Sic1P}] = 0.00641$
$[\text{SPN}] = 0.03$	$[\text{Swi5}] = 0.95$	$[\text{Swi5}]_{\text{T}} = 0.97$	$[\text{Tem1}] = 0.9$

Table 6.3: Basal parameters for the wildtype budding yeast cell.

$\mu = (\ln 2)/90$	$k'_{\text{a,15}} = 0.002$	$k''_{\text{a,15}} = 1$	$k'''_{\text{a,15}} = 0.001$	$k'_{\text{a,20}} = 0.05$
$k''_{\text{a,20}} = 0.2$	$k_{\text{a,apc}} = 0.1$	$k'_{\text{a,cdh}} = 0.01$	$k''_{\text{a,cdh}} = 0.8$	$k_{\text{a,mcm}} = 1$
$k_{\text{a,sbf}} = 0.38$	$k_{\text{a,swi}} = 2$	$k_{\text{as,b2}} = 50$	$k_{\text{as,b5}} = 50$	$k_{\text{as,esp}} = 50$
$k_{\text{as,f2}} = 15$	$k_{\text{as,f5}} = 0.01$	$k_{\text{as,rent}} = 200$	$k_{\text{as,rentp}} = 1$	$k_{\text{d,14}} = 0.1$
$k_{\text{d,20}} = 0.3$	$k'_{\text{d,b2}} = 0.003$	$k''_{\text{d,b2}} = 0.4$	$k_{\text{d,b2p}} = 0.15$	$k'_{\text{d,b5}} = 0.01$
$k''_{\text{d,b5}} = 0.16$	$k_{\text{d,bud}} = 0.06$	$k_{\text{d,cdh}} = 0.01$	$k_{\text{d,n2}} = 0.12$	$k_{\text{d,net}} = 0.03$
$k_{\text{d,ori}} = 0.06$	$k'_{\text{d,ppx}} = 0.17$	$k''_{\text{d,ppx}} = 2$	$k_{\text{d,spn}} = 0.06$	$k_{\text{d,swi}} = 0.08$
$k_{\text{d1,c1}} = 0.01$	$k_{\text{d1,f6}} = 0.01$	$k'_{\text{d1,pds}} = 0.01$	$k_{\text{d2,c1}} = 1$	$k_{\text{d2,f6}} = 1$
$k''_{\text{d2,pds}} = 0.2$	$k_{\text{d3,c1}} = 1$	$k_{\text{d3,f6}} = 1$	$k''_{\text{d3,pds}} = 0.04$	$k_{\text{di,b2}} = 0.05$
$k_{\text{di,b5}} = 0.06$	$k_{\text{di,esp}} = 0.5$	$k_{\text{di,f2}} = 0.5$	$k_{\text{di,f5}} = 0.01$	$k_{\text{di,rent}} = 1$
$k_{\text{di,rentp}} = 2$	$k_{\text{i,15}} = 0.5$	$k_{\text{i,apc}} = 0.15$	$k'_{\text{i,cdh}} = 0.001$	$k''_{\text{i,cdh}} = 0.08$
$k_{\text{i,mcm}} = 0.15$	$k'_{\text{i,sbf}} = 0.6$	$k''_{\text{i,sbf}} = 8$	$k_{\text{i,swi}} = 0.05$	$k'_{\text{kp,net}} = 0.01$
$k''_{\text{kp,net}} = 0.6$	$k_{\text{pp,c1}} = 4$	$k_{\text{pp,f6}} = 4$	$k'_{\text{pp,net}} = 0.05$	$k''_{\text{pp,net}} = 3$
$k_{\text{s,14}} = 0.2$	$k'_{\text{s,20}} = 0.006$	$k''_{\text{s,20}} = 0.6$	$k'_{\text{s,b2}} = 0.001$	$k''_{\text{s,b2}} = 0.04$
$k'_{\text{s,b5}} = 0.0008$	$k''_{\text{s,b5}} = 0.005$	$k_{\text{s,bud}} = 0.2$	$k'_{\text{s,c1}} = 0.012$	$k''_{\text{s,c1}} = 0.12$
$k_{\text{s,cdh}} = 0.01$	$k'_{\text{s,f6}} = 0.024$	$k''_{\text{s,f6}} = 0.12$	$k'''_{\text{s,f6}} = 0.004$	$k'_{\text{s,n2}} = 0$
$k''_{\text{s,n2}} = 0.15$	$k_{\text{s,net}} = 0.084$	$k_{\text{s,ori}} = 2$	$k'_{\text{s,pds}} = 0$	$k_{\text{s,ppx}} = 0.1$
$k_{\text{s,spn}} = 0.1$	$k'_{\text{s,swi}} = 0.005$	$k''_{\text{s,swi}} = 0.08$	$k''_{\text{s1,pds}} = 0.03$	$k''_{\text{s2,pds}} = 0.055$
$\epsilon_{\text{bud,b5}} = 1$	$\epsilon_{\text{bud,n2}} = 0.25$	$\epsilon_{\text{bud,n3}} = 0.05$	$\epsilon_{\text{c1,b2}} = 0.45$	$\epsilon_{\text{c1,b5}} = 0.1$
$\epsilon_{\text{c1,k2}} = 0.03$	$\epsilon_{\text{c1,n2}} = 0.06$	$\epsilon_{\text{c1,n3}} = 0.3$	$\epsilon_{\text{cdh,b2}} = 1.2$	$\epsilon_{\text{cdh,b5}} = 8$
$\epsilon_{\text{cdh,n2}} = 0.4$	$\epsilon_{\text{cdh,n3}} = 0.25$	$\epsilon_{\text{f6,b2}} = 0.55$	$\epsilon_{\text{f6,b5}} = 0.1$	$\epsilon_{\text{f6,k2}} = 0.03$
$\epsilon_{\text{f6,n2}} = 0.06$	$\epsilon_{\text{f6,n3}} = 0.3$	$\epsilon_{\text{ori,b2}} = 0.45$	$\epsilon_{\text{ori,b5}} = 0.9$	$\epsilon_{\text{sbf,b5}} = 2$
$\epsilon_{\text{sbf,n2}} = 2$	$\epsilon_{\text{sbf,n3}} = 10$	$J_{\text{20,ppx}} = 0.15$	$J_{\text{a,apc}} = 0.1$	$J_{\text{a,cdh}} = 0.03$
$J_{\text{a,mcm}} = 0.1$	$J_{\text{a,sbf}} = 0.01$	$J_{\text{a,tem}} = 0.1$	$J_{\text{d2,c1}} = 0.05$	$J_{\text{d2,f6}} = 0.05$
$J_{\text{i,apc}} = 0.1$	$J_{\text{i,cdh}} = 0.03$	$J_{\text{i,mcm}} = 0.1$	$J_{\text{i,sbf}} = 0.01$	$J_{\text{i,tem}} = 0.1$
	$J_{\text{n3}} = 6$	$J_{\text{pds}} = 0.04$	$J_{\text{spn}} = 0.14$	
$B_0 = 0.054$	$C_0 = 0.4$	$D_{\text{n3}} = 1$	$K_{\text{ez}} = 0.3$	$K_{\text{ez2}} = 0.2$

$$[\text{APC}]_{\text{T}} = 1 \qquad [\text{Cdc15}]_{\text{T}} = 1 \qquad [\text{Tem1}]_{\text{T}} = 1$$

$$k_{\text{bub2}} = \begin{cases} 1.0 & \text{if } [\text{ORI}] > 1 \text{ and } [\text{SPN}] < 1, \\ 0.2 & \text{otherwise.} \end{cases}$$

$$k_{\text{lte1}} = \begin{cases} 1.0 & \text{if } [\text{SPN}] > 1 \text{ and } [\text{Clb2}] > K_{\text{ez}}, \\ 0.1 & \text{otherwise.} \end{cases}$$

$$k_{\text{mad2}} = \begin{cases} 0.8 & \text{if } [\text{ORI}] > 1 \text{ and } [\text{SPN}] < 1, \\ 0.01 & \text{otherwise.} \end{cases}$$

6.2.1 Experimental Phenotype

An experimental phenotype describes the observed behavior of a mutant strain in the laboratory. Each experimental phenotype consists of a collection of classifiers. The automated model evaluation procedure uses both quantitative and qualitative classifiers to describe a mutant strain. The most important classifier is whether the mutant strain repeatably proceeds through the budding yeast cell cycle and viably reproduces. A mutant strain is viable if it completes the cell cycle while satisfying the following five rules.

- The model must complete the conceptual cell-cycle stages in the correct order and without repeating a stage.
  1. A drop in the concentration $[\text{ORI}]$, indicating relicensing of the origins of replication on the DNA.
  2. A rise in the concentration $[\text{ORI}]$, indicating the start of DNA replication.

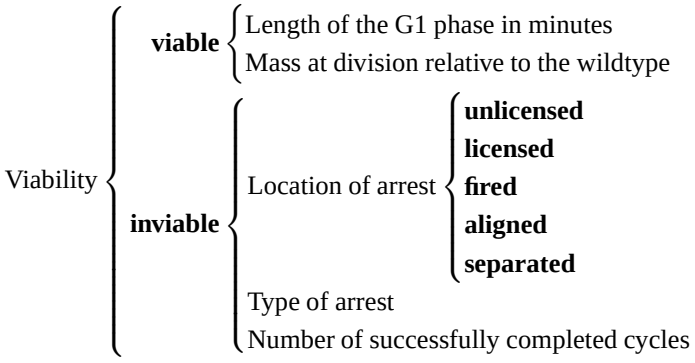
3. A rise in the concentration [SPN], indicating alignment of the chromosomes along the mitotic spindle.
4. A rise in the concentration [Esp1], indicating separation of the chromosomes.
5. A drop in [MASS], indicating the completion of cellular division.

- The start of mitosis in a mutant strain must occur before the time that the wildtype cell in the same growth medium requires to complete two iterations of the budding yeast cell cycle.
- The alignment of chromosomes along the mitotic spindle must occur while the concentration [Esp1] is below the threshold for the separation of the chromosomes.
- The concentration [BUD] must rise above the threshold for budding before cellular division occurs.
- The mass of a mutant strain must never grow beyond four times the mass at division of a wildtype cell in the same growth medium and must never shrink below one-fourth of that same wildtype mass at division.

As an additional requirement to the above rules for viability, the model for a viable mutant strain must exhibit a periodic, steady-state behavior. The phrase 'steady-state' is a mathematical misnomer although biological modelers readily employ that term. The concentrations of chemical species continuously change in a living cell, never reaching a steady-state. In this dissertation, the term 'steady-state behavior' instead refers to the fact that the concentrations of chemical species in a typical, viable cell have an exactly-repeating oscillatory characteristic. A mutant strain is viable only if the model satisfies the above rules for viability and the squared relative differences of both the mass at division and G1 phase length in minutes are less than five percent.

Figure 6.5 shows the experimental phenotype for the budding yeast model. The viability of the mutant strain determines the additional classifiers that the automated model evaluation procedure collects.

Figure 6.5: Phenotype description for an experimental observation of a budding yeast mutant strain.



In viable cells, the commonly observed properties include timings of the budding yeast cell cycle events and the mass of the cell at division. The automated model evaluation procedure times the cell cycle using the duration of the G1 phase. Adding timing data for additional cell cycle events to the automated model evaluation procedure is not difficult, although experimentalists rarely can collect accurate in-vivo time measurements. Experimental observations of the mass at division often are in relative terms, such as "the mutant cells are roughly twice the size of a typical wildtype cell". Therefore, the automated model evaluation procedure works with the ratio between the mutant strain and wildtype masses in the same growth medium. Although the automated model evaluation procedure does not account for errors in the experimental observations directly, the modeler can adjust control parameters in the objective function to indicate the tolerable range for an observation.

