Framework for Formal Automated Analysis of Simulation Experiments

by

Kyle Doud

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 6, 2017

Keywords: Hypothesis Testing, Model Checking, Model Discovery

Abstract

Simulation experiments are becoming a widely feasible medium for scientific discovery due to the amount of traceability and reproducibility which computing systems are able to provide. These characteristics help to ensure the validity of an experimental process, which in turn gives credibility to the propositions supported therein. Simulations can be as complex as the real-world mechanisms they mimic, so the process of experimentation on a simulation can have an explosive search space. We attempt to minimize some of this complexity by reducing data concerns to a more abstract verification model. Additional experimental complexities are reduced by the inclusion of the goal-hypothesis-experiment framework, which includes a domain-specific language for experiment management. By making use of this framework, a scientist needs only to specify the goal of the experiment and give evidence and hypotheses to be tested against a simulation model. By utilizing formal methods of model verification we may automate the process of testing hypotheses against evidence, and facilitate the selection of new hypotheses to maximize the information gained while using the minimum processing requirements. In this thesis, a probabilistic model checker is used as the formal method of verification, and evidence and hypotheses are specified as a temporal logic specifications.

Acknowledgments

I would like to thank my adviser, Dr. Levent Yilmaz for providing guidance, patience, support, and enthusiasm throughout the course of this study. I would also like to express my gratitude to my advisory committee members Dr. Saad Biaz and Dr. David Umphress for their participation in my committee and for the guidance and education they have provided during my graduate studies. Additional thanks must go to Sritika Chakladar for being a valued friend and guide through my time in graduate school, and Joshua Newton for believing in me and providing constant support. Finally, I would like to thank my entire family for providing stability and love throughout my time in Auburn.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As software simulation practices are becoming more prevalent in various scientific disciplines [1, 2], it is becoming apparent that there is room for exploitation of the scientific process by leveraging computational strengths such as formal model checking in the use of these models for experimentation. Given that scientific models exist from a diverse array of fields, ranging from groundwater analysis [3] to nuclear fusion, socioeconomic forecast, and astrophysics [4], it is clear that a process to aid in the simplification of experimentation must be abstract enough to be applicable to such a wide variety of models. The presented solution aims to take some of the guesswork out of experimentation in a systematic way that reduces time spent on designing experiments. This goal is accomplished by infusing simulation models with code generated by an aspect-oriented programming extension (AspectJ) as a method of instrumentation. This allows a scientist to record quantitative observations without impacting the course of program execution, and use the data recorded to form a verification model for automated hypothesis testing.

Before further consideration of the possibilities of testing hypotheses automatically by aid of instrumentation, it is important to first examine the role that in-silico experimentation may play in science. Honavar [5] suggests that, "computation is now serving as language for science, a role not unlike that played by mathematics over the past many centuries, and a powerful formal framework and exploratory apparatus for the conduct of science." His claim is supported by the observation that computation, mathematics, and science are often used ubiquitously to provide a structured way of answering questions. As Honavar's statement alludes to, the appeal of using computation as a language for science is great because algorithms and data structures can perform the same role in science that mathematics played in the past, but at an accelerated rate. For the purposes of this chapter, computation will play the specific scientific role of hypothesis testing. In the established scientific method, hypotheses are used as a tool to test assumptions that seem to explain observations. Once a hypothesis is rigorously tested under varying conditions, a hypothesis can become a widely accepted theory, and can be used as evidence for support of future hypotheses. In regard to this, the point is raised that, "hypothesis tests become rules of 'inductive behavior' (Neyman and Pearson 1933) where an evidential assessment of the tested hypotheses is not of intrinsic interest, but just in so far as it helps us taking the right action" [6]. The basic view that is presented here is that, due to the impossibility that all possible evidence is gathered, a hypothesis test cannot guarantee the hypothesiss accuracy or inaccuracy, but it can be used to lead the scientist to the next question. It is for this reason that a computational system for hypothesis testing is enticing. To elaborate, a computational system with persistent memory naturally excels at traceability and reproducibility. By leveraging these attributes, the complex problem of analyzing the implications of a failed theory or hypothesis can be reduced by automatically rolling back those hypotheses that relied on inaccuracies as evidence.

In addition to the intractable problems with inductive reasoning, in-silico experimentation often suffers from incomplete or inaccurate models that should be representative of real-world mechanisms [7]. This statement may seem obvious, since it is commonly acknowledged that the task of experimentation is to understand reality's mechanisms. In other words, how can a model that only represents those aspects that are already well established from past experiments answer questions that have not been answered by a real-world example? As Savory [8] points out in reference to the formulation of simulation models, "missing an essential element may invalidate the representation provided by the model or make it useless for the intended application." This is an important observation, which exposes a call for constant comparison with real-world results. In the proposed system, this problem is at least partially mitigated by introducing real-world observations to the model as evidence, and guiding the user to refinements of the model that would make it more mimetic of real-world mechanisms.

However, it is clearly impossible to iteratively refine a model ad-infinitum. Tragically, this is the only way to make some models complete due to the randomness that exists in nature. Moskovitz [9] posits some deep philosophic questions relating to this assertion. He argues that the universe is not random, but chaotic. The difference being that, "natural events are chaotic, fully determined but unpredictable," whereas simulated events are undetermined, but predictable. If it were possible to create models that assume complexity rather than randomness, the implication would be that a model could reproduce what is, rather than what may be. However, such methods of modeling do not exist at present, so, for this study, statistical model checking is applied to test hypotheses since alternative methods are unavailable for the foreseeable future.

We propose to address some of the above challenges by providing a framework for hypothesis testing that is simulation platform independent[1] by following principles of Model-Driven Software Development. The proposed solution provides a domain specific language for hypothesis specification and automated model checking to evaluate these hypotheses using a statistical model checker. By incorporating the results of model checking into learning networks, better experiments can be developed at a faster rate to increase knowledge gain.

Organization of the thesis. The next chapter discusses the process of model-driven software development as well as a summary of the framework in which the hypothesis testing unit is enveloped in. Additionally, we explore the current work on experimentation frameworks, and introduce a foundational background for the work. Chapter 3 presents the model-driven methodology that was followed for the development of this study. Chapter 4 explains the implementation details of the framework. Chapter 5 presents a case study that shows the framework in action. The framework for **F**ormal Automated **A**nalysis of **S**imulation **E**xperiments (FASE) is evaluated and validated through exploration of a case

[1]The data recording mechanisms of the system require a language based in Java, but the domain-specific language is platform independent, as it can have alternative reference implementations.

study in chapter 6. Finally, in chapter 7, we conclude the work and discuss potential avenues of future research.

Chapter 2

Background

In this chapter we present the concepts and terminology used in the later chapters. In 2.1, Model-Driven Software Development is described as a means for developing models with re-use and minimalistic coding idioms. Section 2.2 presents the Goal-Hypothesis-Experiment framework, a process that increases the efficiency of digital experimentation. In sections 2.3 and 2.4, instrumentation and formal verification are explored as a means to improve upon the framework of the previous section. In section 2.5 the concept of probabilistic model checking is introduced. Section 2.6 explains the well-defined temporal properties, known as property specification patterns. Finally, section 2.7 introduces the concept of explanatory coherence networks, which are used later in the text to model hypothesis relationships.

2.1 Model-Driven Software Development

Model-Driven Software Development (MDSD) [10, 11, 12] is a strategy of software development that focuses on building software that is correct-by-construction as opposed to construct-by-correction. MDSD provides methods for the formulation, construction, and management of models. It presents a philosophy of re-use and generalization to promote the efficient use of software constructs in complex systems. The goals of MDSD are to increase development speed by automating more of the coding tasks, improving software quality, reducing redundancy and increasing maintainability in systems, managing complexity through abstraction, and improving interoperability and portability of complex systems. The basic process of MDSD is presented in figure 2.1.

Voelter, Salzmann, and Kircher [13] explain that through the practice of MDSD, models are treated as first class development artifacts, instead of just documentation figures. This practice provides an ability to specify various abstract aspects of a system in a modeling language, which can later be converted to source code through a series of transformations. The models used, being far more abstract than implementation code, are specific to the domain that the model is to be used for. The languages to describe these models are called domain-specific languages (DSL) [14]. A DSL must have a meta-model, a concrete syntax, and semantics. The meta-model defines the parts of the language and how they can be combined. A concrete syntax is the notation used to specify models. There is a one-to-many relationship between a meta-model and concrete syntax, since keyword choices may differ, and the medium (text or graphics) may also vary. Finally, the semantics of the DSL ensure that the model's meaning is well-defined for the purpose of facilitating transformations [13].

In *Model-Driven Software Development* [15], some common software development pitfalls are shown to be avoidable by use of MDSD principles:

> "The idea of modeling is not exactly new, and is used mostly for sophisticated development processes to document a softwares inner structure. Developers then try to counteract the inevitable consistency problems with time-consuming reviews. In practice, these reviews and also the models are among the first victims when time presses from a pragmatic point of view, even rightly so. Another approach is round-trip or reverse engineering, which most UML tools offer, which is merely source code visualization in UML syntax: that is, the abstraction level of these models is the same as for the source code itself. Visually it may be clearer, but the essential problem remains the same. Model-Driven Software Development offers a significantly more effective approach: Models are abstract and formal at the same time. Abstractness does not stand for vagueness here, but for compactness and a reduction to the essence. MDSD models have the exact meaning of program code in the sense that the bulk of the final implementation, not just class and method skeletons, can be generated from them. In this case, models are no longer only documentation, but parts of the software, constituting

a decisive factor in increasing both the speed and quality of software development. We emphasize model-driven as opposed to model-based to verbally highlight this distinction."

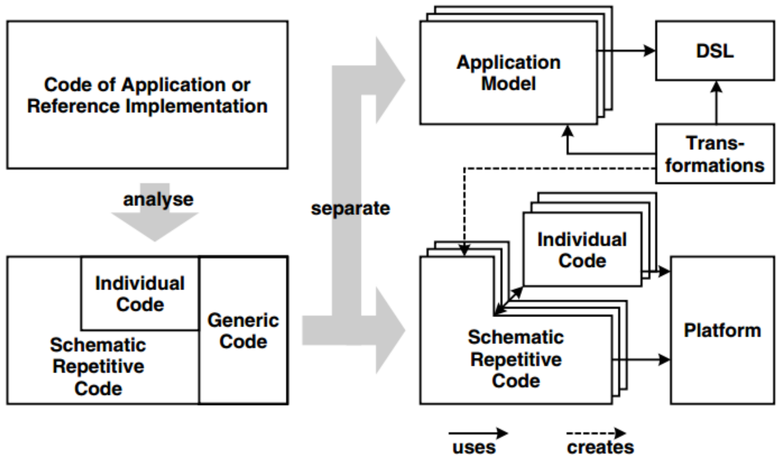


Figure 2.1: Model-Driven Software Development Process [15]

The central idea of this process is to identify the segments of code that are repetitive or generic and create a concrete implementation of these segments on-the-fly by utilizing code generation templates. The practical upshot of all this is that an engineer needs only to code one template and have a wide variety of concrete implementations by customizing the template with a set of arguments.

2.2 Goal-Hypothesis-Experiment Framework

In this section, we delineate the overarching infrastructure of FASE: the Goal-Hypothesis-Experiment (GHE) framework [16, 17]. The goal of FASE, combined with the GHE framework is to extend the tools and efficiency of digital science using simulation models. The GHE framework adheres to the principles of MDSD listed in the preceding section in order to facilitate the range and dynamics of applicable models. It provides a Domain Specific Language (DSL) for experiment specification, and describes methodology for iteratively exploring and refining domain ontologies. The GHE framework gives domain experts the tools they need to perform experiments, test hypotheses, observe phenomena, and refine models without having need for advanced coding and code analysis expertise.

As the name implies, the GHE framework structures the knowledge discovery process into three levels: conceptual, operational, and tactical levels (goals, hypotheses, and experiments respecively) [17]. It is proposed that, after the goal phase, knowledge can be gained most efficiently by iteratively exploring the hypothesis and experiment spaces of a domain, while performing optimizations in between each space change. In order to carry out the execution of these spaces, the DSL needs to be backed by a reference implementation to support hypothesis testing. The requirements for this endeavor include instrumentation (to construct a verification model) and model checking (to validate the model), each of which are discussed in the following sections.