In inviable cells, the automated model evaluation procedure records where in the budding yeast cell cycle the cell arrested and for what reason. The location of arrest in the cell cycle uses the conceptual stages of the model. The reason for arrest uses the rules for viability given above. Each reason for arrest has a numeric code in the software, and the JigCell Comparator provides details about the indicated problem in the user interface. Some mutant strains

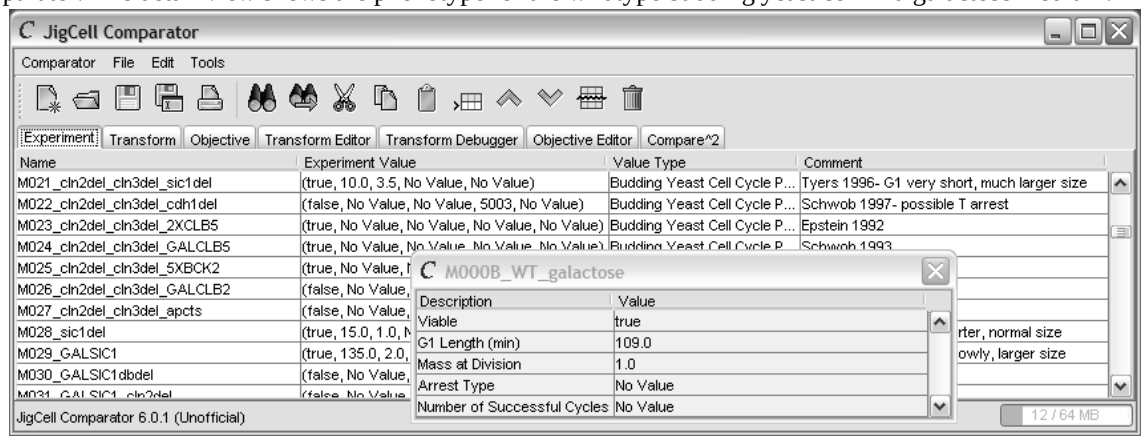
can successfully complete the cell cycle once before becoming stuck, such as the mutant strain *CLB5-dbΔ sic1Δ* [74]. The automated model evaluation procedure records the number of cycles successfully completed before arrest to distinguish this condition. The description of an arrested cell has a hierarchical comparison.

1. Given two experimental phenotypes of arrested cells, first compare the locations of arrest.
2. If the two phenotypes arrest in the same stage, then compare the types of arrest.

Every type of arrest is equally unlike all of the other types. The automated model evaluation procedure can accommodate a new type of arrest without worry of a combinatorial explosion with respect to the number of error conditions.

JigCell displays the phenotype of a mutant strain as the union of the classifiers for **viable** and **inviable** mutant strains. Figure 6.6 shows a portion of the experimental data for the budding yeast model in the JigCell Comparator.

Figure 6.6: Some experimental data for the mutant strains of the budding yeast model of Figure 6.1 in the JigCell Comparator. The detail view shows the phenotype for the wildtype budding yeast cell in a galactose medium.



6.2.2 Data Transformation

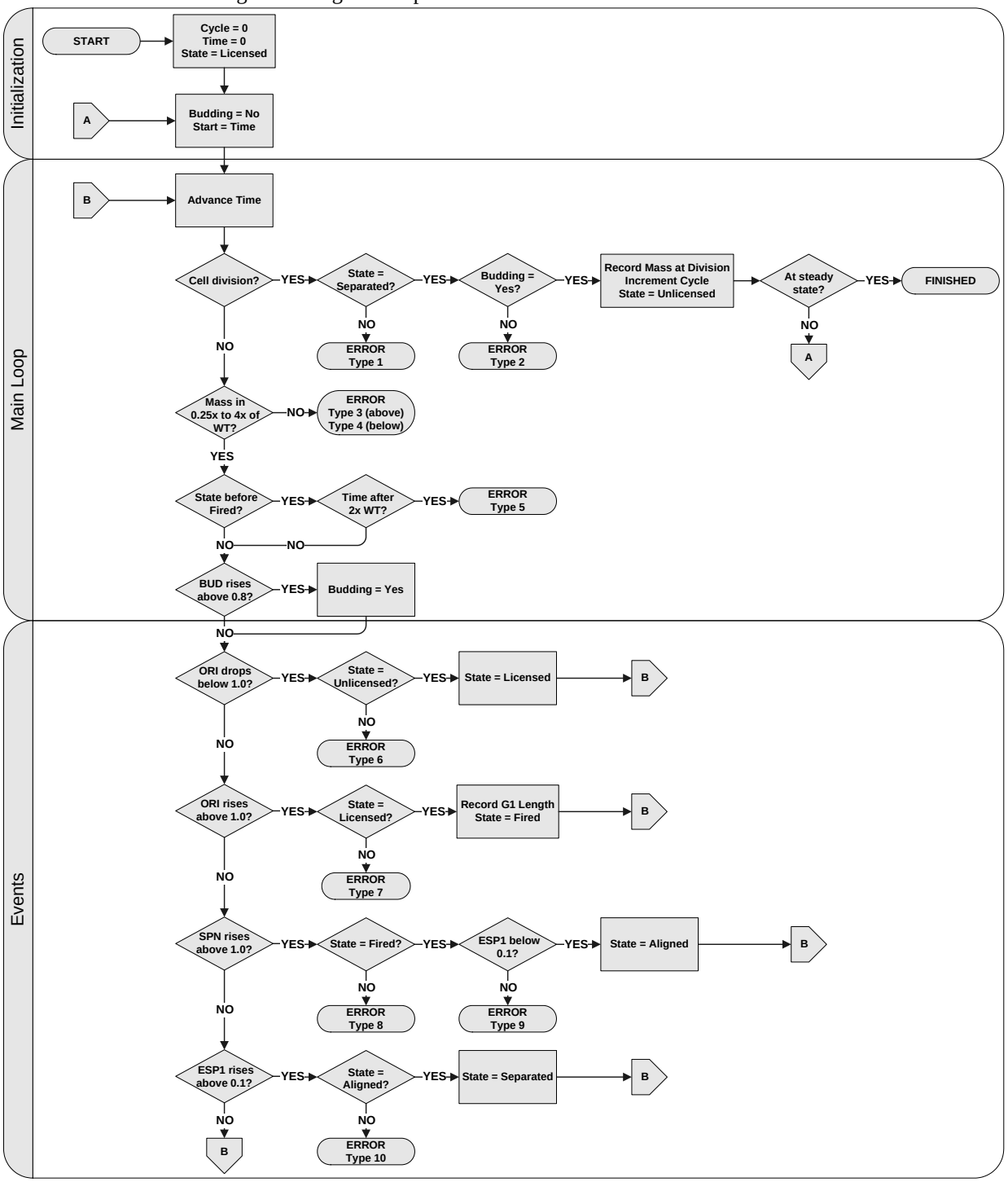
The time-course output of a simulation program is not directly comparable with an experimental phenotype. When evaluating the budding yeast model by hand, modelers must predict a phenotype from the model time course. The automated model evaluation procedure requires a similar algorithm for predicting the phenotypes of mutant strains. As Section 4.4 mentioned, JigCell calls these algorithms transforms. Figure 6.7 shows the transform that the automated model evaluation procedure uses for the budding yeast model.

The budding yeast transform has three sections: initialization, the main loop, and event checking. The initialization section handles bookkeeping at the start of evaluation and at the beginning of each iteration of the budding yeast cell cycle. Modelers specify the initial conditions of the model, and these initial conditions can correspond to any stage of the budding yeast cell cycle. The basal initial conditions given in Table 6.2 configure the budding yeast model just after licensing of the origins of replication of the DNA. Due to these basal initial conditions, the budding yeast transform must treat the first iteration of the cell cycle differently by starting in the **licensed** stage. Successive iterations of the budding yeast cell cycle begin in the **unlicensed** stage.

The main loop monitors the growth and division of the budding yeast cell. Each time through the main loop, the transform requests the next available time-course measurement. First, the transform checks whether cellular division occurred since the previous measurement. Mass is a smooth, exponentially increasing variable in the budding yeast model. Cellular division causes a sharp, discontinuous drop in the mass when the mother and daughter cells separate. Therefore, a drop in mass indicates that cellular division occurred during the previous interval of time.

If the transform detects that the budding yeast cell divided, then it further checks that the cell completed all of the stages of the cell cycle and formed a bud. The auxiliary variable BUD measures the budding response in the cell, which can occur at any time within the budding yeast cell cycle. If the budding yeast cell viably completed the cell cycle and

Figure 6.7: Data transformation for the budding yeast model of Figure 6.1 that predicts a phenotype from the time-course output of the model. The configuration of the transform corresponds to basal initial conditions in Table 6.2 that start the cell after relicensing of the origins of replication of the DNA.



budded, then the transform checks whether the model is exhibiting periodic, steady-state behavior. If the model meets the steady-state criteria, then the transform halts and declares that the mutant strain is viable. Otherwise, the model proceeds with the next iteration of the cell cycle and the transform continues to request time-course measurements.

The budding yeast model predicts that cells stuck in the cell cycle experience unbounded growth. This is physically impossible as an extremely large cellular mass damages the membrane enclosing the cell, and the cell eventually dies. The transform uses the viability rule that restricts the size of a budding yeast cell to detect this condition in mutant strains. With the previously given basal parameters and initial conditions, the minimum budding yeast cell mass is approximately 0.6, and the maximum mass is approximately 10.0. The transform computes the actual minimum and maximum sizes during runtime by simulating the wildtype cell in the appropriate growth medium.

Finally, the transform checks for the cell cycle events that represent stage transitions. At each transition between cell cycle stages, the transform first ensures that the model is making a permissible transition. The cell cycle does not ordinarily permit a cell to go backwards in the cell cycle, repeating a stage, or to suddenly skip ahead in the cell cycle. Such illegal stage transitions in the budding yeast cell cycle are fatal, and the transform declares that mutant strains with these temporal problems are inviable. After the transform validates a transition, the model enters a new stage of the cell cycle, and the transform resumes examination of further time-course measurements.

The transform requires that the modeler specify a simulation program that can detect the discrete event transitions in the budding yeast model. The XPP simulator supports discrete events with the 'global' statement [51]. Other simulation programs, such as LSODAR (Livermore Solver for Ordinary Differential equations, with Automatic method switching for stiff and non-stiff problems, and with Root-finding) [38], have similar support. The JigCell simulator wrapper service for these programs translates the discrete event specification to the native language of the simulator.

When a discrete event transition occurs, the simulator employs a root-finding algorithm to determine precisely the moment that the event trigger condition was first true, independent of the output granularity of the time course. The transform can discover discrete event transition times either by examining the time course retrospectively or by receiving notice of discrete event transitions from the simulation program. Having the transform determine the times of discrete event transitions retrospectively is generally more portable across simulators and is the only method that XPP supports. Having the simulation program report the times of discrete transitions is more efficient.

Two of the discrete event transitions require that the transform apply special handling to correctly validate the rules of viability in budding yeast cells. First, some of the mutant strains linger in the G1 phase beyond the time that the rules of viability permit. In this case, the transform judges that the cell effectively blocks in the G1 phase. Second, at the end of the cell cycle, separation of the chromosomes must occur after the alignment of the chromosomes along the mitotic spindle and before cellular division. The indication of chromosome separation is a rise in the concentration $[\mathrm{Esp1}]$ above a threshold. Hence, the transform requires that the chromosome alignment event must occur when the concentration $[\mathrm{Esp1}]$ is less than this threshold. These two checks distinguish mutant strains with particular defects that are otherwise difficult for the modeler to detect.

6.2.3 Objective Function

The budding yeast objective function takes the experimental phenotype that practitioners observed in the laboratory and the predicted phenotype that the transform generates from the model to compute a non-negative, real-valued score. Call the experimentally observed phenotype O, the predicted phenotype of the model P, and define the objective function $D : O \times P \longmapsto \mathbb{R}_{\geq 0}$. Table 6.4 lists the entries of the observed and predicted phenotypes, which are the six-tuples of classifiers that Figure 6.5 defined.

Table 6.4: Symbols and definitions for the observed and predicted phenotypes.

O_v, P_v	$\equiv$	Viability
O_g, P_g	$\equiv$	Steady state length of the G1 phase
O_m, P_m	$\equiv$	Steady state mass at division
O_a, P_a	$\equiv$	Location of arrest
O_t, P_t	$\equiv$	Type of arrest
O_c, P_c	$\equiv$	Number of successfully completed cycles

A score of zero from the objective function indicates a perfect match between the observed and predicted phenotypes. Larger scores from the objective function indicate increasingly worse matches between the observed and predicted phenotypes. In the JigCell Comparator, the special value ∞ from an objective function indicates that the objective function is not computable due to an error in the configuration of the comparison. As Section 4.4 mentioned, objective function scores have no absolute scale. The modeler must calibrate the objective function by specifying a threshold value that marks the end of the acceptable range of scores. The JigCell Comparator highlights scores that exceed the threshold value given by the modeler. The threshold value indicates the confidence of the modeler that an experimental observation and objective function score together predict the validity of the model.

Due to a lack of experimental observations, an observed phenotype may not have all of the expected classifiers. If O_v is missing, then no comparison is possible and $D(O,P) = \infty$. If $O_v =$ **inviable**, $P_v =$ **viable**, and O_c is missing, then $D(O,P) = \omega_v$, the same as if O_c were zero. Otherwise, the objective function simply drops the contribution of terms with missing classifiers. The objective function score when all of the classifiers are present is

$$D(O,P) = \begin{cases} \omega_g * (\frac{O_g - P_g}{\sigma_g})^2 + \omega_m * (\frac{\ln \frac{O_m}{P_m}}{\sigma_m})^2 & \text{if } O_v = \textbf{viable} \text{ and } P_v = \textbf{viable}, \\ \omega_v * \frac{1}{1+P_c} & \text{if } O_v = \textbf{viable} \text{ and } P_v = \textbf{inviable}, \\ \delta_{O,P} + \omega_c * (\frac{O_c - P_c}{\sigma_c})^2 & \text{if } O_v = \textbf{inviable} \text{ and } P_v = \textbf{inviable}, \\ \omega_v * \frac{1}{1+O_c} & \text{if } O_v = \textbf{inviable} \text{ and } P_v = \textbf{viable}, \end{cases}$$

where $\delta_{O,P}$ is the two-step delta function that scores experimental phenotypes with arrest codes

$$\delta_{O,P} = \begin{cases} \omega_a & \text{if } O_a \neq P_a, \\ \omega_t & \text{if } O_a = P_a \text{ and } O_t \neq P_t, \\ 0 & \text{if } O_a = P_a \text{ and } O_t = P_t. \end{cases}$$

The modeler tunes the objective function by adjusting the relative importance of the classifiers. The control parameters for the objective function given by Table 6.5 place the threshold value around ten for an unacceptable fit between the model and experimental data. For example, using these control parameters and this threshold value, the mass at division for a mutant strain can vary by a factor of two as long as the length of the G1 phase for that mutant strain is within $3 * \sigma_g$ minutes of the experimental observation.

Table 6.5: Definitions and standard values of the objective function constants.

Symbol	Definition	Value
ω_v	Viability weight	40.0
ω_g	Length of G1 phase weight	1.0
σ_g	Length of G1 phase scale	10.0
ω_m	Mass at division weight	1.0
σ_m	Mass at division scale	$\ln 2$
ω_a	Location of arrest weight	10.0
ω_t	Type of arrest weight	5.0
ω_c	Number of successfully completed cycles weight	10.0
σ_c	Number of successfully completed cycles scale	1.0

6.3 Results

The automated model evaluation procedure that this chapter described quickly evaluates the acceptability of an instance of the budding yeast model. This case study compares the performance of an expert modeler using the classic tool, WinPP, to evaluate the budding yeast model manually against the automated model evaluation procedure in JigCell.

To make the evaluation fair, JigCell uses the same simulation program that the expert modeler used. Table 6.6 presents the results. Afterwards, this section discusses how using different simulation programs with JigCell affects the results.

Each case study trial consisted of a small number of parameter changes along with a list of ten mutant strains to evaluate for acceptability with the given parameters. The expert modeler first evaluated whether each mutant was viable. For viable mutant strains, the expert modeler provided the length of the G1 phase in minutes and the mass at division. The study requested unscaled values for the mass at division rather than requiring that the expert modeler compute the ratio between the mutant strain and wildtype masses. For inviable mutant strains, the expert modeler provided the reason that the mutant strain died. This reason for arrest was descriptive, such as 'telophase arrest', and did not require technical justification. Prior to the start of the trial, the expert modeler received instructions that the acceptable tolerance of numerical results was to within 10% of the actual value. 91% of the answers that the expert modeler gave met this quality requirement. The remaining answers were within 20% of the actual value.

The study classified tasks according to whether the task required the continuous presence of the user. The simulation runs that the expert modeler requested during manual model evaluation complete in around five seconds. This is not enough time for the user to concurrently perform another task. The JigCell Comparator batches simulation runs into a contiguous block. The user could leave the software unattended and perform other work during this time.

Fixed costs, such as launching the applications, loading data files, and entering parameter changes amortize over the number of mutant strains. The results of this study present the amortized times for these activities.

Table 6.6: Budding yeast mutant strain evaluation times when the expert modeler employs manual versus automated model evaluation. Values are average times in seconds spent per mutant. $(*)$ indicates an amortized time.

	Manual	Automated
Startup $*$	0.6	5.0
Basal parameter entry $*$	21.8	5.4
Mutant parameter entry	24.1	-
Simulation	27.5	14.3
Evaluation	86.5	9.7
Time modeler present	160.4	20.2
Total time	160.4	34.5

For manual model evaluation, the expert modeler used WinPP, which is a Windows-based version of the XPP simulator by Bard Ermentrout. The expert modeler received a model file that contains the differential equations of the budding yeast model along with the basal parameters and initial conditions. The expert modeler entered parameter changes directly into the model file, replacing the old values. WinPP does not have a mechanism for storing the parameter and initial condition changes of each mutant strain. The expert modeler entered these changes from memory. A user who could not recall the changes for a mutant strain would need to consult a lookup table.

A typical model evaluation sequence starts with the expert modeler entering the mutant strain parameter and initial condition changes. The expert modeler then produced time-course plots of the model. Often, the expert modeler requested multiple plots before providing an answer. The time from starting a simulation until WinPP produced a plot is 'simulation' time in Table 6.6. The time that the expert modeler spent studying these plots is 'evaluation' time.

For automated model evaluation using JigCell, the expert modeler received the SBML model, run, and comparison files. The run file has all of the parameter and initial condition changes for the mutant strains, so the user does not need to provide these values. The collection of mutant descriptions required three hours to create originally. Since then, modelers have performed billions of simulations using that run file.

When using JigCell, the time from starting a comparison until the objective function computes a score, including the time for simulation and running the data transformation, is 'simulation' time. The time that the expert modeler spends reviewing the output from the JigCell Comparator is 'evaluation' time. Although JigCell used the same simulator and performed extra processing, the simulation time for automated model evaluation is smaller than for manual model evaluation. Typically, the expert modeler performing manual model evaluation requested a plot of the first 500 minutes of simulation. If the mutant strain had a lengthy transient phase, then the expert modeler needed further simulation to view the steady-state behavior. The expert modeler might request a time-course plot with the start of a

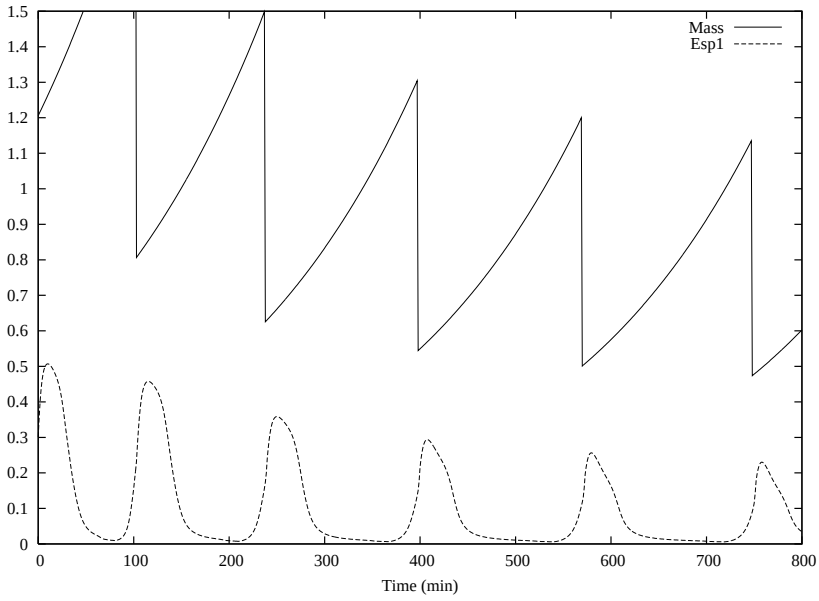
steady-state cycle shifted to time zero. Additionally, the expert modeler might redo runs after making an error entering the parameters and initial conditions. In contrast, JigCell always made a single request for the first 2000 minutes of time-course measurements.

The study did not record the number of self-detected and corrected errors that the expert modeler made. However, the expert modeler described when a mutant strain required more work due to the perception of errors. The time for evaluating a mutant strains averaged more than twice as long when the expert modeler believed that there was an error. The expert modeler restarted evaluation of 14% of the mutant strains due to error.

Expected gains in correctness when using the automated model evaluation procedure

The use of a transform in automated model evaluation enables the discovery of subtle defects that modelers find difficult to identify visually in a time course. For example, the transition from the **aligned** stage to the **separated** stage occurs when the concentration $[\mathrm{Esp1}]$ rises above a threshold value. In a viable mutant strain, chromosome separation must take place before cellular division. Figure 6.8 shows the concentration $[\mathrm{Esp1}]$ and $[\mathrm{MASS}]$ versus time over five complete cycles of the GAL-CLN3 mutant strain with the given basal parameters and initial conditions.

Figure 6.8: The concentration $[\mathrm{Esp1}]$ and $[\mathrm{MASS}]$ versus time for 800 minutes in the GAL-CLN3 mutant strain with the given basal parameters and initial conditions. The transform marks the GAL-CLN3 mutant strain inviable because cellular division occurs before chromosome separation in the fifth and later iterations of the cell cycle.



During the first iteration of the budding yeast cell cycle, the concentration $[\mathrm{Esp1}]$ rises above its threshold value more than five minutes before the drop of $[\mathrm{MASS}]$. Successive iterations of the cell cycle do not appear appreciably different in their relative timings of these events. However, during the fifth iteration of the cell cycle, chromosome separation in the GAL-CLN3 mutant strain takes place fractionally after cellular division. By the twelfth iteration of the cell cycle, chromosome separation occurs more than five minutes after cellular division.