2.3 Instrumentation

This section presents the concept of instrumentation on simulation experiments and how it is supported by Aspect-Oriented Programming (AOP) [18]. Here, instrumentation refers to measuring instruments used to indicate, measure and record quantities observed in the course of an experiment. Though it is obvious that physical apparatus are not needed to make measurements in a digital system, code to extract values from variables at certain instances during program execution is a means of mimicking the same process of data collection that is used during in-vivo experimentation. To facilitate instrumentation for experiments, AspectJ [19] is presented as a means of recording variable values during simulation execution by AOP methods.

Aspect-Oriented Programming aims to address *crosscutting concerns* in software. A crosscutting concern is a structure in which "methods related to those concerns intersect, but which cannot be neatly separated from each other" [18]. An example of this is data logging code, where data is logged in-line with business logic. These concerns are addressed by separating *components* and *aspects*, then weaving them together in a manner that untangles the problem. Aspects are defined as "properties that affect the performance or semantics of the components in systemic ways", such as, "memory access patterns and synchronization of concurrent objects" [20]. Stated formally, the goal of AOP is, "to support the programmer in cleanly separating components and aspects from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system" [20].

AspectJ is an AOP language that works as an extension of Java. It serves as a tool to facilitate instrumentation by intercepting variable values at well-defined points of program execution. Kiczales defines the terminology of AspectJ as follows: "*join points* are well-defined points in the execution of the program; *pointcuts* are a means of referring to collections of join points and certain values at those join points; *advice* are method-like constructs used to define additional behavior at join points; and *aspects* are units of modular crosscutting implementation, composed of pointcuts, advice, and ordinary Java member declarations" [21]. In summary, a join point can trigger execution of advice during the simulation run, thus acting as an observer of experimental quantities.

At first it may not be clear whether it is necessary to mimic the real-world process of instrumentation. Is this approach just for the sake of mimicry? If not, why use instrumentation rather than standard testing procedures? Why not skip the data recording and costly simulation execution and instead break down a model into pieces and test in a divide-and-conquer manner, as in unit testing? For one, this process can be extremely time consuming, and difficult to automate, as it usually requires design expertise. Additionally, for exploratory work like scientific experimentation, system testing is more suited to identify *when* or *where* a mechanism exists when its workings are previously unknown. Finally, the questions asked as scientific hypotheses often involve the interactions between components. By taking a temporal approach to verification modeling, we can analyze these interactions formally.

2.4 Formal Verification

Model checking is a formal verification technique for finite-state concurrent systems. It was first comprehensively presented by Clarke, Grumberg, and Peled [22] as a method of verifying sequential circuit designs and communication protocols. The process of model checking consists of several tasks, including modeling, specification, and verification. In the modeling step, a design is converted into a model that is compatible with model checking software by either abstraction (to eliminate irrelevant details) or compilation. The specification step involves defining the properties that the design must satisfy. These properties are generally defined in terms of temporal logic formulae, which can be used describe system behavior over time. Finally, the verification step involves the measures taken to reduce the state explosion and evaluate the temporal properties on the model from the previous steps [22].

Using model checking as a tool for formal verification is now a well-established methodology. As a concise overview of its ability, it is expressed that safety ("Bad things will never happen") and liveness ("Good things will eventually happen") properties verification will ensure component validity. Components that violate these properties exhibit illegal behavior. The process of model checking is to first define the model, then check whether the properties hold by means of assertions (invariants) and temporal logic formulae. When a property is violated, the model checker provides a counterexample. The counterexample is represented as a sequence of actions that leads to an error [23].

So, a component checked by a model checker will either pass the tests or the model checker will identify the point of failure. It is also observed that, "formal verification usually only applies to models of limited size, which means that only abstract models of complex programs can be verified," however, it is highlighted that, "in general, anything can be tested with model checkers if there is a suitable model, that is sufficiently abstract to allow verification in realistic time" [23]. In accordance with this claim, the proposed hypothesis testing system constructs an abstract model of the simulation for testing. Several use cases exist in which the model checking method has been successful, ranging from the correctness of music playing software to biomedical applications. In these use cases, it is shown that model checking is applicable to applications "exhibiting probabilistic, nondeterministic and real-time characteristics" [24].

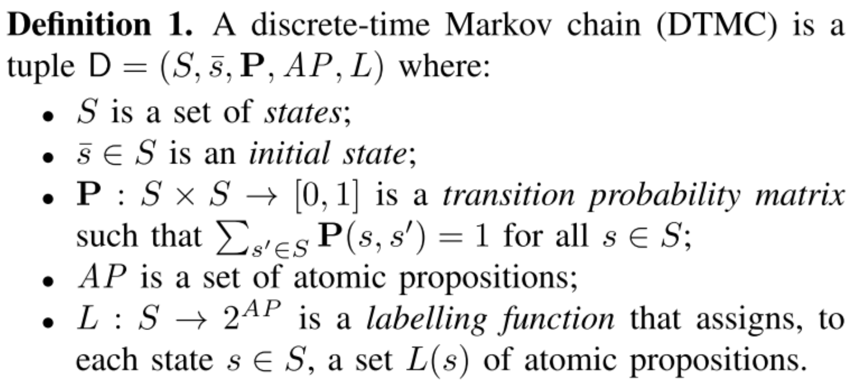
To speak briefly on the utility of model checking, Holtzmann and Smith (1999) explain some of the challenges present with the formal verification methodology. They claim that its practicality can be limited because, while, "to check the accuracy of a formal model for any application of substance can be significant, often consuming weeks or months," the software application development will often accelerate beyond the period of relevance for the verification results. However, this challenge may be overcome by means of abstraction and automation. The solution they propose is, "to devise a way to map a given program into an element of this subset by some well-defined process. That well-defined process is called abstraction, and the result of this process can be called a verification model." By use of the verification model, the state explosion of complex systems becomes irrelevant.

2.5 Probabilistic Model Checking

Probabilistic model checking [25] is defined as a method for modeling and analysis of systems with stochastic behavior. It is a variant of *model checking* [26], which is a formal method for verifying the correctness of real-life systems. The difference between the two formalisms is that the input to a probabilistic model checker is a model which includes quantitative information about the likelihood of state transitions and the times at which they occur. Additionally, temporal properties can be specified in terms of a probability that the property is satisfied [27].

The case study for this paper utilizes the open-source **Pr**obab**i**listic **S**ymbolic **M**odel Checker (PRISM) for hypothesis evaluation. The newest version of PRISM provides "formal modelling and analysis capabilities for systems with probabilistic, nondeterministic, and real-time characteristics, through support for verification of (priced) probabilistic timed automata" [28]. These capabilities are necessary to check models exhibiting unreliable or unpredictable behavior, as is the case for the models developed to aid the process of scientific discovery, since those models are built by the process of inductive reasoning and exhibit random behavior to model mechanisms that are not yet fully understood [29].

In this study, simulation system behavior is modeled as a Discrete-Time Markov chain (DTMC)[30]. A DTMC is defined as the following:



Definition 1. A discrete-time Markov chain (DTMC) is a tuple $\mathsf{D} = (S, \bar{s}, \mathbf{P}, AP, L)$ where:

- S is a set of *states*;
- $\bar{s} \in S$ is an *initial state*;
- $\mathbf{P} : S \times S \to [0, 1]$ is a *transition probability matrix* such that $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ for all $s \in S$;
- AP is a set of atomic propositions;
- $L : S \to 2^{AP}$ is a *labelling function* that assigns, to each state $s \in S$, a set $L(s)$ of atomic propositions.

> For a state $s \epsilon S$ of a DTMC D, the probability of moving to a state $s' \epsilon S$ in one discrete step is given by P(s, s'). Each state in D represents one possible configuration of the system being modelled; each transition represents the possibility to evolve from one configuration to another. A path of D, which gives one possible evolution of the Markov chain, is a sequence of states $s_0 s_1 s_2 ...$ such that $s_0 = \bar{s}$ and $P(s_i, s_{i+1}) > 0$ for all $i \geq 0$. [24]

By producing a model of an experiment with a DTMC, we gain an increasingly accurate model as more experiments are performed and more data is collected. Evaluation of this type of model provides verification of hypotheses with a certain error margin rather than true exhaustive model checking, which aims to build a verification model as a deep copy of the simulation model source code and claim with complete certainty that a model is correct based on the property specifications.

2.6 Property Specification Patterns

A property specification pattern is a high-level characterization of a common requirement on the acceptable state/event sequences in a finite-state model of a system [31]. These patterns are modeled after the philosophy of design patterns. That is, they are a means of applying expert knowledge to a diverse set of problems. The motivation for developing this set of patterns was to address the concerns that prevent wide-scale use of model checking as a formal method of software verification. These concerns are based in the observation that the applied use of temporal logic formalisms is difficult even for model checking practitioners and researchers. Additionally, once the formulae are composed, they are still difficult to reason about, debug, and modify [32].

Dwyer, Avrunin, and Corbett propose a collection of patterns to solve this problem. They are divided into three categories: occurrence patterns, ordering patterns, and compound patterns. Among the occurrence patterns are absence, existence, bounded existence, and universality. The ordering patterns are precedence, and response, and the compound patterns include chain precedence, chain response, and boolean combinations [31]. Each of these patterns can be specified using high-level language. For example, the occurrence patterns can each contain an "occurs" keyword and an ordering, such as "A occurs before B." This high-level characterization lends itself nicely to the development of a domain specific language for hypothesis testing.

2.7 Explanatory Coherence Networks

Explanatory coherence theory [33, 34] offers a set of principles and methods for studying the degree of coherence existing between a set of hypotheses. The hypotheses are connected forming a network, which is then balanced to determine which hypotheses cohere with one another and which ones do not. This is called a coherence network. The coherence network tells us how well the hypotheses (and the observed evidence resulting from an experiment) work together. Our confidence in the truth of a new hypothesis will be affected if it coheres with a network of previously held hypotheses.

A coherence network is a qualitative model that represents a set of propositions and their relations to one another. The propositions may be in the form of a hypothesis or in the form of an evidence which may either support or refute a hypothesis. The theory of explanatory coherence is informally stated in the following principles:

> Principle E1. Symmetry. Explanatory coherence is a symmetric relation, unlike, say, conditional probability.
>
> Principle E2. Explanation. (a) A hypothesis coheres with what it explains, which can either be evidence or another hypothesis; (b) hypotheses that together explain some other proposition cohere with each other; and (c) the more hypotheses it takes to explain something, the lower the degree of coherence.
>
> Principle E3. Analogy. Similar hypotheses that explain similar pieces of evidence cohere.
>
> Principle E4. Data priority. Propositions that describe the results of observations have a degree of acceptability on their own.
>
> Principle E5. Contradiction. Contradictory propositions are incoherent with each other.
>
> Principle E6. Competition. If P and Q both explain a proposition, and if P and Q are not explanatorily connected, then P and Q are incoherent with each other. (P and Q are explanatorily connected if one explains the other or if together they explain something.)
>
> Principle E7. Acceptance. The acceptability of a proposition in a system of propositions depends on its coherence with them. [33]

Units typically start with an activation level of 0, except for the special evidence unit whose activation is always 1. Activation spreads from it to the other units. Inhibitory links between units make them suppress each other's activation.

Activation of each unit aj is updated according to the following equations:

$$\begin{array}{ll} \textbf{If } net_j > 0 & a_j(t+1) = a_j(t)(1-d) + net_j(max - a_j(t)) \\ \textbf{Otherwise} & a_j(t+1) = a_j(t)(1-d) + net_j(a_j(t) - min) \end{array}$$

Here d is a decay parameter (say .05) that decrements each unit at every cycle, min is minimum activation (-1), max is maximum activation (1).

Based on the weight wij between each unit i and j, we can calculate neti, the net input to a unit, by:

$$net_j = \Sigma_i \left(w_{ij} * a_i(t)\right)$$

Chapter 3

Methodology

The hypothesis testing framework was developed by following the idioms of model-driven software development principles. In this regard, the development process is divided into three sections of related development, and each can be performed in parallel. The sections are namely Domain, Transformations, and Platform. The Domain section includes those steps of development which are related to the domain-specific language. This is the most abstract area, where language requirements are identified and the language is defined. In the Transformations section, the model transformations are defined to create implementation specific artifacts from the DSL text. In the platform section, the simulation is implemented for evaluating the usability of the DSL. The activity diagram in figure 3.1 illustrates the steps taken during this process.