The transform marks the GAL-CLN3 mutant strain inviable after the model completes four successful iterations of the budding yeast cell cycle. The mutant strain phenotype is arrest in the **aligned** stage with the reason for arrest that cellular division occurred before the completion of all required cell cycle events. Figure 6.7 indicates that this type of arrest has error code number one. A modeler that is examining the time course manually would not recognize this defect in the mutant strain without considerable difficulty. The modeler is therefore likely to incorrectly declare that the mutant strain is viable. One complete evaluation of the budding yeast model for a particular guess of parameters

during parameter twiddling requires many such checks for viability. Moreover, the modeler must repeat each viability check over the examined cycles for all of the mutant strains.

Expected gains in time when using the automated model evaluation procedure

This study gives costs for startup and basal parameter entry that are larger than those expected for normal work. The expert received mutant strains in sets of ten to eliminate fatigue as a factor. When evaluating all 131 mutant strains, these costs contribute less to the total time. Extrapolating to sets of 131 mutant strains, manual model evaluation requires 139.8 seconds per mutant strain. Automated model evaluation requires 24.9 seconds per mutant strain, with the expert modeler present for 10.5 seconds.

Changing the numerical integration routine would lead to further reductions in evaluation time for JigCell. For this study, the automated model evaluation procedure used the XPP simulation program to provide parity with manual model evaluation. High performance simulators, such as LSODAR [38], achieve the same numerical accuracy on the budding yeast model while expending significantly less effort. It is difficult to use a different simulator with the manual model evaluation procedure as modelers rely on the user interface of WinPP. Furthermore, totally eliminating simulation time from the manual model evaluation procedure does not reduce the average mutant strain evaluation time below 112 seconds. During parameter estimation, which removes all human intervention and user-interface support, automated model evaluation against the entire suite of mutant strains requires less than 15 seconds, or approximately 0.1 seconds per mutant strain on average. Automated model evaluation is capable of a more than 1000-fold improvement in evaluation time over manual model evaluation.

Chapter 7

Conclusions

This dissertation covered the construction of models of biochemical reaction networks through mathematical methods, primarily using ordinary differential equations. Using the previously existing modeling tools, a skilled modeler could create accurate mathematical models of biological systems with moderate complexity and gain an understanding of dynamical processes. However, this process of building models is tedious and error-prone. An expert modeler spends much time turning a biological idea into a mathematical model. Furthermore, this modeling process is not suitable for novice modelers, who must typically spend several months building models before they become proficient enough to attempt original work. Teaching novice modelers how to build models is difficult because there are few resources that explain how experienced modelers build models.

This dissertation describes how to make the construction of models of biological systems easier and documents the process that experts use to build models. Much of the tedious mathematical bookkeeping that modelers previously performed manually is amenable to computer automation. This dissertation introduces computational software that performs this automation. Further progress in biological modeling requires the use of better tools and modeling processes. Investigations of complex eukaryotic organisms requires models at least an order of magnitude larger than the current state of the art. Existing modeling processes and tools do not scale to models of this size. Already, bringing a model from conception to publication requires several years of concerted effort by skilled modelers.

The computational software that this dissertation presents reduces the amount of time that an expert modeler spends on bookkeeping and model evaluation. Model evaluation was a particularly significant bottleneck in the model development process. Moreover, modelers were not performing regular model evaluation because of its expense. Alleviation of the model evaluation bottleneck should lead directly to a reduction in the construction time for biological models and an increase in operational correctness.

The remainder of this section summarizes the major contributions and conclusions of this dissertation. Section 7.1 summarizes the contributions of this dissertation. The major contributions in this dissertation are the observations of the original modeling process, the construction of the revised modeling process, the JigCell modeling environment, the requirements specification for the JigCell modeling environment, and the quantitative measurements of the impact that the JigCell modeling environment has on model evaluation. Section 7.2 describes the history of the JigCell project and presents a postmortem report of the software engineering successes and failures of the project. Section 7.3 examines the user requirements from Chapter 5 in light of the software engineering difficulties that the JigCell project faced contemporarily.

Contents

7.1 Contributions

Documentation for how biological modelers build models

Section 3.1 described the original modeling process observed in the modeling laboratory of John Tyson at Virginia Tech. The literature has few good examples describing how modelers, particularly biological modelers, work to build models. This modeling group did not have existing written documentation that describes their modeling process.

The original modeling process came from the observational study of the modeling group working in the Tyson laboratory to build models over a period of several months. Documenting the modeling process of biological modelers is important for understanding the activities that limit the capabilities of modelers and for calibrating any proposed new modeling process. Much of the work in this dissertation and similar modeling and simulation studies is not possible without this kind of observational study.

A new modeling process for constructively building biological models

Section 3.3 described the revised modeling process, a new modeling process based on modeling methodology and design but styled after the original modeling process in Section 3.1. The revised modeling process makes the process of building biological models more amenable to computer automation.

Simply automating the original modeling process is not effective. For example, automating model evaluation in the original modeling process leaves the modeler in-the-loop making parameter guesses during parameter twiddling. The modelers optimized the original modeling process for throughput given their existing computational resource restrictions. Vastly expanding the computational capabilities of modeling tools creates the potential for new modeling activities, such as parameter estimation, which do not integrate easily with the original modeling process.

The revised modeling process replicates the inputs and products of the original modeling process but optimizes the modeling process with computer automation in mind. The revised modeling process provides a blueprint for building new modeling tools that advance the state of the art in modeling.

More teams that build modeling tools should document their corresponding modeling processes. Having the documented modeling process of a modeling tool allows for analysis of the design of the tool without confusion with the implementation of the tool. Moreover, biological modeling is a still-expanding field, and modeling tool builders should expect that modelers will demand support for new modeling activities. Unless the modeling tool builders record their modeling process, the adaptation of modeling tools to new activities is unnecessarily difficult.

Modeling software that can handle large-scale biological models

Chapter 4 described the JigCell modeling environment for building models of reaction networks. Section 7.2 details the software engineering lessons learned by building the JigCell modeling environment. Furthermore, the JigCell modeling environment closely aligns with the revised modeling process and can act as a research environment for studying the practice of modeling.

The biological modeling community is in sore need of tools for efficiently, reliably, and repeatably building large models of biochemical reaction networks. The JigCell applications support larger models than comparable modeling tools, making the JigCell applications especially suitable for particular modeling efforts. Moreover, modelers can readily adapt much of the JigCell modeling environment to other modeling domains. The JigCell Model Builder and Run Manager are specific to continuous reaction-oriented models, but these applications are only specific to the domain of biology to the extent of choices of words and concepts for displaying information in the user interfaces. The JigCell Comparator is not specific to any particular modeling domain.

Development of the JigCell modeling environment also gave insight into the design of the revised modeling process. The revised modeling process remains weak in the areas of problem formulation and model accreditation. Researchers can test problem formulation and model accreditation in the JigCell modeling environment experimentally before integrating these activities into the revised modeling process. Experimenting with an implementation can aid the development of the modeling process when the available domain experts do not engage in a modeling activity often enough to support direct study.

Requirements gathering and analysis for biological modeling tools

Chapter 5 described the process of gathering methodological, domain, and user requirements for biological modeling software and applied those requirements to the JigCell modeling environment. The methodological and domain requirements are not specific to biological modeling, making the term 'domain requirements' a misnomer. Other modeling environments can apply these requirements to better support domain expert users.

The user requirements are specific to biological modeling and to the revised modeling process. Although other modeling efforts could make use of these user requirements to measure their support for the modeling activities that this dissertation identifies as important, the user requirements act primarily as a benchmark for the progress of the JigCell modeling environment. These user requirements help align the developers of JigCell with the originally stated needs of the biological modelers. Section 7.3 gives a caveat to the use of these user requirements.

Practical, measurable impact of using the modeling software

Chapter 6 quantified the efficacy of the JigCell modeling environment by building an automated model evaluation procedure in JigCell and applying that evaluation procedure to a recent model of cell cycle control in budding yeast. This case study provided the first quantifiable demonstration that the revised modeling process and JigCell modeling environment help biological modelers build models. Early measurements of the original modeling process showed that 'parameter twiddling', the process of repeatedly guessing new parameter values and checking model fitness, was a major bottleneck for model development. Automated model evaluation speeds up checking model fitness by more than 1000-fold, and parameter estimation greatly speeds up the guessing of new parameter values.

The results of this case study demonstrate that automated model evaluation works for large-scale models with complex outputs, such as cell cycle models, which modelers previously evaluated manually. Moreover, further work shows that parameter estimation is viable for models of this scale and complexity, indicating that computer automation can vastly reduce model development time for models at the state of the art of biology.

7.2 Software Engineering Experiences

The JigCell project started with the goal of building a modeling tool for a particular group of biological modelers. The set of planned features quickly expanded to include more applications and user groups. Development of the JigCell modeling environment spanned from early 2001 through, so far, late 2005, a period of nearly sixty months. During this time, the development team remained small, generally averaging around five developers working part-time.

Marc Vass contributed the first code to the JigCell project, a database-driven model builder, in early 2001. The project restarted in late 2001 with new funding from the DARPA BioSPICE [30] program. Vass started work on a new model builder, an interface to the XPPAUT simulation program by G. Bard Ermentrout [52], and a tool for executing models. Nicholas Allen started work on infrastructure and a tool for performing model analysis. Jason Zwolak started work on an interface to the LSODAR simulation program [38] and a tool for parameter estimation.

The scope of the JigCell project expanded from a simple modeling tool to a suite of applications, backed by a database of historical model information, that would provide an 'end-to-end experience' for building models of biochemical reaction networks. This suite of tools was nearly identical in concept to the applications that Chapter 4 described, consisting of a Model Builder, Run Manager, and Comparator.

By the end of 2001, Vass produced an early version of the Model Builder, and Allen produced the build system for the JigCell project and a barely functional Comparator. At this time, Allen was integrating all of the developed code and third-party libraries into the build system by hand, acting as the 'build master' and performing integration. Within the first few months of 2002, Vass created the Run Manager and integrated the XPPAUT simulator and Allen developed the Comparator to the point that a modeler could build, simulate, and analyze a model.

In May 2002, the BioSPICE program made an initial release of software tools. Allen struggled to create a working build of the JigCell modeling environment with the Model Builder, Run Manager, and Comparator, primarily due to the late start of integration. Allen and Vass quickly created an installer, documentation, and a test plan for the JigCell modeling environment. Although Allen ultimately delivered the JigCell modeling environment to SRI past the

scheduled deadline, few of the other tool builders in the BioSPICE program delivered contributions. Due to this early success, the JigCell project received considerable latitude within the BioSPICE program.

Vass used the Open Agent Architecture [89] by SRI to interface the JigCell modeling environment with simulators, primarily due to outside funding requirements. This was deleterious, causing significant problems with performance, integration, and installation, and leading to a generally unsatisfactory user experience. Originally, Vass and Allen could only get the simulator to run on a single machine, forcing users to run all simulations remotely over a network. Even simple models required minutes to perform a simulation. After feedback by Allen and Cliff Shaffer to SRI, SRI produced a new version of the Open Agent Architecture that addressed some of these problems.

At approximately the same time, Michael Hucka and Andrew Finney produced a major revision to the SBML language, SBML Level 2 Version 1 [54]. The JigCell Model Builder was a prerelease adopter of this new version of the SBML language. Vass produced the first parser implementing SBML Level 2 Version 1 and modified the Model Builder to load and save models in this language, replacing the custom 'mechanism' format that the Model Builder used previously. This transition to the SBML language caused significant instability and data-loss problems in the Model Builder, which would plague the JigCell project for more than two years.