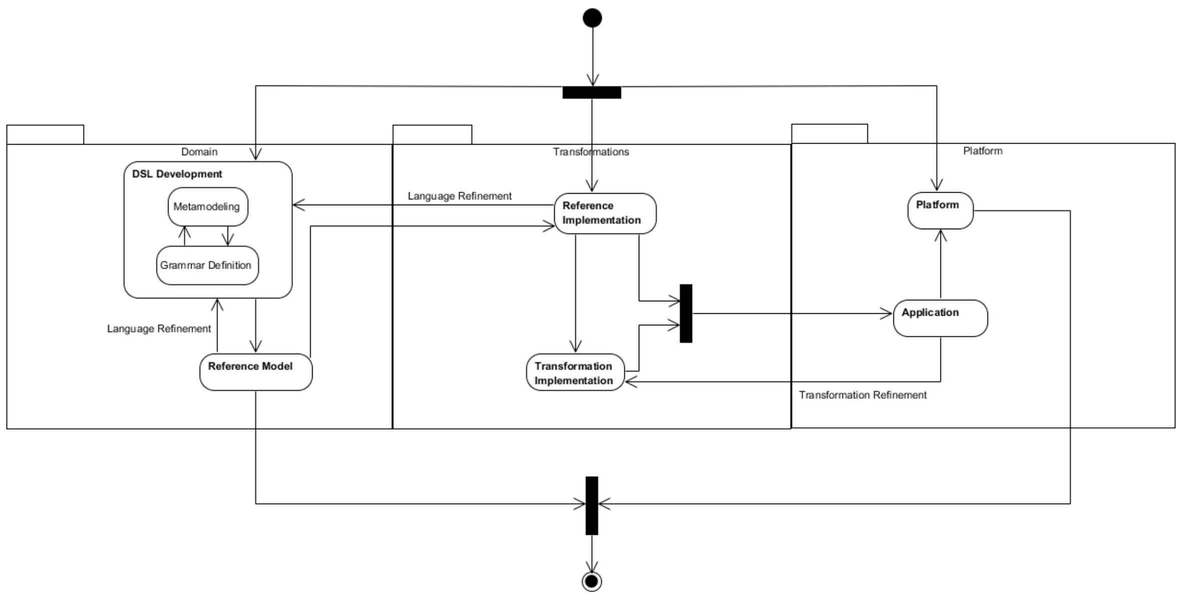


Figure 3.1: FASE Development Process

3.1 Domain

The first step in the Domain development process is to define the language that is to be used for the hypothesis testing framework. One of the most difficult parts of the process is defining the language in a way that enables developers with little programming or domain experience to be effective users [35]. To promote greater validity of the language, the DSL development step starts by rigorously defining a metamodel for the language in the form of a UML class model. This metamodel helps the language engineer to understand the domain, and is a basis for requirements specification. After the metamodel is developed, the formal syntax of the language is defined by taking the objects and associations from the metamodel and developing a context-free grammar in Backus-Naur Form (BNF). This is done by a process of assigning keywords to associations from the metamodel and defining rules for their legal order of occurrence. Once the language is defined in BNF, it can be implemented in a language engineering framework. An editor for the language may be provided by the framework, or one may be developed simultaneously. With the implementation of the language complete, a reference model will be constructed using the language. A reference model is a concrete example of the DSL in use. By evaluating the readability and writeability of the reference model, this step provides an opportunity to evaluate the usability of the language and refine it for unusual cases.

3.1.1 DSL Development

The initial stage of development for the FASE framework requires iterative refinement of the grammar. The process of developing the grammar from the metamodel keeps the language well structured, while providing visual aid to the developer.

- Metamodeling

The metamodeling step involves adding potential keywords to the language in the form of a UML class model. This model conveys the connections between the language elements and gives the initial representation of the domain concepts. After defining the metamodel, the next step should be to define the grammar in BNF. If any problems are found in the BNF definition, development should return to the metamodel, restructure the language or add/remove keywords, and return to BNF definition.

- Grammar Definition

Readers are reminded that a BNF grammar is defined as an ordered sequence of terminal and non-terminal elements. The grammar definition step requires that a BNF grammar is developed from the metamodel. To do this, one should consider leaf nodes of the diagram to be terminals in the grammar. There will be one node of the model that represents the entry point. Relationships defined in the metamodel such as composition or inheritance help to characterize the relationships in the grammar. For example, if an element is composed of several parts, it is a non-terminal which contains the modeled words in a certain ordering. An inheritance relationship indicates that there should be multiple orderings of the children of the parent non-terminal. After finalization of the BNF grammar, the syntax can be implemented in a language engineering framework.

3.1.2 Reference Model

Once the grammar is defined in a language engineering framework, it is possible to write sentences with syntax highlighting using the language. The reference model is defined as a vigorous exercise of the language. The language is exercised by attempting to develop all the sentences that are necessary for the framework to specify goals, hypotheses, and experiments. If there is difficulty in conveying a necessary element to any of these sections in a clear and concise way, additional refinement may be necessary.

3.2 Transformations

The Transformation section starts with development of a reference implementation. The reference implementation is an example of the code that would be required to do the actions that are expected of the DSL execution. By evaluation of the reference implementation, sections of code that are candidates for text-to-model transformations are identified. These transformations bridge the gap between reference model, reference implementation, and application.

3.2.1 Reference Implementation

The reference implementation is the backbone of the language. Most of the classes and methods that allow the language to function will be constructed through development of the reference implementation. This code base will be written in an API-style format, allowing for simple substitutions into method calls to perform the functions of the DSL.

3.2.2 Transformation Implementation

The transformation from DSL to Application is accomplished by use of source code templates. The reference implementation is used to identify sections of code that could be filled in with text from the DSL and used as arguments to methods that can handle these inputs.

3.3 Platform

The platform section includes the development of the simulation model and the generation of the system application. Once the simulation model is defined, the application can make slight modifications to it programmatically. In this case, the platform is altered to incorporate data recording mechanisms. By executing the DSL, outputs ultimately come from the platform, but are interpreted by the system application.

3.3.1 Simulation Platform

The implementation of the simulation platform can be incredibly diverse. Since the augmenting code is generated automatically, the framework can be applied to nearly any application, assuming it is written in a language compatible with an aspect-oriented framework.

3.3.2 Application

The application is a union of the code generated by the template engine and the reference implementation. This is what is finally executed after saving the DSL changes. If the application encounters any errors, development may need to return to the transformation section to revise the template parametrization.

Chapter 4

Implementation

The implementation of the FASE framework follows from the methodology of the previous chapter. First, we will give a high-level overview of the system behavior. Then we will discuss the development of the FASE language and how it is implemented in the language engineering framework, Xtext. Next, we will review how the template engine, Xtend is used to generate an evaluation model to carry out the tasks required by the DSL. Afterwards, we will present a detailed design of the reference implementation. Finally, we will take a look at the implementation of the In-Silico-Hepatocyte-Culture(ISHC) platform used in the case study of chapter 5.

4.1 Overview

Figure 4.1 presents a high-level overview of the activities that occur during the use of FASE. The only input to the process is the text of the DSL editor. After execution, the final product is a coherence network which models the current, holistic, understanding of the domain. The following items explain the details of each activity. Afterwards, the system is also presented as a sequence diagram to give a clear depiction of how data flows from one unit to another.

- User Saves GHE Specification

In this step, the user employs the GHE DSL to define an appropriate experiment specification, model description, hypotheses, and goals. More details on how these concepts are defined can be found in section 4.2. Once the specification is saved, the transformations take place, and the next activity is executed.

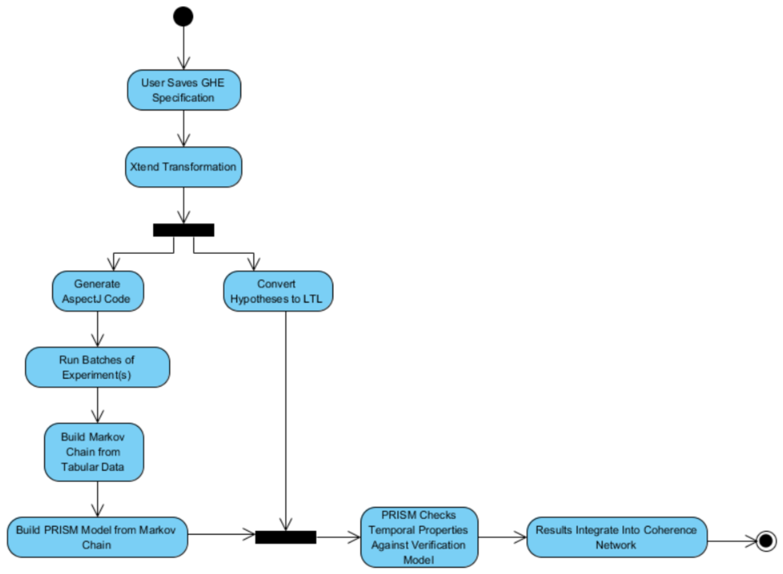

User Saves GHE Specification
Xtend Transformation
Generate AspectJ Code
Convert Hypotheses to LTL
Run Batches of Experiment(s)
Build Markov Chain from Tabular Data
Build PRISM Model from Markov Chain
PRISM Checks Temporal Properties Against Verification Model
Results Integrate Into Coherence Network

Figure 4.1: FASE System Activity Diagram

- Model Transformation

Here, the DSL text is used to fill in a template that corresponds with a generic driver for the system. After the driver generation is completed, it will immediately be executed, leading to the next step.

- Convert Hypotheses to LTL

In this step, the hypotheses from the specification will be taken and syntactically analyzed to identify the temporal property that they correspond to, and match them to a formula. Additionally, the condition will be identified as either an event or a logic statement so that the distinction can be made later at PRISM model building time.

- Generate AspectJ code

When execution reaches this point, the system generates pointcuts and advice to record the variables from the hypotheses.

- Run Batches of Experiments

Once the data recording elements have been generated, the experiments are run to collect that data into a table formatted file. Here, each row of the table represents a change in one of the variables' value.

- Build Markov Chain from Tabular Data

In this section, the data from the experiments is used to build a Markov Chain. Transitions in the chain are determinant upon the ranges defined in the hypotheses and/or whether the condition is an event. Repeat states increase the likelihood, while not adding the same state twice.

- Build PRISM Model from Markov Chain

Here, the Markov chain is used to generate the PRISM model by converting each state into a statement in the PRISM language. These statements include the state name, transitions, and probabilities associated with the transitions.

- PRISM Checks Temporal Properties Against Verification Model

After the PRISM model is built, it is executed. The PRISM system constructs an omega automata from the LTL formula, and a DTMC from the generated PRISM code. To evaluate the validity of the model compared to the temporal property, the model checker builds a cross product of the omega automata and the DTMC, and evaluates the reachability of the bottom strongly connected components.

- Model Checking Results Integrate Into Coherence Network

When model checking is completed, the result is a probability of acceptance of the LTL property. This value is interpreted as a confidence interval based on number of experiments performed, and incorporated into a coherence network representing the current knowledge base of domain experiments. Once the new data is added to the network, competing hypotheses are evaluated and the network is stabilized. The resulting network is displayed to the user.

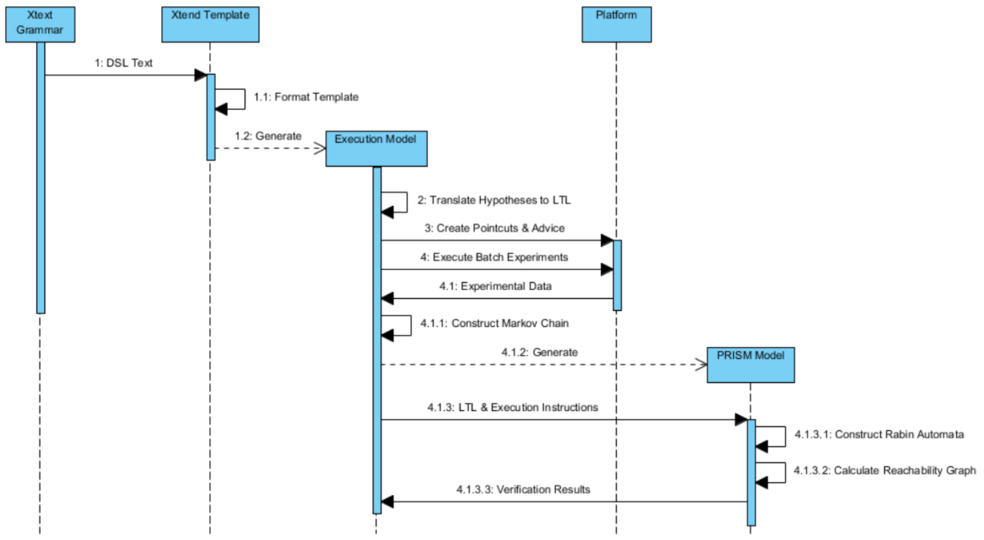

Xtext Grammar
Xtend Template
Platform
1: DSL Text
1.1: Format Template
1.2: Generate
Execution Model
2: Translate Hypotheses to LTL
3: Create Pointcuts & Advice
4: Execute Batch Experiments
4.1: Experimental Data
4.1.1: Construct Markov Chain
4.1.2: Generate
PRISM Model
4.1.3: LTL & Execution Instructions
4.1.3.1: Construct Rabin Automata
4.1.3.2: Calculate Reachability Graph
4.1.3.3: Verification Results

Figure 4.2: FASE System Sequence Diagram

4.2 FASE Language

In this section, we will explain the development and design of the FASE language. Readers will be reminded that the FASE framework is an extension of the Goal-Hypothesis-Experiment framework that addresses the ”hypothesis” section of the framework’s concerns. The development of the FASE language started with the creation of a metamodel of the terms needed to support property specification, acceptance definitions, condition declarations, and model connection. In 4.2.1, the structure of the GHE framework is presented, and the parts addressed by FASE are highlighted. In figure 4.3, the hypothesis testing-specific concepts are in blue, and the experiment management concepts are in white, and are presented at a higher level. For more information on the experiment management terms, see [17]. The terms for hypothesis testing are discussed in detail in section 4.2.2.

4.2.1 Structure

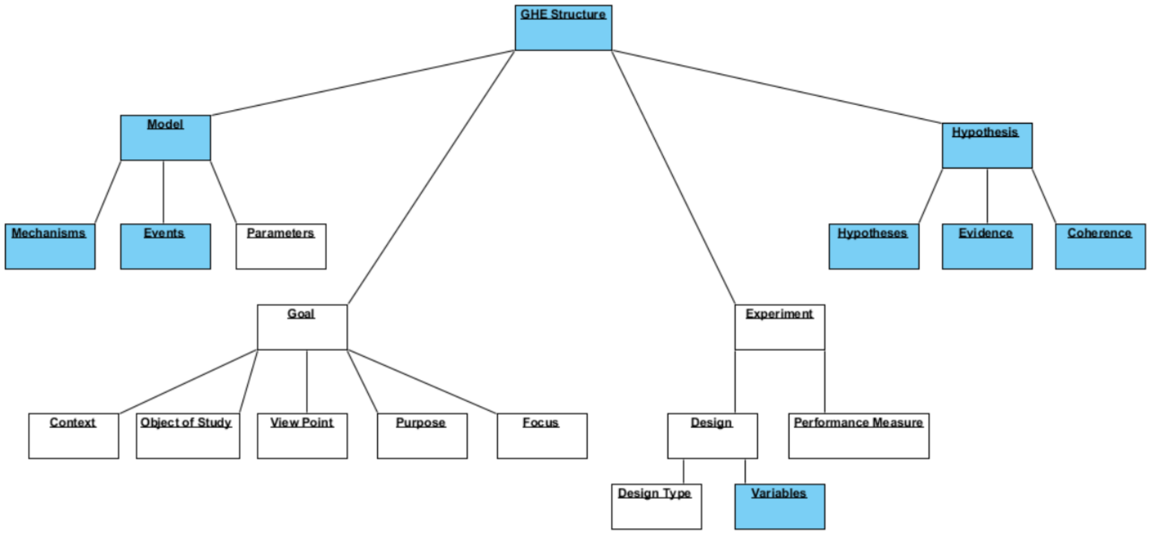


Figure 4.3: Goal-Hypothesis-Experiment Language Structure

- Model

The Model section has 3 elements: Mechanisms, Events, and Parameters. Mechanisms represent the actions that are needed to produce a phenomena. A mechanism could be an object, a method, or a section of code. Mechanistic hypotheses are in terms of mechanisms and their outcomes. Events are declarations of methods and their system path. These are incorporated in the model section to increase the readability of the hypothesis section, and because events are a characteristic of the model. Finally, parameters are the variables that are under scrutiny in the execution of the experiment.

- Goal

The Goal section has 5 elements: Context, Object of Study, View Point, Purpose, and Focus. These elements represent the conceptual level of the experiment. They define the targets of experimentation, which in turn aids in the experimentation and evaluation steps of the process.

- Hypothesis

In this section of the DSL, the hypotheses, evidence, and coherence model are defined. A hypothesis can be either phenomenological (concerning inputs and expected outputs), or mechanistic (concerning inner workings of the simulation). A mechanistic hypothesis can be either a fine-grain, or higher-order hypothesis. A higher-order hypothesis is defined as an instantiation of a number of mechanisms from the model section, with temporal properties to describe the order the mechanisms should occur in, while a fine-grain hypothesis targets a specific working, such as variables and/or events. Evidence represents hypotheses that a user considers irrefutable. Hypotheses and evidence are connected in the coherence model.

- Experiment

In the experiment section, there is a design specification, which includes design type and variables, and a performance measure specification. Design type determines how many view points should be examined in the experiments. It can define either a full factorial design, fractional factorial, or a custom design type. The variables element allows a user to define which variables are dependent or independent. Finally, the performance measure element defines the margin of error allowed for the experiment to be considered a success.

4.2.2 Syntax

The syntax of the FASE language was developed by first modeling the words needed and their relationships in UML. Figure 4.4 represents the metamodel of the language elements. In this section, we will explain what each part of the language is for, and present key parts of the grammar in Xtext.

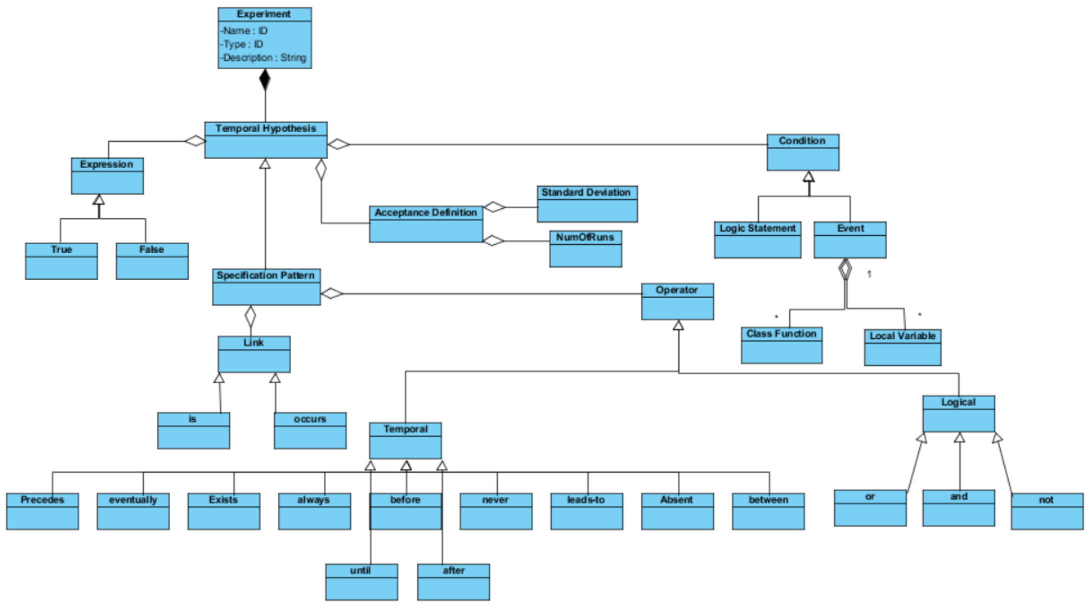


Figure 4.4: FASE DSL Metamodel

A temporal hypothesis consists primarily of a specification pattern and conditions. A specification pattern is a specific ordering of temporal operators with certain lexical links (is, occurs) that are included for the sake of readability. An acceptance definition can be added in the special case where a hypothesis is acting as evidence. An expression represents a logical assertion (true or false) for a boolean variable, for an example, see the property *condition = true until flag >10.*

- Xtext Grammar

For the sake of illustration, a section of the Xtext grammar is given below. The non-terminals for each of the main sections (Model, Goal, Hypothesis, Experiment) are defined by *Experiment Ontology*, and the Hypothesis section is displayed afterwards with its own set of non-terminals.

```
ExperimentOntology :
        ModelSection | Goals | Hypothesis | Experiment
;
```

```
Hypothesis :
          'hypothesis'
          '{'

          ('mechanistic' 'hypothesis'  '{'
          (mechHypothesis += MechHypothesis)*
          '}')?

          ('evidence' '{'
          (evidences += Evidence)*
          '}')?

          ('coherence' 'model' '{'
          (coherenceLinks += CoherenceLink)*
          '}')?

          '}'
;
```

4.3 Reference Model

The hypothesis testing sections of the reference model span the model and hypothesis modules of the subsuming GHE language. However, the bulk of the hypothesis testing constructs reside in the hypothesis section. It is in this section that we define what the hypothesis is and what evidence supports it. Hypotheses are defined in terms of mechanisms which are found in the model section. The model section also contains support for hypothesis testing, because it is where variables and event paths are defined. The following listing gives an example of the reference model text.

```
hypothesis
{
        mechanistic hypothesis
        {
                H3 : M1 occurs before M2
        }

        evidence
        {
                E1: inflammation occurs after inflammatoryAgent >
                inflammatoryAgentThreshold
                activation weight : 0.5
                E2: inflammation is absent after cytokine <
                cytokineThreshold
                activation weight : 0.5
        }

        coherence model
        {
                EXPLAIN (H1)(E1)
                EXPLAIN (H1 H2)(E1)
                ANALOGOUS (H1)(H2)
                DATA (Experiment1)(E1 E2)
        }
}
```

4.4 Transformations

Transformations are defined using Xtend, a template-based model generation extension of Xtext. The Xtend templates are written as a driver for the hypothesis testing framework. Grammar elements are extracted from the specification and substituted into the template.

4.5 Reference Implementation

The reference implementation is the Java code that should execute as a result of running the DSL. It performs 5 specific functions: converting the hypothesis specification to an LTL property, adding pointcuts to the simulation model, running the simulation model in batches and recording data, constructing a verification model from that data, and verifying the temporal properties are not invalidated by the verification model. The nature of the reference implementation is such that the DSL text can be taken and transferred into API calls by a text-to-text transformation to carry out the above functions. Some of the vital classes are described below.

- HypothesisTesting.java

This class takes the hypothesis definition from the DSL and converts it to an LTL formula. Hypotheses have to be in the form of a property specification pattern or else the DSL will not validate. The process of conversion is aided by an XML file that contains the formula version of each specification pattern. Thus, in order to convert a hypothesis into a temporal logic formula, the process comes down to two steps: identifying the pattern and substituting the variable text into the formula.

- Property.java

This class is a helper to HypothesisTesting.java. It identifies what specification property the hypothesis belongs to and decomposes its parts, identifying conditions and events.