During 2002, the development of the JigCell project slowed. The project source code increased to more than one million lines of code, including third-party libraries compiled during the build process. Allen still produced builds by manually integrating source code contributions. Times for compiling the JigCell project from a clean source tree exceeded 40 minutes, slowing the rate of development and testing. Often, the integration of a new contribution required several complete compiles and more than a day of labor. Allen modified the IBM Jikes Java compiler [1] to handle the JigCell project source code and developed an automatic syntax rewriter to fix the remaining problems, reducing the compile time by more than two-thirds. Although typical machine speeds have increased since, the JigCell project continues to include its own Java compiler, a rarity today for projects of this size.

In 2003, the JigCell project suffered a variety of setbacks. The long-planned database, parameter estimation tool, and new simulator never materialized. Although the database was ancillary to the functions of the JigCell project, modelers sorely needed the new simulator based on LSODAR to replace the aging XPPAUT simulator. The JigCell project repeatedly promised parameter estimation to modelers, to truly complete the end-to-end experience, but similarly never delivered this tool in the JigCell modeling environment. Allen completed the majority of the Comparator during this time. However, Vass left the JigCell project before finishing updates to the Model Builder and Run Manager. User feedback turned strongly negative as the data-loss bugs from converting the Model Builder to use SBML instead of the mechanism format made work impossible.

The JigCell project began to nearly miss the regular, semiannual BioSPICE releases due to low code quality despite the small number of features that the developers were adding. During that year, BioSPICE also changed its integration method from the Open Agent Architecture to the BioSPICE Dashboard, based on the Java NetBeans development environment [30, 97]. Despite a lack of user interest in the BioSPICE Dashboard, Allen and Vass spent several months modifying the JigCell modeling environment to support the BioSPICE Dashboard. Ultimately, few users employed the BioSPICE Dashboard to run the JigCell applications.

A succession of new personnel worked on the JigCell project between 2002 and early 2004, none ultimately contributing code. The project hit a low point in early 2004, making no user-visible progress for several months. The JigCell project began integrating a tool for project management that would connect the applications together and manage data. However, the project management component later ended due to a lack of developer time.

In the middle of 2004, the developers of the JigCell project decided to totally overhaul their software engineering methods. Thomas Panning, who originally joined the JigCell project to work on parameter estimation, set up and administrated version control and bug tracking systems for the project. For the first time, developers could access the latest version of the JigCell project source code without Allen performing the integration manually. After a several month period of acclimation, the JigCell project developers began using bug tracking and version control effectively, working to catalog the defects in need of correction. By the end of 2004, the total number of identified defects numbered several hundred, including dozens of critical bugs.

Subsequent to the introduction of new software engineering methods, Allen took over day-to-day management of the software development. Allen targeted for total replacement the areas of the source code with the highest concentrations of bugs. During 2004, Allen developed a new SBML parser and a new integration method for simulators, supplanting the Open Agent Architecture and BioSPICE Dashboard approaches with a simpler, direct connection

method. Panning worked on manually porting the model data stored in JigCell data files to the completed pieces of the parameter estimation tool. Ranjit Randhawa joined the JigCell project and began rewriting the Model Builder. The new Model Builder would use the SBML Parser for all data storage and file handling.

By the middle of 2005, Randhawa completed rewriting the Model Builder, reducing its code size by more than eighty percent. The total number of identified bugs dropped below fifty, and for the first time since the start of bug tracking, there was a release containing no known critical bugs. The project accelerated from milestones of several months in length to producing regular, functioning releases every six weeks. Allen rewrote the Run Manager with a new, simpler file format, and Panning converted the data for the large model of budding yeast by Katherine Chen [41] for parameter estimation. Panning began performing regular production runs on the budding yeast model. Additionally, Panning created a nightly build system that automatically compiled the latest version of the software and posted a working release online each night.

Development of the JigCell modeling environment continues into late 2005. The JigCell project source code dropped nearly twenty percent in size over the previous year and the developers dramatically reduced the number of known defects. Emery Conrad contributed an interface to the SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers) simulator [67], lessening the need to rely on the XPPAUT simulation engine. Hucka and Finney developed a test suite of SBML models for simulation. Allen connected the SBML test suite with the JigCell modeling environment simulators to monitor their quality and compliance with the SBML language.

The focus of the JigCell project changed during 2005 from correcting critical bugs to improving the user experience of the applications. The user base of the JigCell modeling environment imploded because of the data-loss problems of 2003 and 2004. Regaining the trust of that population of users is a difficult task. The JigCell project has now lost the large head start that it enjoyed in 2002 over similar, contemporary modeling projects. The continued success of the JigCell project depends on regularly increasing the quality of the software and attracting new users.

Allen and Panning plan to leave the JigCell project shortly after the end of 2005. Randhawa, along with other new JigCell project developers, will continue to develop JigCell. Zwolak continues to work on an interface to the LSODAR simulation program and a tool for parameter estimation.

7.2.1 What Went Right

Choosing the right tools and languages for the job

From the beginning, the JigCell project used Java as its primary programming language. Rapid development of the applications during 2001 and 2002 allowed the JigCell modeling environment to capture the attention of the modeling community before other contemporary efforts were functional. However, the JigCell project could not tolerate the slower speed of scripting languages due to numerical computation needs. Moreover, it is doubtful that a project of this size and duration could remain manageable and maintainable if the JigCell project had used the state of the art of scripting languages in 2001. The Java language supports rapid development, is easily portable to the platforms that the BioSPICE program supports, and efficient enough to work with large models.

The JigCell project also benefited from the build system that Allen developed in 2001 and the introduction of better software engineering methods in 2004. Having a completely automated build system saved immeasurable time during development and allowed the later introduction of automatic nightly builds. Furthermore, the build system included production of the web site and documentation, averting a major disaster when a shared, central server was lost in 2003 with no backups. The JigCell project, unlike others that this machine hosted, lost no data because the developers could easily reconstruct the data from their local copies.

Deciding to build many of the core components in-house

Another source of the early competitive advantage that the JigCell project enjoyed was the decision to develop independent implementations for reading and writing data. Vass developed a parser for SBML Level 2 Version 1 before the official adoption of the standard and concurrently with an unannounced project to develop a quasi-official parser written by the SBML authors. The JigCell Model Builder had support for SBML before many other modeling groups attempted to adopt the new standard. Moreover, the experience with implementing SBML gave the JigCell project considerable influence within the SBML community in exchange for describing problems with the proposed standard.

Although there was additional cost associated with producing this independent implementation, the JigCell project was better able to control its own destiny because it did not rely on external support for this core component.

The JigCell Comparator enjoyed a similar competitive advantage with its format for experimental data. Allen developed a lightweight format for representing experimental data [4] for the JigCell applications in 2001. The BioSPICE program later standardized this data format, but the JigCell Comparator retained its own separate parser. The JigCell project was immune to implementation changes in the BioSPICE Dashboard because of this independence.

Preventing feature creep from getting out-of-hand

Although the scope of the JigCell project expanded rapidly during the first year, there was relatively little feature creep since that time. Attempts to change the direction of the JigCell project generally did not affect the development of the core components, and these attempts died peacefully without consuming excessive resources. The JigCell project maintained a small team, and with the limited available programmer time, it was important that the scope and scale of the design remained under control.

A notable exception to this general area of success was the inclusion of support for the Open Agent Architecture and BioSPICE Dashboard. Although this support was a funding requirement, these efforts had cost for the JigCell project with little user benefit. Adding support for the BioSPICE Dashboard delayed the JigCell project by approximately six months. If the JigCell project had not supported the BioSPICE Dashboard, then it is likely that the developers would have completed stabilization of the critical bugs in the software much sooner. Reducing the time that the JigCell project spent in this chaotic period would have reduced the loss of users.

Realizing that the project needed a restart

Overhauling the software engineering practices of the JigCell project and targeting the buggiest portions of the source code for rewriting probably best advanced the project since the original release in 2002. By early 2004, it was clear that the JigCell project needed serious intervention to prevent failure. A large impetus to the project turnaround was the use of better software engineering methods. With better software engineering methods, rewriting portions of the code became valuable, as developers decreased both the number of lines of source code and the number of defects per line of source code. Almost ninety percent of the critical bugs resided in just three percent of the source code.

The adoption of a public bug-tracking system allowed the JigCell project developers to prioritize their work more effectively and prevented developers from losing bugs. Previously, there was little coordinated effort to prioritize development. Developers created their own schedules for performing work and delivered completed modules to Allen for integration. The only time constraints were the biannual BioSPICE releases that had fixed release dates. Developers often delivered modules that were buggy and missing critical features. As developers could delay working on a bug indefinitely, the JigCell project ignored many bug reports. Establishing a bug-tracking system stopped developers from losing bugs and focused attention on the large number of critical bugs.

7.2.2 What Went Wrong

Not having a clear prioritization of work

The JigCell project started foremost as a research effort. It is understandable that during the early period, there was no directed vision for how software development should proceed. Predicting the outcome and direction of research is difficult. However, after 2002, there was a greater attempt to push the JigCell modeling environment into production use. While internal users may tolerate “point, click, and crash” behavior in software because you can train internal users to avoid the buggy behavior, this approach does not scale to external users.

The JigCell project needed a vision of what modeling tools to build, which users to support, and when to deliver key functionality. Developers relied on their own initiative to plan and deliver features for most of the project. However, delivering a functioning suite of interconnected applications requires more planning and project management to coordinate development efforts. Having a clear prioritization would have helped avoid putting off critical bug fixes to work on relatively unimportant new features. Another issue was the early focus on producing demonstrations rather

than getting the software into the hands of real users. Biological modelers did not receive software that they could try on real models until after the JigCell project developers completed much of the implementation.

Failing to attract the right personnel

Finding the right personnel is a tough job for any software development project. However, the JigCell project seemed to have particular difficulty attracting developers with experience working on large projects. In part, this difficulty is inherent for an academic research laboratory, where many of the developers are students or recent graduates. Allen was the only developer on the JigCell project with experience working on large-scale software projects.

As the size of the JigCell project expanded past several hundred thousand lines of code, this problem grew particularly acute. New developers struggled to learn how their work fit into the overall software architecture and to learn how to plan and organize work on a large project over a long period. The length of the project, nearly five years, contributed to this problem as few developers had the 'institutional memory' of past decisions and experiences. The high rate of turnover of JigCell project developers during 2003 and 2004 was also a factor. The JigCell project could have eased this problem by producing better developer documentation and by investing more time teaching effective software engineering practices to new developers.

Not challenging the initial vision after gaining more experience

As Section 7.2 noted, the JigCell modeling environment today strikingly resembles the concept design of 2001. This was due more to stubbornness rather than prescience. For example, the JigCell Run Manager originally could not exist inside the Model Builder because the Run Manager used a separate model for each run. Later, the JigCell Run Manager changed to use a single model for all of the runs in a run file. However, the JigCell project did not revisit the decision to have separate applications for building a model and defining a collection of runs for that model.

The JigCell project developers also made several early decisions without sufficient information or long-range guidance. The use of the Open Agent Architecture, and later the BioSPICE Dashboard, was a funding requirement. Correcting this design decision, leading to substantial improvements in performance, reliability, and ease-of-use, took several years despite the early detection of problems.