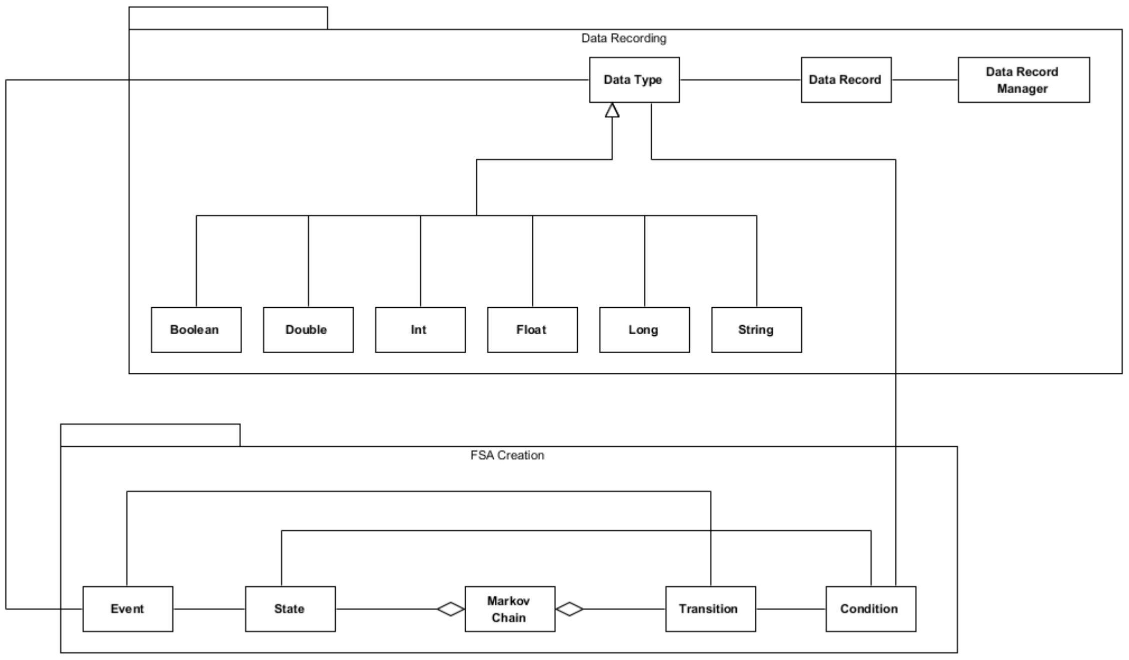


Figure 4.5: Data Recording & FSA Metamodel

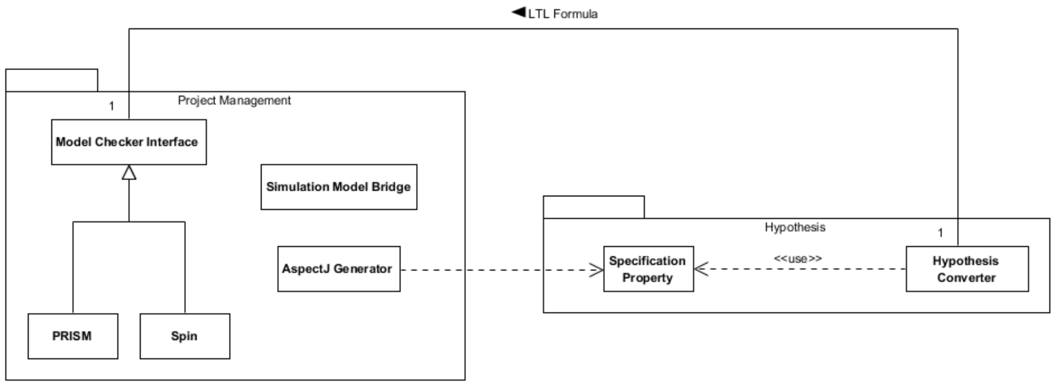


Figure 4.6: Hypothesis & Project Management Metamodel

- AspectJGenerator.java

This class defines the pointcuts based on the events from the model section of the DSL. It also generates the advice that is used to record the value of variables. Pointcuts are defined by taking the variables from the hypothesis(-es) and identifying those accessor methods that correspond with these variables. Whenever a variable value is changed, its accessor method will be called and the pointcut will be activated and the advice will cause the variable to be recorded into a table in a data file.

- SimulationModelInterface.java

This class executes the simulation model in batches in order to observe the experiment from different possible outcomes. Due to random variables and probabilistic simulations, it is important to get as much data as possible before construction of the verification model. This is because we are making assertions based on inductive reasoning, and the error rate is higher when there is less data.

- DataRecordManager.java

This class is used to structure the incoming data from the simulation into a matrix of abstract data types. Once the simulation completes, it writes the matrix into a file for later use by the Markov Chain builder.

- MarkovChain.java

This class constructs the verification model from the simulation data. It reads through the data file that the Data Record Manager created and adds new states when they are encountered. If a new state is satisfiable by an old state, a new transition is added instead, or the transition probability is increased.

- Condition.java

This class represents a condition range, such as A <B. It checks whether a set of values can satisfy this condition definition. This simplifies the code in the Markov Chain class by making it simple to identify if a condition range is exceeded by the data set (meaning a new state or transition should exist).

- PrismInterface.java

This class generates the PRISM model from the Markov Chain and runs the model checker. The PRISM model is generated by first writing the parameters to specify that the model is a DTMC, and to start the model specification. After the initial code is written, the class begins processing each state in the Markov Chain and creating a line of PRISM code for each state and its transitions. When the process is complete the entire Markov Chain will be modeled in the PRISM language and the model will be executed with the LTL formula generated earlier by the Hypothesis Testing class.

4.6 Platform

Figure 4.7 is a simplified metamodel of the implementation of the ISHC simulation used in the case study of Chapter 5. This system is an agent-based model representing an In-Silico Hepatocyte Culture, written in the MASON simulation framework. The model is primarily concerned with how certain cells react when introduced to chemical compounds. A comprehensive review of the mechanisms of the model is outside the scope of this document; however, a closer look at the metamodel can be found in Appendix B, and reviewing [36] will give a detailed explanation of the mechanisms.

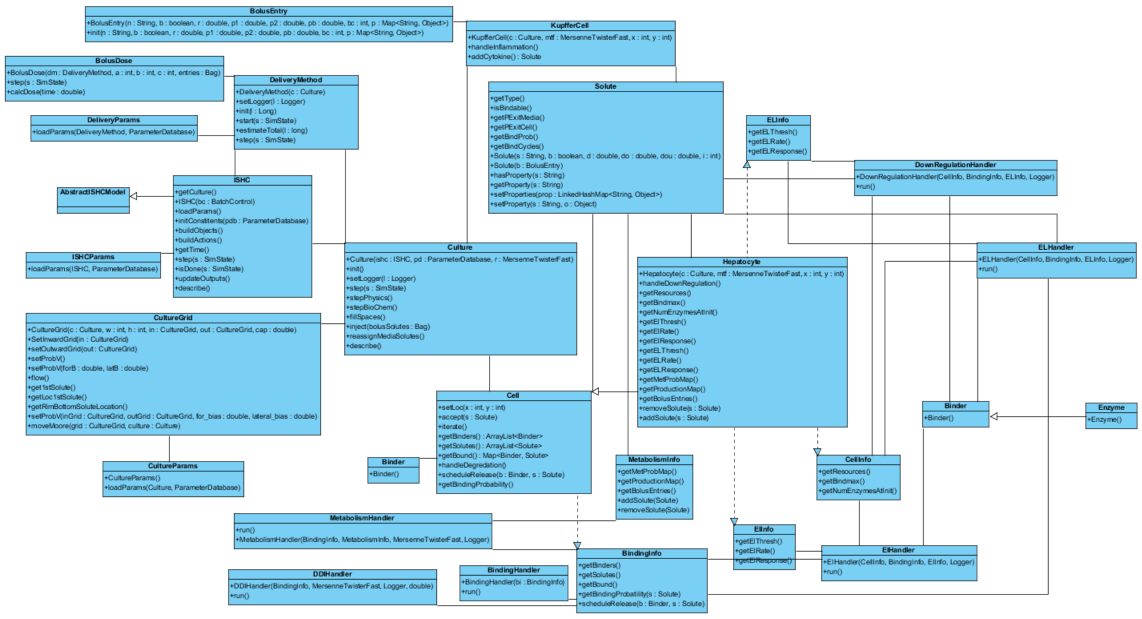

BolusEntry
+BolusEntry(n : String, b : boolean, r : double, p1 : double, p2 : double, pb : double, bc : int, p : Map<String, Object>)
+init(n : String, b : boolean, r : double, p1 : double, p2 : double, pb : double, bc : int, p : Map<String, Object>)
KupfferCell
+KupfferCell(c : Culture, mtf : MersenneTwisterFast, x : int, y : int)
+handleInflammation()
+addCytokine() : Solute
BolusDose
+BolusDose(dm : DeliveryMethod, a : int, b : int, c : int, entries : Bag)
+step(s : SimState)
+calcDose(time : double)
DeliveryMethod
+DeliveryMethod(c : Culture)
+setLogger(l : Logger)
+init(l : Long)
+start(s : SimState)
+estimateTotal(l : long)
+step(s : SimState)
DeliveryParams
+loadParams(DeliveryMethod, ParameterDatabase)
Solute
+getType()
+isBindable()
+getPExitMedia()
+getPExitCell()
+getBindProb()
+getBindCycles()
+Solute(s : String, b : boolean, d : double, do : double, dou : double, i : int)
+Solute(b : BolusEntry)
+hasProperty(s : String)
+getProperty(s : String)
+setProperties(prop : LinkedHashMap<String, Object>)
+setProperty(s : String, o : Object)
ELInfo
+getELThresh()
+getELRate()
+getELResponse()
DownRegulationHandler
+DownRegulationHandler(CellInfo, BindingInfo, ELInfo, Logger)
+run()
AbstractISHCModel
ISHC
+getCulture()
+ISHC(bc : BatchControl)
+loadParams()
+initConstitents(pdb : ParameterDatabase)
+buildObjects()
+buildActions()
+getTime()
+step(s : SimState)
+isDone(s : SimState)
+updateOutputs()
+describe()
ISHCParams
+loadParams(ISHC, ParameterDatabase)
Culture
+Culture(ishc : ISHC, pd : ParameterDatabase, r : MersenneTwisterFast)
+init()
+setLogger(l : Logger)
+step(s : SimState)
+stepPhysics()
+stepBioChem()
+fillSpaces()
+inject(bolusSolutes : Bag)
+reassignMediaSolutes()
+describe()
ELHandler
+ELHandler(CellInfo, BindingInfo, ELInfo, Logger)
+run()
Hepatocyte
+Hepatocyte(c : Culture, mtf : MersenneTwisterFast, x : int, y : int)
+handleDownRegulation()
+getResources()
+getBindmax()
+getNumEnzymesAtInit()
+getElThresh()
+getElRate()
+getElResponse()
+getELThresh()
+getELRate()
+getELResponse()
+getMetProbMap()
+getProductionMap()
+getBolusEntries()
+removeSolute(s : Solute)
+addSolute(s : Solute)
CultureGrid
+CultureGrid(c : Culture, w : int, h : int, in : CultureGrid, out : CultureGrid, cap : double)
+SetInwardGrid(in : CultureGrid)
+setOutwardGrid(out : CultureGrid)
+setProbV()
+setProbV(forB : double, latB : double)
+flow()
+get1stSolute()
+getLoc1stSolute()
+getRimBottomSoluteLocation()
+setProbV(inGrid : CultureGrid, outGrid : CultureGrid, for_bias : double, lateral_bias : double)
+moveMoore(grid : CultureGrid, culture : Culture)
Cell
+setLoc(x : int, y : int)
+accept(s : Solute)
+iterate()
+getBinders() : ArrayList<Binder>
+getSolutes() : ArrayList<Solute>
+getBound() : Map<Binder, Solute>
+handleDegredation()
+scheduleRelease(b : Binder, s : Solute)
+getBindingProbability()
Binder
+Binder()
Enzyme
+Enzyme()
CultureParams
+CultureParams()
+loadParams(Culture, ParameterDatabase)
Binder
+Binder()
MetabolismInfo
+getMetProbMap()
+getProductionMap()
+getBolusEntries()
+addSolute(Solute)
+removeSolute(Solute)
CellInfo
+getResources()
+getBindmax()
+getNumEnzymesAtInit()
MetabolismHandler
+run()
+MetabolismHandler(BindingInfo, MetabolismInfo, MersenneTwisterFast, Logger)
ElInfo
+getElThresh()
+getElRate()
+getElResponse()
BindingInfo
+getBinders()
+getSolutes()
+getBound()
+getBindingProbatility(s : Solute)
+scheduleRelease(b : Binder, s : Solute)
ElHandler
+ElHandler(CellInfo, BindingInfo, ElInfo, Logger)
+run()
DDIHandler
+DDIHandler(BindingInfo, MersenneTwisterFast, Logger, double)
+run()
BindingHandler
+BindingHandler(bi : BindingInfo)
+run()

Figure 4.7: ISHC Metamodel

Chapter 5

Case Study

Here we will walk through each of the above steps with a concrete example to demonstrate the process. Our case study will be of the In-Silico Hepatocyte Culture (ISHC) simulation developed by Petersen et. al [36].

5.1 Hypothesis Space

In the hypothesis space, the model, experiment, and hypotheses are defined.

5.1.1 Simulation Model

In this study, we will present a section of the ISHC model that will relate to the hypotheses about metabolic capacity and inflammation, which will be the focus of these experiments. At a high level, Figure 5.1 shows the mechanism to control inflammation, and below are details on the mechanism's workings.

> INFLAMMATION HANDLER maps to cytokine production in response to an inflammatory stimulus like LPS; it is unique to KUPFFER CELLS. It first counts how many INFLAMMATORY STIMULI are in the CELL. An inflammatory stimulus is any SOLUTE for which the Boolean property inflammatory is true. In simulations here, only LPS is an INFLAMMATORY STIMULUS. If the number of INFLAMMATORY STIMULI exceeds the value of inflammatoryThreshold, a random draw from U[0,1) determines whether to produce a CYTOKINE. The associated probability is determined by:

$$pCyotkine = 1 - \exp\left(-\frac{\#stimuli - inflammatoryThreshold}{exponentialFactor}\right)$$

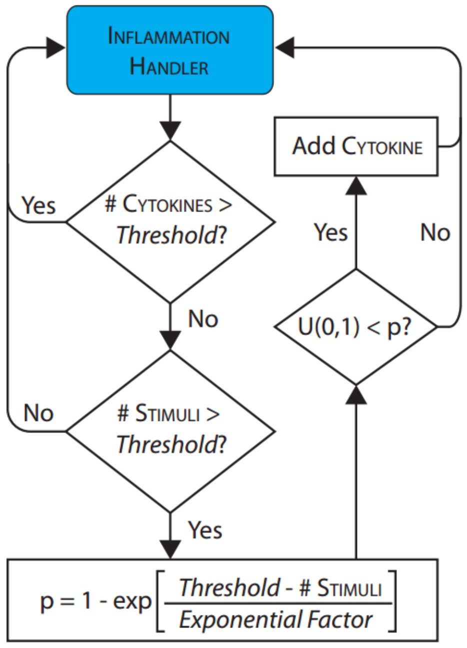


Figure 5.1: Inflammation Handler Mechanism

Thus, exponentialFactor controls the degree to which increasing stimulus causes CYTOKINE formation. Lastly, to prevent excess CYTOKINE formation in the presence of significant INFLAMMATORY STIMULUS, CYTOKINE cannot be created if there are already more than cytokineThreshold CYTOKINES in the CELL. The above equation is an example of a biomimetic agent rule. We lack sufficient detailed knowledge to describe exactly how a Kupffer cell in a particular lobule location determines whether to produce cytokine. The equation is an in silico placeholder for a yet to be specified fine-grain mechanism. A risk of relying on such rules is committing inscription error, the logical fallacy of assuming the conclusion and programming in (consciously or not) aspects of the result we expect to see (Smith, 1996). Cognizant of inscription error, we were careful not to explicitly encode a sigmoidal dose-response into INFLAMMATION HANDLER. We chose the above equation to mimic a Poisson distribution in which the probability of at least one binding event occurring within a simulation cycle depends on the extent to which the number of inflammatory stimuli exceeds a threshold value. The ability to generate a sigmoidal dose-response curve arises from a complex interplay among multiple analog mechanisms, facilitated by flexibility in the parameters inflammatoryThreshold and exponentialFactor.[36]

The code for this mechanism is the handleInflammation() function of the KupfferCell class:

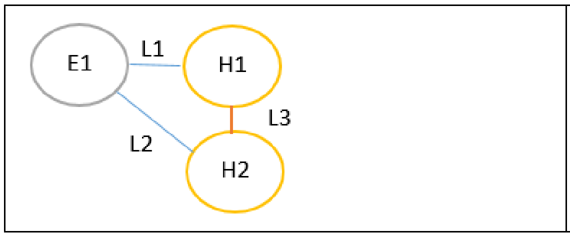
Listing 5.1: Inflammation Handler

```
//Count the number of stimuli and Cytokines in the cell
        int numInflammatoryStimuli = 0;
        int numCytokines = 0;
        for(Object o : solutes)
        {
            Solute s = (Solute) o;
            if(s.hasProperty("inflammatory") &&
            ((Boolean)s.getProperty("inflammatory")))
            {
                numInflammatoryStimuli++;
            }
            if(s.type.equals("Cytokine"))
            {
                numCytokines++;
            }
        }

        //If past Cytokine threshold, stop creating Cytokines
        if(numCytokines >= parent.cytokineThreshold)
        {
            return;
        }
```

```
//If past inflammation threshold, there's a chance to
produce Cytokine
if(numInflammatoryStimuli >=
parent.inflammatoryStimulusThreshold)
{
    double probability = 1.0 -
    Math.exp(-1*(numInflammatoryStimuli -
    parent.inflammatoryStimulusThreshold)
    / parent.exponentialFactor);

    double draw = rng.nextDouble();
    if(draw <= probability)
        addCytokine();
}
```

5.1.2 Coherence Model

Assume we are given the following coherence model from past observations (L3 indicates inhibition; L1 and L2 indicate coherence):



Activation of E1 = 1
Activation of H1 = 1
Activation of H2 = 0
Weight of L1 = 0.5
Weight of L2 = 0.5
Weight of L3 = -0.5

Figure 5.2: Initial Coherence Model

```
hypothesis 1 (H1) = inflammatoryAgent < inflammatorythreshold
```

```
    occurs before INCREASE in cytokine

    hypothesis 2 (H2) = inflammatoryAgent < inflammatorythreshold
    occurs after INCREASE in cytokine

    evidence 1 (E1) = inflammatoryAgent DECREASES
    after cytokine > cytokinethreshold
```

After the experimentation is complete, the coherence model will be reevaluated.

5.1.3 Hypothesis

The goal of modeling a hepatocyte culture is to discover how the liver will respond to various drugs; therefore, we should be able to formulate hypotheses in terms of drugs and ISHC mechanisms. An example hypothesis may be as follows: In response to lipopolysaccharide, Kupffer cells down-regulate hepatic P450 levels via inflammatory cytokines, thus leading to a reduction in metabolic capacity. At this level of abstraction, the hypothesis is easily readable. With the help of a domain specific language and a domain ontology, the specification in code is shockingly similar to the hypothesis above, as we will see in section 5.6.

5.1.4 Domain Ontology

By defining a domain ontology, the DSL can target domain-specific experiments tailored to the variables of a specific simulation model. The selection of variables is manually specified at this point in development, but there is potential for an automatic variable recognition sweep of the source code, which would take less time and effort than manual specification. This xml file will be referenced when validating the DSL text to ensure that terms used in the DSL correspond with terms in the model.

Listing 5.2: Domain Ontology

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<model>
  <events>
          <event  id="inflammation"
          path="void␣ishc.model.KupfferCell.handleInflammation()"/>
  </events>
  <variables>
             <variable  id="inflammatoryAgent"
             path="void␣ishc.model.KupfferCell.handleInflammation()
␣␣␣␣␣␣␣␣␣␣␣.numInflammatoryStimuli"/>
             <variable  id="inflammatorythreshold"
             path="ishc.model.KupfferCell.parent
␣␣␣␣␣␣␣␣␣␣␣.inflammatoryStimulusThreshold"/>
             <variable  id="noOfCytokines"
             path="void␣ishc.model.KupfferCell.handleInflammation()
␣␣␣␣␣␣␣␣␣␣␣.numCytokines"/>
             <variable  id="cytokinethreshold"
             path="ishc.model.KupfferCell.parent.cytokineThreshold">
             <variable  id="Cytokines"
             path="ishc.model.KupfferCell.parent.solutes"/>
          </variables>
</model>
```

5.1.5 Experiment Design

Based on the model's parameters and their levels, hypotheses, and the goal of the experiment, a design is created that is used in subsequent steps of the experiment life-cycle.

The experiment design is defined by dependent variables, control variables, independent variables and their levels, constraints, and values, which, in turn, are mappings of the variables provided by the user. Based on this design, one can define what is known as a design matrix, which specifies the actual experimental runs. I.e., the combination of factor levels. The design matrix, and its effects on the simulation may be found in section 5.9. Following is the DSL specification for the ISHC experiment design.

Listing 5.3: Experiment Design

```
experiment Experiment1
{
    design
    {
        designType FULLFACTORIAL
        variables
        {
            independent variables
            {
                LPS are at levels : LOW where LOW is in the
                range 1.0 to 1.0
                TOL are at levels : LOW where LOW is in the
                range 1.0 to 1.0
                DZ are at levels : LOW where LOW is in the
                range 1.0 to 1.0
            }
            dependent variables
            {
                cytokines : type SIMPLE
            }
        }
    }
}
```

5.1.6 Temporal Specification Properties

Following are the properties we would like to check against the model:

```
inflammatoryAgent > inflammatorythreshold occurs before
    cytokine < 10
```

These properties will affect which variables are recorded during the simulation run. Only the information needed to verify these properties will be recorded. I.e., the inflammation and degredation events, and any changes in the inflammatoryAgent, inflammatorythreshold, cytokine, or cytokineThreshold variables.

5.1.7 Hypothesis Definitions

Mechanistic hypotheses are defined as an instantiation of mechanisms.

H3 : M1 occurs before M2

Mechanisms M1 and M2 are defined in the model section of the DSL:

```
mechanism M1 = inflammatoryAgent + KupfferCell
    [inflammatoryAgent > inflammatorythreshold]-> Cytokines
mechanism M2 = inflammatoryAgent + KupfferCell
    [noOfCytokines < cytokinethreshold] -> Cytokines
```

These mechanisms together represent the hypothesis we mentioned in section 5.2: In response to lipopolysaccharide, Kupffer cells down-regulate hepatic P450 levels via inflammatory cytokines, thus leading to a reduction in metabolic capacity.

5.2 Experiment Space

After submitting the changes in the DSL by saving the source document, control moves to the experiment space to transform the source to generate application code.

5.2.1 Code Generation

The code generation step involves selecting parts of sentences from the DSL specification and substituting them into a template for the driver of the application program. The temporal property from section 5.6 is substituted as an input to a method that converts it to an LTL formula:

```
Property p2 = new Property("inflammatoryAgent >
inflammatorythreshold occurs before cytokine < 10");
HypothesisTesting e1 = new HypothesisTesting("E1", p1);
String LTLe1 = e1.toLTL();
```

- Temporal Properties

The LTL formula generated by the previous step is as follows:

```
G !(inflammatoryAgent > inflammatorythreshold) W cytokine > 10
```

This property is now in the form of a temporal logic formula as a string. It is submitted to the model checker in the last step, section 5.12.

- AspectJ

The generated AspectJ code takes the path from the model section and the conditions from the HypothesisTesting objects to develop its pointcuts:

```
String[] inputList = e1.getConditions();
AspectJGenerator fileMgr = new AspectJGenerator("ishc.IshcBatch");
fileMgr.generate("answers", "Answer1", inputList);
```

- Simulation Executor

This generated code then executes the simulation a number of times equal to that specified by the dsl to gather data.