Another early decision was the division of functionality among several applications. Applications took ownership of particular features primarily because of the personnel available in 2001 rather than any intrinsic reason. While the priorities and design shifted, the applications remained in their original arrangements. For example, the revised modeling process in Section 3.3 has five conceptual divisions of functionality, but the implementation of the JigCell modeling environment has four discrete applications. There are good arguments for reducing the number of discrete applications further, possibly to a single application like several other modeling environments. The push to develop an integrated project management component shows that there were concerns about application cohesiveness.

Letting the situation become dire

The JigCell project reached a low point early in 2004 when users refused to use the applications due to the number of data-loss bugs. The number of bugs grew at an unchecked rate for several reasons, many of which Section 7.2 discussed previously. However, a serious failure was not taking action sooner to regain control of the project. The high rate of bugs combined with the unresponsive of the JigCell project developers to fix those bugs, leading users to abandon JigCell entirely. This is not a small feat as many biological modelers are desperate to make use of any modeling tools that they can obtain.

There were many opportunities for the JigCell project to remediate its poor software engineering practices during 2003 and 2004. The JigCell project missed these opportunities because of insufficient effort to bring the poor software engineering practices to light. If the JigCell project developers had acknowledged that the software engineering practices that they used were not sustainable, then they would have wasted much less time before fixing the flaws in the software. After the JigCell project adopted better software engineering practices in the middle of 2004, the project stabilized and the developers finally started fixing more bugs than they created.

7.3 Software Evaluation Experiences

During 2004, the score for the JigCell applications with respect to the user requirements in Chapter 5 was acceptable. However, users were not satisfied with the quality of the JigCell applications during this period. It is reasonable to question why the user requirements did not identify these quality problems.

The requirements specification in Chapter 5 measures the availability of features and infers the quality of the project from those measurements. In 2004, the JigCell modeling environment had many features but poor quality, confounding the estimation of quality by the requirements specification. If the JigCell modeling environment had had a higher-quality basic implementation, then the score from the requirements specification would more accurately reflect the overall quality of the project. A more specific cause of this problem is that user requirements typically do not set detailed quality goals for a feature.

Users ask for a particular feature or capability in a software product, assuming that the software developers will act with due diligence to implement the feature. It is atypical for a user to specifically ask that the software product not crash while attempting to save their data. This approach is reasonable. The number of cases that can go wrong with a software product is unbounded, and users instead focus on the bounded set of features of the product.

The JigCell modeling environment had problems more fundamental than what the requirements specification measures. A requirements specification is only useful when a software product already meets a minimal bar for quality. This quality bar depends upon the level of detail of the requirements specification. The requirements specification in Chapter 5 focused on modeling activities that a modeler performs on a model. This requirements specification therefore assumes that the measured software supports fundamental operations on a model, such as loading and saving, which the JigCell Model Builder could not do reliably in 2004. The error was applying the requirements specification to the JigCell modeling environment in 2004 and expecting to get meaningful results.

Bibliography

[1] Christopher Abbey et al. Jikes home. Available at `http://http://jikes.sourceforge.net`, 2005.

[2] Aditya Agrawal, Gabor Karsai, and Akos Ledeczi. An end-to-end domain-driven software development framework. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 8–15. ACM Press, 2003.

[3] Bruce Alberts, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts, and James D. Watson. *Molecular Biology of the Cell*. Garland Science Publishing, New York, NY, USA, 1994.

[4] Nicholas A. Allen. BioSPICE Time Series Format Version 2. Available at `http://jigcell.biol.vt.edu/jigcell/docs/Comparator/timeseries/Time%20Series%20Format.txt`, 2004.

[5] Nicholas A. Allen. Jigcell comparator documentation. Available at `http://jigcell.biol.vt.edu/jigcell/docs/Comparator/index.html`, 2005.

[6] Nicholas A. Allen, Laurence Calzone, Katherine C. Chen, Andrea Ciliberto, Naren Ramakrishnan, Clifford A. Shaffer, Jill C. Sible, John J. Tyson, Marc T. Vass, Layne T. Watson, and Jason W. Zwolak. Modeling regulatory networks at Virginia Tech. *OMICS: A Journal of Integrative Biology*, 7(3):285–299, 2003.

[7] Nicholas A. Allen, Katherine C. Chen, Clifford A. Shaffer, John J. Tyson, and Layne T. Watson. Computer evaluation of network dynamics models with application to cell cycle control in budding yeast. *IEE Systems Biology*, accepted 21 September 2005.

[8] Nicholas A. Allen, Clifford A. Shaffer, et al. Jigcell project homepage. Available at `http://jigcell.biol.vt.edu`, 2005.

[9] Nicholas A. Allen, Clifford A. Shaffer, Naren Ramakrishnan, Marc T. Vass, and Layne T. Watson. Improving the development process for eukaryotic cell cycle models with a modeling support environment. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 79(12):674–688, 2003.

[10] Nicholas A. Allen, Clifford A. Shaffer, Marc T. Vass, Naren Ramakrishnan, and Layne T. Watson. Improving the development process for eukaryotic cell cycle models with a modeling support environment. In Stephen E. Chick, Paul J. Sanchez, David Ferrin, and Douglas J. Morrice, editors, *Proceedings of the 2003 Winter Simulation Conference*, pages 782–790. IEEE Computer Society Press, 2003.

[11] Nicholas A. Allen, Clifford A. Shaffer, and Layne T. Watson. Building modeling tools that support verification, validation, and testing for the domain expert. In Michael E. Kuhl, Natalie M. Steiger, Frank Brad Armstrong, and Jeffrey A. Joines, editors, *Proceedings of the 2005 Winter Simulation Conference*, pages 419–426. IEEE Computer Society Press, 2005.

[12] Rutherford Aris. Prolegomena to the rational analysis of systems of chemical reactions. *Archive for Rational Mechanics and Analysis*, 19:81–99, 1965.

[13] Vladimir I. Arnold and Roger Cooke. *Ordinary Differential Equations*. Springer-Verlag, New York, NY, USA, 1992.

[14] James D. Arthur and Richard E. Nance. Verification and validation without independence: a recipe for failure. In Jeffrey A. Joines, Russell R. Barton, Keebom Kang, and Paul A. Fishwick, editors, *Proceedings of the 2000 Winter Simulation Conference*, pages 859–865. Society for Computer Simulation International, 2000.

[15] Uri M. Ascher and Linda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[16] Ron Ausbrooks, Stephen Buswell, David Carlisle, Stephane Dalmas, Stan Devitt, Angel Diaz, Max Froumentin, Roger Hunter, Patrick Ion, Michael Kohlhase, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical markup language (MathML) version 2.0. Available at `http://www.w3.org/TR/MathML2/`, 2003.

[17] Osman Balci. Credibility assessment of simulation results. In James R. Wilson, James O. Henriksen, and Stephen D. Roberts, editors, *Proceedings of the 1986 Winter Simulation Conference*, pages 38–44. ACM Press, 1986.

[18] Osman Balci. Credibility assessment of simulation results: The state of the art. Technical Report TR-86-31, Department of Computer Science, Virginia Polytechnic Institute and State University, 1986.

[19] Osman Balci. Verification, validation, and accreditation. In Deborah J. Medeiros, Edward F. Watson, John S. Carson, and Mani S. Manivannan, editors, *Proceedings of the 1998 Winter Simulation Conference*, pages 41–48. IEEE Computer Society Press, 1998.

[20] Osman Balci and Richard E. Nance. The Simulation Model Development Environment: an overview. In James J. Swain, David Goldsman, Robert C. Crain, and James R. Wilson, editors, *Proceedings of the 1992 Winter Simulation Conference*, pages 726–736. ACM Press, 1992.

[21] Osman Balci, Richard E. Nance, James D. Arthur, and William F. Ormsby. Expanding our horizons in VV&A research and practice. In Enver Yucesan, Russel C. H. Cheng, Jane L. Snowdon, and John M. Charnes, editors, *Proceedings of the 2002 Winter Simulation Conference*, pages 653–663. IEEE Computer Society Press, 2002.

[22] Osman Balci, Richard E. Nance, E. Joseph Derrick, Ernest H. Page, and John L. Bishop. Model generation issues in a simulation support environment. In Osman Balci, Randall P. Sadowski, and Richard E. Nance, editors, *Proceedings of the 1990 Winter Simulation Conference*, pages 257–263. IEEE Computer Society Press, 1990.

[23] Osman Balci, William F. Ormsby, John T. Carr III, and Said D. Saadi. Planning for verification, validation, and accreditation of modeling and simulation applications. In Jeffrey A. Joines, Russell R. Barton, Keebom Kang, and Paul A. Fishwick, editors, *Proceedings of the 2000 Winter Simulation Conference*, pages 829–839. Society for Computer Simulation International, 2000.

[24] Osman Balci and Robert G. Sargent. A methodology for cost-risk analysis in the statistical validation of simulation models. *Communications of the ACM*, 24(4):190–197, 1981.

[25] Osman Balci, Cengiz Ulusarac, Poorav Shah, and Edward A. Fox. A library of reusable model components for visual simulation of the NCSTRL system. In Deborah J. Medeiros, Edward F. Watson, John S. Carson, and Mani S. Manivannan, editors, *Proceedings of the 1998 Winter Simulation Conference*, pages 1451–1460. IEEE Computer Society Press, 1998.

[26] David W. Balmer. Modelling styles and their support in the CASM environment. In Arne Thesen, Hank Grant, and W. David Kelton, editors, *Proceedings of the 1987 Winter Simulation Conference*, pages 478–485. ACM Press, 1987.

[27] David W. Balmer and Ray J. Paul. Integrated support environments for simulation modelling. In Osman Balci, Randall P. Sadowski, and Richard E. Nance, editors, *Proceedings of the 1990 Winter Simulation Conference*, pages 243–249. IEEE Computer Society Press, 1990.

[28] Alfredo Bellen and Marino Zennaro. *Numerical Methods for Delay Differential Equations*. Oxford University Press, London, United Kingdom, 2003.

[29] Jeremy M. Berg, John L. Tymoczko, and Lubert Stryer. *Biochemistry*. W. H. Freeman and Company, New York, NY, USA, 2002.

[30] BioSPICE. The BioSPICE development project. Available at `http://www.biospice.org`, 2005.

[31] Louis G. Birta and F. Nur Ozmizrak. A knowledge-based approach for the validation of simulation models: the foundation. *ACM Transactions on Modeling and Computer Simulation*, 6(1):76–98, 1996.

[32] John L. Bishop and Osman Balci. General purpose visual simulation system: a functional description. In Osman Balci, Randall P. Sadowski, and Richard E. Nance, editors, *Proceedings of the 1990 Winter Simulation Conference*, pages 504–512. IEEE Computer Society Press, 1990.

[33] Paul T. Boggs, Richard H. Byrd, Janet R. Donaldson, and Robert B. Schnabel. Algorithm 676 - ODRPACK: Software for weighted orthogonal distance regression. *ACM Transactions on Mathematical Software*, 15(4):348–364, 1989.

[34] Paul T. Boggs, Richard H. Byrd, Janet E. Rogers, and Robert B. Schnabel. *User's Refernce Guide for ODRPACK Version 2.01: Software for Weighted Orthogonal Distance Regression*. Center for Computing and Applied Mathematics, U.S. Department of Commerce, Gaithersburg, MD, USA, 1992.

[35] Paul T. Boggs, Richard H. Byrd, and Robert B. Schnabel. A stable and efficient algorithm for nonlinear orthogonal distance regression. *SIAM Journal on Scientific and Statistical Computing*, 8(6):1052–1078, 1987.