```
String modelPath = "C:/Mason/models/ISHC";
String modelName = "ucberkeley.src.sim.ishc.ISHCModel";
String parametersPath =
"C:/Mason/models/Ishc/batch/batch_params.xml";
SimulationModelInterface ishc = new
SimulationModelInterface(modelPath, modelName, parametersPath);
ishc.runSimulation(1);
```

5.2.2 Simulation Data

By executing the simulation, we accumulate data on the variables of interest from the hypotheses. Each time there is a change in one of the variables, its new value is recorded. Table 5.1 is a reduced table of values for the purposes of illustration.

inflammatoryAgent	inflammatoryThreshold	cytokine
int	int	int
0	0	1
1	0	1
0	0	2
0	1	2
0	0	2

Table 5.1: Simulation Data

5.2.3 Convert to DTMC

Figure 5.3 shows the result of analyzing the above data set and creating a Discrete-Time Markov Chain.

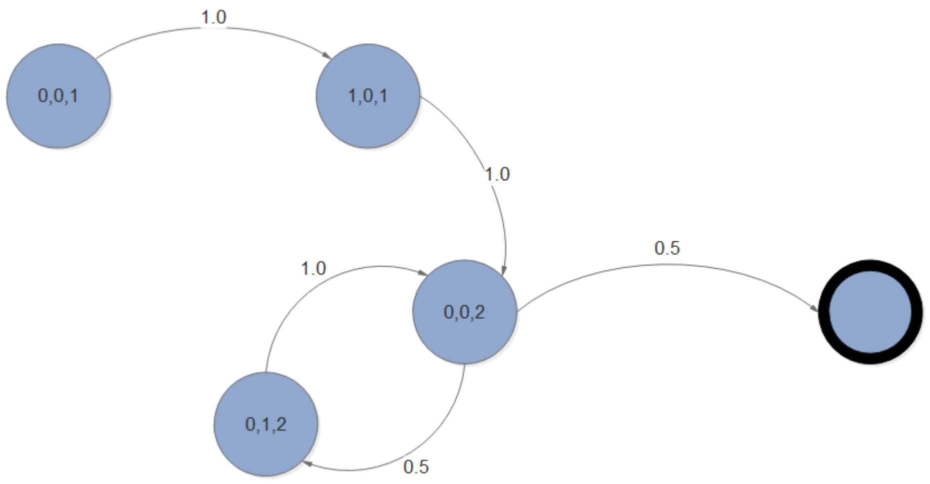


Figure 5.3: DTMC Representation of Simulation Data

5.2.4 Model Checking

With the Markov Chain constructed, the PRISM model is able to be produced by transforming each state into a statement in the PRISM language. As an example, a generated statement from state 0,0,1 in the chain of Figure 5.3 looks like this:

```
[] inflammatoryAgent = 1 & inflammatoryAgentThreshold = 0
& cytokine = 1 -> 1.0 : (inflammatoryAgent'=1)&
(inflammatoryAgentThreshold'=0)&(cytokine'=1);
```

This syntax shows the current state before the $->$ symbol, and the probability (1.0) that the values change to the new (prime) values after the $->$ symbol.

Upon running the model checker with the LTL formula from the previous step, we receive as output that the model is invalid with probability 1.0, and a state trace from 0,0,1 to 1,0,1.

5.2.5 Results

Upon receiving the result from the previous step, the coherence model from section 5.2 needs to be updated. The new model will include 3 hypotheses and the same evidence:

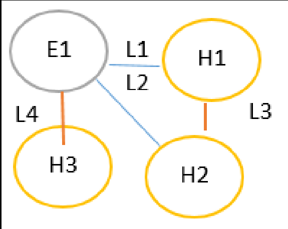
hypothesis 1 (H1) = inflammatoryAgent < inflammatorythreshold occurs before INCREASE in cytokine

hypothesis 2 (H2) = inflammatoryAgent < inflammatorythreshold occurs after INCREASE in cytokine

hypothesis 3 (H3) = inflammatoryAgent > inflammatorythreshold occurs before cytokine < 10

evidence 1 (E1) = inflammatoryAgent DECREASES after cytokine > cytokinethreshold

Since H3 was run with only one experiment execution, while H1 and H2 were run multiple times, its default activation weight is less. The resulting coherence model is as follows:



Activation of E1 = 1
Activation of H1 = 0.25
Activation of H2 = 0.25
Activation of H3 = -0.33
Weight of L1 = 0.33
Weight of L2 = 0.33
Weight of L3 = -0.33
Weight of L4 = -0.33

Figure 5.4: Final Coherence Model

This representation allows a user to easily see that, given 3 competing hypotheses, H1 and H2 have the strongest support.

Chapter 6
Evaluation

In this chapter, it will be shown how the model-driven methodology presented in chapter three was applied in the creation of FASE. First, we will discuss the process of developing a domain specific language for hypothesis testing in terms of conceptualization. In this section, it is shown how development of a metamodel of the language is an effective method of identification of relevant concepts to the language. The metamodel is then examined in the next subsection to reveal insights of value for the development of the language's grammar. The grammar is discussed in the next subsection as a Backus-Naur Form context-free grammar. Next, we address how the DSL for hypothesis testing is implemented. A reference model and reference implementation for the DSL is characterized, and finally, we take a look at the model verification aspects that are central to the hypothesis testing framework. This section includes how a verification model is constructed, as well as how the model checking process is initiated, and what is done with the verification results.

6.1 Language Conceptualization

The fundamental premise that led to the development of the FASE language was to bring the power and efficacy of formal verification to domain experts who are not trained in verification methodology, and to decrease the time required to perform verification. The end-user of the system is a scientist or simulation developer working with both in-vivo and in-silico experiments. He/she should be aware of scientific concepts like hypotheses, temporal properties, factors, and mechanisms. By providing the ability to interact with these concepts on a domain-specific platform, the scientist is given more time to focus on innovation, rather than being burdened by carrying out massive amounts experiments and performing analysis by hand.

In order to facilitate this goal, the language needed to be easily readable and naturally writable. The availability of model checking software allows for formal verification to happen autonomously with the aid of a verification model; however, model checking requires intimate understanding of the fundamentals of temporal logic, which is a difficult to understand and distinctly unnaturally writable requirement. The present solution to this predicament is to implement hypotheses as an abstraction of temporal logic, known as property specification patterns for finite-state verification (see section 2.7). These patterns represent the well-understood, but imprecise, conceptions of system behavior and provide translations into precise statements in common formal specification languages. In other words, the patterns offer a text-to-text transformation from high-level language text to temporal pattern formulae; which is exactly the goal of the language. As the project developed further, the language was expanded to include support for mechanism definitions and model declarations. This meant that the link between simulation software and the language had to be more tightly coupled. In this regard, the process initially requires scanning the simulation model to identify variables, then writing code in the DSL using variables from the simulation to phrase hypotheses in terms of those variables.

6.1.1 Metamodel

The metamodel for the hypothesis testing language consists of the components necessary to define hypotheses in terms of logical operators and conditions, the latter of which defined as either an event or a logic statement. A logic statement is a statement that can be evaluated to true or false at a given point in program execution. For example, x is less than 10, may be an condition which indicates a change in state. Events, on the other hand, may include the calling of a function, and do not have a value associated with them other than the fact that they occurred. The metamodel for the language can be found in Figure 6.1.

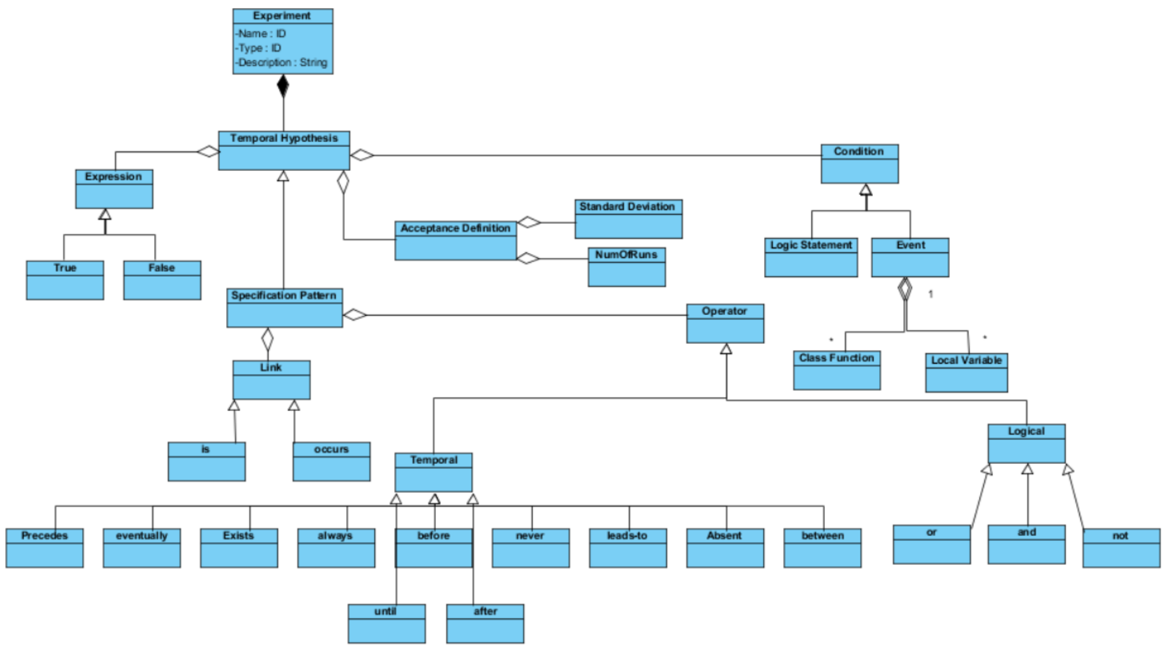


Figure 6.1: FASE Language Metamodel

The above metamodel was created by identifying keywords needed to produce the specification pattern requirements, and to provide support for defining conditions and a definition of acceptance for the hypothesis tests. In some cases the metamodel has overlap, where the same pattern can be written in multiple ways. For example, one of the aforementioned patterns, the precedence pattern, can be identified by "A precedes B," or "A comes before B." Here, "precedes" and "comes before" are to be keywords used in the language grammar. The question becomes, should both options be included to make the language more intuitive, or use only one option to avoid confusion over different syntax? Given that the goal of the language is to be user-friendly, it seems to be a harmless inclusion to support both natural sounding alternatives.

6.1.2 Grammar

To formally define the language, the metamodel was used to define a grammar for the DSL in BNF (introduced in chapter 3.2). The process of grammar definition consisted of developing basic sentences using the keywords in the metamodel, and generalizing the sample sentences into BNF. The process was complete when each specification pattern was able to be produced by the grammar. To consider the language a success, the following concerns still needed to be addressed:

1. Can LTL formulae easily be written and read?
2. Can formulae be in terms of mechanistic hypotheses from the experiment specification?
3. Can the DSL connect to the simulation model programmatically?
4. Can there be allowances for a certain degree of error in the model?

To fully address each of these concerns, the grammar required inclusion of additional features that aren't represented in the metamodel. To address (1), the definition of the conditions takes place in the model section of the experiment specification so that the defined alias can be used in the hypothesis declaration. To address (2), hypotheses are written as an instantiation of a mechanism. The next concern (3) is addressed by including a declaration of the model path in the model section of the DSL. And finally, (4) is addressed by associating the acceptance definition with a number of batches to be run before building a verification model. In its current state, the grammar is ambiguous by definition, meaning there can be more than one leftmost derivation of the sentence "". This implies that invalid sentences could potentially be produced. This problem could be mitigated by adding limits to the number of times a non-terminal can be repeated. However, since the BNF of the grammar is used only to evaluate the usability of the language, its ambiguity can be addressed in the next step (language implementation). Once the grammar was defined, the sentences that could be produced by it could be enumerated. By means of validation, we can check if all of the requirements are met in the producible sentences.

After rigorous evaluation, the grammar was formally completed and presented for the next step, implementation. The complete listing of the BNF grammar can be found in Appendix A.

6.2 Language Implementation

In this section, the steps taken to change the DSL from a collection of words to a functioning language will be evaluated. First we will examine how the language was converted from BNF to Xtext[37], a language engineering Java extension. Next, we will evaluate the reference model and analyze its usability. Afterwards, we will examine how the reference model and the reference implementation use templates and transformations to close the loop and enable the language to automatically test the defined hypotheses after the source code is saved.

6.2.1 Xtext

After the DSL was defined as a BNF grammar, it was implemented in the language engineering framework, Xtext. Readers are reminded that FASE is implemented as a module to an experimentation language: the GHE framework. Before incorporation of FASE, the GHE framework was lacking any support for hypothesis testing, and mainly served as an experiment specification language. The FASE module was incorporated into the language by adding a new non-terminal, representing a hypothesis section into the language entry point. The implementation of the grammar from BNF to Xtext was straightforward because they operate on the same terminal/non-terminal philosophy.

As a simple example, the following implementation of the temporal pattern grammar in xtext, and the BNF version afterwards, illustrates the similarities. Here, there is just an introduction of variables such as l2, exp, op, etc. and inclusion of repeatability (* and ?).