[36] William E. Boyce and Richard C. DiPrima. *Introduction to Differential Equations*. John Wiley & Sons, Inc., New York, NY, USA, 1970.

[37] George Edward Briggs and John Burdon Sanderson Haldane. A note on the kinetics of enzyme action. *Biochemistry Journal*, 19:339–339, 1925.

[38] George D. Byrne and Alan C. Hindmarsh. Stiff ODE solvers: a review of current and coming attractions. *Journal of Computational Physics*, 70(1):1–62, 1987.

[39] Ray J. Carroll, David Ruppert, and Lenny A. Stefanski. *Measurement Error in Nonlinear Models*. Chapman and Hall/CRC, New York, NY, USA, 1995.

[40] Don Caughlin. An integrated approach to verification, validation, and accreditation of models and simulations. In Jeffrey A. Joines, Russell R. Barton, Keebom Kang, and Paul A. Fishwick, editors, *Proceedings of the 2000 Winter Simulation Conference*, pages 872–881. Society for Computer Simulation International, 2000.

[41] Katherine C. Chen, Laurence Calzone, Attila Csikasz-Nagy, Frederick R. Cross, Bela Novak, and John J. Tyson. Integrative analysis of cell cycle control in budding yeast. *Molecular Biology of the Cell*, 15(8):3841–3862, 2004.

[42] Katherine C. Chen, Attila Csikasz-Nagy, Bela Gyorffy, John Val, Bela Novak, and John J. Tyson. Kinetic analysis of a molecular model of the budding yeast cell cycle. *Molecular Biology of the Cell*, 11(1):369–391, 2000.

[43] Candace L. Conwell, Rosemary Enright, and Marcia A. Stutzman. Capability maturity models support of modeling and simulation verification, validation, and accreditation. In Jeffrey A. Joines, Russell R. Barton, Keebom Kang, and Paul A. Fishwick, editors, *Proceedings of the 2000 Winter Simulation Conference*, pages 819–828. Society for Computer Simulation International, 2000.

[44] Athel Cornish-Bowden and Jan-Hendrik S. Hofmeyr. The role of stoichiometric analysis in studies of metabolism: An example. *Journal of Theoretical Biology*, 216(2):179–191, 2002.

[45] Autumn A. Cuellar, Catherine M. Lloyd, Poul F. Nielsen, David P. Bullivant, David P. Nickerson, and Peter J. Hunter. An overview of CellML 1.1, a biological model description language. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 79(12):740–747, 2003.

[46] Fatima Cvrckova and Kim Nasmyth. Yeast G1 cyclins CLN1 and CLN2 and a GAP-like protein have a role in bud formation. *EMBO Journal*, 12:5277–5286, 1993.

[47] Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[48] Dursun Delen, Perakath C. Benjamin, and Madhav Erraguntla. Integrated modeling and analysis generator environment (IMAGE): a decision support tool. In Deborah J. Medeiros, Edward F. Watson, John S. Carson, and Mani S. Manivannan, editors, *Proceedings of the 1998 Winter Simulation Conference*, pages 1401–1408. IEEE Computer Society Press, 1998.

[49] Rodney D. Driver. *Ordinary and Delay Differential Equations*. Springer-Verlag, New York, NY, USA, 1977.

[50] Mireille Ducasse and Anna-Maria Emde. A review of automated debugging systems: Knowledge, strategies and techniques. In *Proceedings of the 10th International Conference on Software Engineering*, pages 162–171. IEEE Computer Society Press, 1988.

[51] Bard Ermentrout. *Simulating, Analyzing, and Animating Dynamical Systems: a Guide to XPPAUT for Researchers and Students*. SIAM Press, Philadelphia, PA, USA, 2002.

[52] Bard Ermentrout. XPP/XPPAUT homepage. Available at `http://www.math.pitt.edu/~bard/xpp/xpp.html`, 2004.

[53] Martin Feinberg. Necessary and sufficient conditions for detailed balancing in mass action systems of arbitrary complexity. *Chemical Engineering Science*, 44:1819–1827, 1989.

[54] Andrew Finney, Michael Hucka, John C. Doyle, and Hiroaki Kitano. Systems Biology Markup Language (SBML) Level 2: Structures and facilities for basic model definitions. Available at `http://www.sbml.org/specifications/sbml-level-2/version-1/sbml-level-2.pdf`, 2003.

[55] Martin R. Frank and Pedro Szekely. Adaptive forms: an interaction paradigm for entering structured data. In *Proceedings of the 3rd International Conference on Intelligent User Interfaces*, pages 153–160. ACM Press, 1998.

[56] Akira Funahashi, Naoki Tanimura, Mineo Morohashi, and Hiroaki Kitano. CellDesigner: a process diagram editor for gene-regulatory and biochemical networks. *BioSilico*, 1(5):159–162, 2003.

[57] C. William Gear. Differential algebraic equations, indices, and integral algebraic equations. *SIAM Journal on Numerical Analysis*, 27(6):1527–1534, 1990.

[58] Michael A. Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *Journal of Physical Chemistry A*, 104(9):1876–1889, 2000.

[59] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[60] Daniel T. Gillespie. A rigorous derivation of the chemical master equation. *Physica A: Statistical Mechanics and its Applications*, 188(1–3):404–425, 1992.

[61] Daniel T. Gillespie. The chemical Langevin equation. *The Journal of Chemical Physics*, 113(1):297–306, 2000.

[62] Amit Goel, Constantinos Phanouriou, Frederick A. Kamke, Calvin J. Ribbens, Clifford A. Shaffer, and Layne T. Watson. WBCSim: A prototype problem solving environment for wood-based composites simulations. *Engineering with Computers*, 15(2):198–210, 1999.

[63] Albert Goldbeter and Douglas E. Koshland Jr. An amplified sensitivity arising from covalent modification in biological systems. *Proceedings of the National Academy of Science*, 78(11):6840–6844, 1981.

[64] Gordon G. Hammes. *Enzyme Catalysis and Regulation*. Academic Press, New York, NY, USA, 1982.

[65] Jian He, Layne T. Watson, Naren Ramakrishnan, Clifford A. Shaffer, Alex Verstak, Jing Jiang, Kyung Bae, and William H. Tranter. Dynamic data structures for a direct search algorithm. *Computational Optimization and Applications*, 23:5–25, 2002.

[66] James O. Henriksen. The integrated simulation environment. *Operations Research*, 31(6):1053–1073, 1983.

[67] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*, 31(3):363–396, 2005.

[68] Jan-Hendrik S. Hofmeyr and Athel Cornish-Bowden. The reversible Hill equation: How to incorporate cooperative enzymes into metabolic models. *Computer Applications in the Biosciences*, 13(4):377–385, 1997.

[69] Jan-Hendrik S. Hofmeyr, Henrik Kacser, and Kirsten J. van der Merwe. Metabolic control analysis of moiety-conserved cycles. *European Journal of Biochemistry*, 155(3):631–641, 1986.

[70] Haowei Hsieh and Frank M. Shipman. Manipulating structured information in a visual workspace. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology*, pages 217–226. ACM Press, 2002.

[71] Michael Hucka, Andrew Finney, Benjamin J. Bornstein, Sarah M. Keating, Bruce E. Shapiro, Joanne Matthews, Ben L. Kovitz, Maria J. Schilstra, Akira Funahashi, John C. Doyle, and Hiroaki Kitano. Evolving a lingua franca and associated software infrastructure for computational systems biology: the Systems Biology Markup Language (SBML) project. *IEE Systems Biology*, 1(1):41–53, 2004.

[72] Michael Hucka, Andrew Finney, Herbert M. Sauro, Hamid Bolouri, John C. Doyle, Hiroaki Kitano, Adam P. Arkin, Benjamin J. Bornstein, Dennis Bray, Athel Cornish-Bowden, Autumn A. Cuellar, Serge Dronov, Ernst Dieter Gilles, Martin Ginkel, Victoria Gor, Igor I. Goryanin, Warren J. Hedley, T. Charles Hodgman, Jan-Hendrik S. Hofmeyr, Peter J. Hunter, Nick S. Juty, Jay L. Kasberger, Andreas Kremling, Ursula Kummer, Nicolas Le Novere, Leslie M. Loew, Daniel Lucio, Pedro Mendes, Eric Minch, Eric D. Mjolsness, Yoichi Nakayama, Melanie R. Nelson, Poul F. Nielsen, Takeshi Sakurada, James C. Schaff, Bruce E. Shapiro, Tom S. Shimizu, Huge D. Spence, Jorg Stelling, Kouichi Takahashi, Masaru Tomita, John Wagner, and Jian Wang. The Systems Biology Markup Language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[73] Nobuyuki Ikeda, Shinzo Watanabe, Masatsushi Fukushima, and Hiroshi Kunita. *Ito's Stochastic Calculus and Probability Theory*. Springer-Verlag Tokyo, Tokyo, Japan, 1996.

[74] Matthew D. Jacobson, Samantha Gray, Maria Yuste-Rojas, and Frederick R. Cross. Testing cyclin specificity in the exit from mitosis. *Molecular and Cellular Biology*, 20(13):4483–4493, 2000.

[75] Donald R. Jones, Cary D. Perttunen, and Bruce E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, 1993.

[76] Ioannis Karatzas and Steven E. Shreve. *Brownian Motion and Stochastic Calculus*. Springer-Verlag, New York, NY, USA, 1997.

[77] Holger M. Kienle. Using smgn for rapid protoptyping of small domain-specific languages. *SIGPLAN Notices*, 36(9):64–73, 2001.

[78] Hiroaki Kitano. A graphical notation for biological networks. *BioSilico*, 1(5):169–176, 2003.

[79] Peter E. Kloeden and Eckhard Platen. *Numerical Solution of Stochastic Differential Equations*. Springer-Verlag, New York, NY, USA, 2000.

[80] Kurt W. Kohn. Molecular interaction map of the mammalian cell cycle control and DNA repair system. *Molecular Biology of the Cell*, 10(8):2703–2734, 1999.

[81] David Krahl. Debugging simulation models. In Michael E. Kuhl, Natalie M. Steiger, Frank Brad Armstrong, and Jeffrey A. Joines, editors, *Proceedings of the 2005 Winter Simulation Conference*, pages 62–68. IEEE Computer Society Press, 2005.

[82] Sari Kujala and Marjo Kauppinen. Identifying and selecting users for user-centered design. In *Proceedings of the Third Nordic Conference on Human-Computer Interaction*, pages 297–303. ACM Press, 2004.

[83] Don S. Lemons and Anthony Gythiel. Paul Langevin's 1908 paper "On the Theory of Brownian Motion". *American Journal of Physics*, 65(11):1079–1081, 1997.

[84] Joachim J. Li. Once, and only once. *Current Biology*, 5:472–475, 1995.

[85] Catherine M. Lloyd, Matt D. B. Halstead, and Poul F. Nielsen. CellML: its future, present and past. *Progress in Biophysics and Molecular Biology*, 85(2–3):433–450, 2004.

[86] Leslie M. Loew and James C. Schaff. The Virtual Cell: a software environment for computational cell biology. *Trends in Biotechnology*, 19:401–406, 2001.

[87] Gabor Marlovits, Christopher J. Tyson, Bela Novak, and John J. Tyson. Modeling M-phase control in Xenopus Oocyte extracts: the surveillance mechanism for unreplicated DNA. *Biophysical Chemistry*, 72:169–184, 1998.

[88] Jerrold E. Marsden and Anthony J. Tromba. *Vector Calculus*. W. H. Freeman and Company, New York, NY, USA, 1996.

[89] David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The Open Agent Architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1):91–128, 1999.

[90] Stephen C. Mathewson. Simulation support environments. In *Computer Modelling for Discrete Event Simulation*, pages 57–100. John Wiley & Sons, Inc., 1989.

[91] Sven Erik Mattsson and Gustaf Soderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal of Scientific Computing*, 14(3):677–692, 1993.

[92] Donald A. McQuarrie. Stochastic approach to chemical kinetics. *Journal of Applied Probability*, 4(2):413–478, 1967.

[93] Michael D. Mendenhall and Amy E. Hodge. Regulation of Cdc28 cyclin dependent protein kinase activity during the cell cycle of the yeast Saccharomyces cerevisiae. *Microbiological Molecular Biology Review*, 62:1191–1243, 1998.

[94] Pedro Mendes. Biochemistry by numbers: Simulation of biochemical pathways with Gepasi 3. *Trends in Biochemical Sciences*, 22:361–363, 1997.

[95] Pedro Mendes and Douglas B. Kell. MEG (model extender for Gepasi): a program for the modelling of complex, heterogeneous, cellular systems. *Bioinformatics*, 17(3):288–289, 2001.

[96] Leonor Michaelis and Maud Menten. Die kinetik der invertinwirkung. *Biochemische Zeitschrift*, 49:333–369, 1913.

[97] Sun Microsystems. Welcome to NetBeans. Available at `http://www.netbeans.org`, 2005.

[98] Keith William Morton and David F. Mayers. *Numerical Solution of Partial Differential Equations*. Cambridge University Press, Cambridge, United Kingdom, 1994.

[99] Andrew Murray and Tim Hunt. *The Cell Cycle: an Introduction*. W.H. Freeman and Company, New York, NY, USA, 1993.

[100] Richard E. Nance. Model development revisited. In Sallie Sheppard, Udo W. Pooch, and C. Dennis Pegden, editors, *Proceedings of the 1984 Winter Simulation Conference*, pages 74–80. IEEE Computer Society Press, 1984.

[101] Richard E. Nance. The conical methodology: A framework for simulation model development. Technical Report TR-87-08, Department of Computer Science, Virginia Polytechnic Institute and State University, 1987.

[102] Richard E. Nance. The conical methodology and the evolution of simulation model development. *Annals of Operations Research*, 53:1–45, 1994.

[103] Richard E. Nance and James D. Arthur. The methodology roles in the realization of a model development environment. In Micahel A. Abrams, Peter L. Haigh, and John C. Comfort, editors, *Proceedings of the 1988 Winter Simulation Conference*, pages 220–225. Society for Computer Simulation International, 1988.

[104] Richard E. Nance, C. Michael Overstreet, and Ernest H. Page. Redundancy in model specifications for discrete event simulation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):254–281, 1999.

[105] Kim Nasmyth. At the heart of the budding yeast cell cycle. *Trends in Genetics*, 12:405–412, 1996.

[106] Bela Novak, Zsuzsa Pataki, Andrea Ciliberto, and John J. Tyson. Mathematical model of the cell division cycle of fission yeast. *Chaos*, 11(1):277–286, 2001.

[107] Paul Nurse. Universal control mechanism regulating onset of M-phase. *Nature*, 344:503–508, 1990.

[108] Paul Nurse. A long twentieth century of the cell cycle and beyond. *Cell*, 100(1):71–78, 2000.

[109] C. Michael Overstreet. *Model Specification and Analysis for Discrete Event Simulation*. PhD thesis, Department of Computer Science, Virginia Polytechnic Institute and State University, 1982.

[110] C. Michael Overstreet. Model testing: is it only a special case of software testing? In Enver Yucesan, Russel C. H. Cheng, Jane L. Snowdon, and John M. Charnes, editors, *Proceedings of the 2002 Winter Simulation Conference*, pages 641–647. IEEE Computer Society Press, 2002.

[111] Natarajan Padmanaban, Perakath C. Benjamin, and Richard J. Mayer. Integrating multiple descriptions in simulation model design: a knowledge based approach. In Christos Alexopoulos, Keebom Kang, William R. Lilegdon, and David Goldsman, editors, *Proceedings of the 1995 Winter Simulation Conference*, pages 714–719. ACM Press, 1995.

[112] Ernest H. Page. *Simulation Modeling Methodology: Principles and Etiology of Decision Support*. PhD thesis, Department of Computer Science, Virginia Polytechnic Institute and State University, 1994.

[113] Thomas D. Panning, Layne T. Watson, Nicholas A. Allen, Clifford A. Shaffer, and John J. Tyson. Deterministic global parameter estimation for a budding yeast model. In *Proceedings of the 2005 High Performance Computing Symposium*, submitted October 2005.

[114] Earl D. Rainville. *Elementary Differential Equations*. Macmillan Company, New York, NY, USA, 1952.

[115] Naren Ramakrishnan, Layne T. Watson, Dennis G. Kafura, Cal J. Ribbens, and Clifford A. Shaffer. Programming environments for multidisciplinary grid communities. *Concurrency and Computation: Practice and Experience*, 14:1241–1273, 2002.

[116] Muruhan Rathinam, Linda R. Petzold, Yang Cao, and Daniel T. Gillespie. Stiffness in stochastic chemically reacting systems: the implicit tau-leaping method. *Journal of Chemical Physics*, 119(24):12784–12794, 2003.

[117] John R. Rice and Ronald F. Boisvert. From scientific software libraries to problem-solving environments. *IEEE Computational Science and Engineering*, 3(3):44–53, 1996.

[118] Javier Nicolas Sanchez and Pat Langley. An interactive environment for scientific model construction. In *Proceedings of the International Conference on Knowledge Capture*, pages 138–145. ACM Press, 2003.

[119] Pavi Sandhu. *The MathML Handbook*. Delmar Thomson Learning, Clifton Park, NY, USA, 2002.

[120] Robert G. Sargent. The use of graphical models in model validation. In James R. Wilson, James O. Henriksen, and Stephen D. Roberts, editors, *Proceedings of the 1986 Winter Simulation Conference*, pages 237–241. ACM Press, 1986.

[121] Robert G. Sargent. Validation and verification of simulation models. In Ricki G. Ingalls, Manuel D. Rossetti, Jeffrey S. Smith, and Brett A. Peters, editors, *Proceedings of the 2004 Winter Simulation Conference*, pages 17–28. IEEE Computer Society Press, 2004.

[122] Herbert M. Sauro. Jarnac: A system for interactive metabolic analysis. In *Animating the Cellular Map: Proceedings of the 9th International Meeting on BioThermoKinetics*, pages 221–228. Stellenbosch University Press, 2000.

[123] Herbert M. Sauro, Michael Hucka, Andrew Finney, Cameron Wellock, Hamid Bolouri, John C. Doyle, and Hiroaki Kitano. Next generation simulation tools: the Systems Biology Workbench and BioSPICE integration. *OMICS: A Journal of Integrative Biology*, 7(4):355–372, 2003.

[124] Herbert M. Sauro and Brian Ingalls. Conservation analysis in biochemical networks: Computational issues for software writers. *Biophysical Chemistry*, 109(1):1–15, 2004.

[125] James C. Schaff, Charles C. Fink, Boris Slepchenko, John H. Carson, and Leslie M. Loew. A general computational framework for modeling cellular structure and function. *Biophysical Journal*, 73(3):1135–1146, 1997.

[126] Patricia Schank and Lawrence Hamel. Hiding UML and promoting data examples in NEMo. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 574–577. ACM Press, 2004.

[127] Lawrence F. Shampine and Marilyn K. Gordon. *Computer Solution of Ordinary Differential Equations, The Initial Value Problem*. W. H. Freeman and Company, New York, NY, USA, 1975.

[128] Jiang Shu, Layne T. Watson, Naren Ramakrishnan, Frederick A. Kamke, and Balazs G. Zombori. An experiment management component for the WBCSim problem solving environment. *Advances in Engineering Software*, 35(2):115–123, 2004.

[129] Charles A. Standridge and Steven A. Walker. An introduction to simulation support software. In Jerry Banks and Bruce Schmeiser, editors, *Proceedings of the 1983 Winter Simulation Conference*, pages 381–384. IEEE Computer Society Press, 1983.

[130] Kouichi Takahashi, Masahiro Ishikawa, Yasuhiro Sadamoto, Hiroykui Sasamoto, Seiji Ohta, Akira Shiozawa, Fumihiko Miyoshi, Yasuhiro Naito, Yoichi Nakayama, and Masaru Tomita. E-CELL2: Multi-platform E-CELL simulation system. *Bioinformatics*, 19(13):1727–1729, 2003.

[131] Masaru Tomita. Whole-cell simulation: a grand challenge of the 21st century. *Trends Biotechnology*, 19(6):205–210, 2001.

[132] John J. Tyson and Bela Novak. Regulation of the eukaryotic cell cycle: Molecular antagonism, hysteresis, and irreversible transitions. *Journal of Theoretical Biology*, pages 249–263, 2001.

[133] Edwin C. Valentin and Alexander Verbraeck. Guidelines for designing simulation building blocks. In Enver Yucesan, Russel C. H. Cheng, Jane L. Snowdon, and John M. Charnes, editors, *Proceedings of the 2002 Winter Simulation Conference*, pages 563–571. IEEE Computer Society Press, 2002.

[134] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[135] Marc T. Vass, Nicholas A. Allen, Clifford A. Shaffer, Naren Ramakrishnan, Layne T. Watson, and John J. Tyson. The JigCell Model Builder and Run Manager. *Bioinformatics*, 20(18):3680–3681, 2004.

[136] Marc T. Vass and Pete Schoenhoff. Error detection support in a cellular modeling end-user programming environment. In *IEEE 2002 Symposia on Human Centric Languages and Environments*, pages 104–106, 2002.

[137] Layne T. Watson, Vinod K. Lohani, David F. Kibler, Randy L. Dymond, Naren Ramakrishnan, and Clifford A. Shaffer. Integrated computing environments for watershed management. *Journal of Computational Civil Engineering*, 16:259–268, 2002.

[138] Paul Wonnacott and David Bruce. The APOSTLE simulation language: Granularity control and performance data. In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, pages 114–123. IEEE Computer Society Press, 1996.

[139] Sammuel A. Wright and Kenneth W. Bauer. Covalidation of dissimilarly structured models. In Sigrun Andradottir, Kevin J. Healy, David H. Withers, and Barry L. Nelson, editors, *Proceedings of the 1997 Winter Simulation Conference*, pages 311–318. ACM Press, 1997.

[140] Francois Yergeau, Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0. Available at `http://www.w3.org/TR/2004/REC-xml11-20040204/`, 2004.

[141] Wolfgang Zachariae and Kim Nasmyth. Whose end is destruction: Cell division and the anaphase-promoting complex. *Genes and Development*, 13:2039–2058, 1999.

Vita

Nicholas A. Allen was born in Columbia, MD, USA. He first arrived at the Department of Computer Science of Virginia Tech in 1997. He soon enrolled in the Departments of English (Literature), Mathematics (Applied Discrete Mathematics), and Physics. After completing the B.S. degree in Mathematics (magna cum laude) with a minor in English Literature, he completed the B.S. degree in Computer Science (magna cum laude) in 1999. Starting in 1999, he began working as a research assistant at the university, variously employed by the Departments of Biology, Computer Science, and Mathematics. He received M.S. degrees in both Mathematics and Computer Science in 2001. Between completing his Ph.D. research and writing this dissertation, he managed software development in the laboratory of Dr. John Tyson in the Department of Biology. He completed this dissertation in November 2005.