```
TemporalPattern:
        Condition ((l2 += Links)?)* ((exp += Expression)?)*
        ((op1 += Temporal | op2 += Logical)?)* (l3 = Links)?
;

<TemporalPattern> ::= <Condition> <Links> <Expression>
```

```
<Operator> <Links>
```

The full listing of the Xtext grammar can be found on the project Github repository [38].

6.2.2 Reference Model

Following the initial implementation of the grammar in Xtext, the reference model was able to be developed using the Eclipse editor provided by the Xtext Eclipse plugin. The reference model includes syntax highlighting for the keywords from the grammar, and is structured properly into the sections of model, goal, hypothesis, and experiment. If any syntax is violated, the DSL will not generate new code. This is an important quality, because the DSL is the mechanism that ensures that a temporal property adheres to a specification pattern. If this quality did not hold, the reference implementation would not be able to interpret the property correctly.

6.2.3 Reference Implementation

The reference implementation provides the classes necessary to carry out all of the functions of the FASE language. The implementation is structured such that all the necessary translations from DSL to code are as simple as generating an API call for each statement in the DSL specification.

6.3 Model Verification

The focus of the FASE framework is formal verification of simulation models. The groundwork leading up to this goal has been evaluated thus far. Now, the actual process of model checking needs to be evaluated.

6.3.1 Verification Model Construction

The first step in constructing the verification model is the development of a Discrete Time Markov Chain (DTMC). The DTMC is built by taking each line of data from each experiment batch and evaluating whether it is a new state or a previously visited state. Each time a state is visited, its probability of being chosen in a branch increases. If a state has only one transition, that transition will have a probability of 1.0. Once the DTMC is constructed, the task remains of converting it into a model that is compatible with the model checker. In this case, we had to construct a model in the PRISM language by converting each state in the Markov chain into a state in the language. We also take the transitions and the "to" states and define the next states in the language. Thus completing the model.

6.3.2 Model Checking

By this point, the model checker is prepared to construct the verification model and check it against the temporal logic formula that was created by the reference implementation. PRISM does this by building a Rabin automata from the LTL property and evaluating whether the DTMC's Bottom Strongly Connected Components(BSCCs) are reachable following the rules of the Rabin automata.

6.3.3 Results

The results of the model checking are formatted as the percentage of times the model did not invalidate the temporal properties. If a property was invalidated, the model checker tells which property was invalidated and at which state. The final step in this process is to accept or update the current coherence network based on the output of the model checker. In the event that an explanatory evidence is falsified, the simulation model will be asserted to be in error. At this point we can stop the process and make alterations to the models source code manually. On the other hand, if the evidence is confirmed to be sound in the simulation model, we can officially make a positive connection between the evidence and hypothesis.

Chapter 7
Conclusions

The FASE framework provides users with an approach to experiment analysis that is faster than manual methods, and increases the traceability that is needed to ensure experimental validity. The implication of this contribution is that a greater number of parallel experiments can be tested and validated in-silico than those that could be done by hand in the same amount of time. This is made possible by the use of model checking, a formal method of model verification, and Model-Driven software development practices, such as code generation and domain specific languages.

7.1 Summary

The inspiration for the FASE framework was sparked by the perception of opportunities to accelerate the scientific method with software models. The essential focus of this study has been to apply formal methods of software verification to simulation models in such a way to produce elements of scientific discovery. Adhering to the idioms of Model-Driven software development, specifically the process of transforming metamodel to grammar to implementation, allows this work to be easily reproduced and extended. The domain-specific language was shown to be an effective means to specify hypotheses in temporal logic-like syntax. Through the case study in Chapter 4, we found that the results of analyzing hypotheses with model checking are consistent with the expectation that hypotheses can be tested in-silico based on observed behavior. The use of a coherence network to explain the domain understanding provides a useful explanatory mechanism for evaluation of competing hypotheses.

7.2 Findings

This research has shown that automated formal software analysis can be a useful tool to aid in the process of scientific experimentation. It was found that a quantitative metric of hypothesis validity and explanatory mechanisms can be provided to guide the process of experimentation in a positive direction. These findings insist that further avenues of automation can continue to increase the speed of scientific discovery.

The proposed framework does, however, have limitations in regards to aspect-oriented data recording and specification patterns. The strength and appeal of the aspect-oriented approach is the straightforward detection of event occurrences, such as method calls, but it struggles to obtain consistent results when attempting to record local variable changes. It currently records variables by calling accessor methods at intervals defined in the DSL, but the spacing of these intervals may cause the system to miss important changes in state. To address this issue, a more sophisticated model analysis engine may be needed to identify insertion points for pointcuts around critical variable changes. In regards to specification patterns, the framework is limited to only the patterns that have been discovered. There is currently no way to define an LTL property in the DSL if the property cannot be expressed as a specification pattern. The implication of this limitation is speculatively minimal, as the purpose of specification patterns is to apply expert knowledge in this tricky domain, and the patterns are generally applicable.

7.3 Future Research

7.3.1 Point of Failure Identification

In the current system, when a temporal property is violated, the model checker returns with a counterexample and a sequence of transitions that lead to that counterexample. With additional work, this system could trace its steps even further and report the mechanism that caused the hypothesis to be invalidated. This would be a useful tool when attempting to refine the model if wet-lab results are conflicting with in-silico experiment results.

7.3.2 Automated Model Evolution

In relation to the prior topic, if the mechanism that is in error is able to be identified, it may be able to be corrected autonomously by making minor alterations to the simulation source code. This goal would require a more elaborate characterization of mechanisms so that they could be analyzed and revised based on irregularities with the expected properties.

7.3.3 Bayesian/Coherence Synergy

A Bayesian network is a probabilistic graphical model that represents a set of random variables and their conditional dependencies via a directed acyclic graph. Each node in the graph represents a random variable, each of which has a set of possible states it can be in (or values it can take). In addition, attached to each node is a probability distribution. The arcs indicate a dependency between two nodes. If there is a dependency, then the probability distribution of the child node is a conditional probability that depends on the probability of the parent node.

While the coherence network can function as a long-term memory repository of hypotheses and their relationships, Bayesian networks can be used as a short-term memory interface with the experiments performed. Bayesian networks can perform a variety of tasks, but here we will only discuss potential applications that demonstrate their use as interfaces between coherence networks and experiments.

An important component of coherence networks are the evidence nodes. Evidence nodes, in an experimental context, represent observations obtained from an experiment. Often, these evidence nodes are determined by the experimenter as the result of interpreting the experimental data. We propose using a Bayesian network to infer these statements from the data.

7.3.4 Coherence Network Analyst Agent

As part of a longer-term project, we envision an intelligent agent designed to analyze the dependencies of the coherence network and propose new experiments to maximize information gain. This agent would be able to perform optimizations between the experiment and hypothesis space in order to predict the most valuable parametrization of experiments and hypotheses that would lead to faster discovery.

References

[1] J. B. Passioura, “Simulation models: Science, snake oil, education, or engineering?”, *Agronomy Journal*, vol. 88, no. 5, pp. 690–694, 1996.

[2] R. G. Sargent, “Validation and verification of simulation models”, in *Proceedings of the 36th conference on Winter simulation*, Winter Simulation Conference, 2004, pp. 17–28.

[3] M. A. W. W. R. Hunt, *Applied Groundwater Modeling, Simulation of Flow and Advective Transport*. Academic Press, 2015, ISBN: 9780120581030.

[4] H. Simon, T. Zacharia, R. Stevens, *et al.*, “Modeling and simulation at the exascale for energy and the environment”, *Department of Energy Technical Report*, 2007.

[5] V. G. Honavar, M. D. Hill, and K. Yelick, “Accelerating science: a computing research agenda”, 2016. arXiv: 1604.02006.

[6] J. Sprenger, “How do hypothesis tests provide scientific evidence? reconciling karl popper and thomas bayes”, 2011.

[7] S. Robinson, “Simulation model verification and validation: Increasing the users’ confidence”, in *Proceedings of the 29th conference on Winter simulation*, IEEE Computer Society, 1997, pp. 53–59.

[8] P. Savory and G. Mackulak, “The science of simulation modeling”, 1994.

[9] P. A. Moskovitz, “Randomness does not occur in nature : philosophical assumptions at the boundary of knowledge and certainty”, *Synthesis: A Journal of Science, Technology, Ethics, and Policy*, vol. 4, pp. 10–11, 2013.

[10] S. Beydeda, M. Book, V. Gruhn, *et al.*, *Model-driven software development*. Springer, 2005, vol. 15.

[11] J. Bettin, "Model-driven software development", *MDA Journal*, vol. 1, 2004.

[12] S. W. Liddle, "Model-driven software development", in *Handbook of Conceptual Modeling*, Springer, 2011, pp. 17–54.

[13] M. Voelter, C. Salzmann, and M. Kircher, "Model driven software development in the context of embedded component infrastructures", *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*, vol. 3778, pp. 143–163, 2005, ISSN: 0302-9743. DOI: 10.1007/11591962_8.

[14] R. C. Gronback, *Eclipse modeling project: A domain-specific language (DSL) toolkit.* Pearson Education, 2009.

[15] T. S. M. Volter, *Model-Driven Software Development.* John Wiley & Sons Ltd., 2006, ISBN: 978-0-470-02570-3.

[16] S. Chakladar, "A model driven engineering framework for simulation experiment management", 2016.

[17] L. Yilmaz, K. Doud, and S. Chakladar, "The goal-hypothesis-experiment framework: a generative cognitive domain architecture for simulation experiment management", *Winter Simulation Conference*, 2016.

[18] K. W. K. Lee, "An introduction to aspect-oriented programming", *COMP610E: Course of Software Development of E-Business Applications (Spring 2002), Hong Kong University of Science and Technology*, 2002.

[19] B. Griswold, E. Hilsdale, J. Hugunin, W. Isberg, G. Kiczales, and M. Kersten, "Aspect-oriented programming with aspectj", *AspectJ. org, Xerox PARC*, 2001.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, and C. Lopes, "Aspect-oriented programming aspect-oriented programming", *European Conference on Object-Oriented Programming*, 1997.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj", in *ECOOP 2001 - Object-Oriented Programming*, 4, vol. 2072, 2001, pp. 327–354, ISBN: 3540422064. DOI: `10.1007/3-540-45337-7_18`.

[22] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking.* Edmund M. Clarke, Jr., Orna Grumberg, and Lucent Technologies, 1999, ISBN: 0-262-03270-8.

[23] B. Blaskovic, "Model checking executable specification for reactive components", pp. 107–113, 2012.

[24] M. Kwiatkowska, G. Norman, and D. Parker, "Advances and challenges of probabilistic model checking", in *Communication, Control, and Computing (Allerton), 2010 48th Annual Allerton Conference on*, IEEE, 2010, pp. 1691–1698.

[25] G. Norman and D. Parker, "Quantitative verification: Formal guarantees for timeliness, reliability and performance", The London Mathematical Society and the Smith Institute, Tech. Rep., 2014.

[26] K. L. McMillan, "Symbolic model checking", in *Symbolic Model Checking*, Springer, 1993, pp. 25–60.

[27] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic model checking for systems biology", 2010.

[28] ——, "Prism 4.0: Verification of probabilistic real-time systems", in *International Conference on Computer Aided Verification*, Springer, 2011, pp. 585–591.

[29] ——, "Stochastic model checking", in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, Springer, 2007, pp. 220–270.

[30] B. Sericola, "Discrete-time markov chains", *Markov Chains*, pp. 1–87,

[31] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification", *Information Sciences*, vol. 2,

[32] J. Corbett, M. Dwyer, and J. Hatcliff, “A language framework for expressing checkage properties of dynamic software”, *Proceedings of the SPIN Workshop*, 2000.

[33] P. Thagard, “Explanatory coherence”, *Behavioral and brain sciences*, vol. 12, no. 03, pp. 435–467, 1989.

[34] ——, “Probabilistic networks and explanatory coherence.”, in *Automated Abduction: Inference to the best explanation*, AAAI Press. Menlo Park, 1997.

[35] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages”, *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[36] B. K. Petersen, G. E. Ropella, and C. A. Hunt, “Virtual experiments enable exploring and challenging explanatory mechanisms of immune-mediated p450 down-regulation”, *PloS one*, vol. 11, no. 5, e0155855, 2016.

[37] M. Eysholdt and H. Behrens, “Xtext: Implement your language faster than the quick and dirty way”, in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ACM, 2010, pp. 307–309.

[38] K. Doud, *Fase framework*, `https://github.com/Kyle-Doud/FASE-Framework`, 2017.

Appendices

Appendix A

BNF Grammar

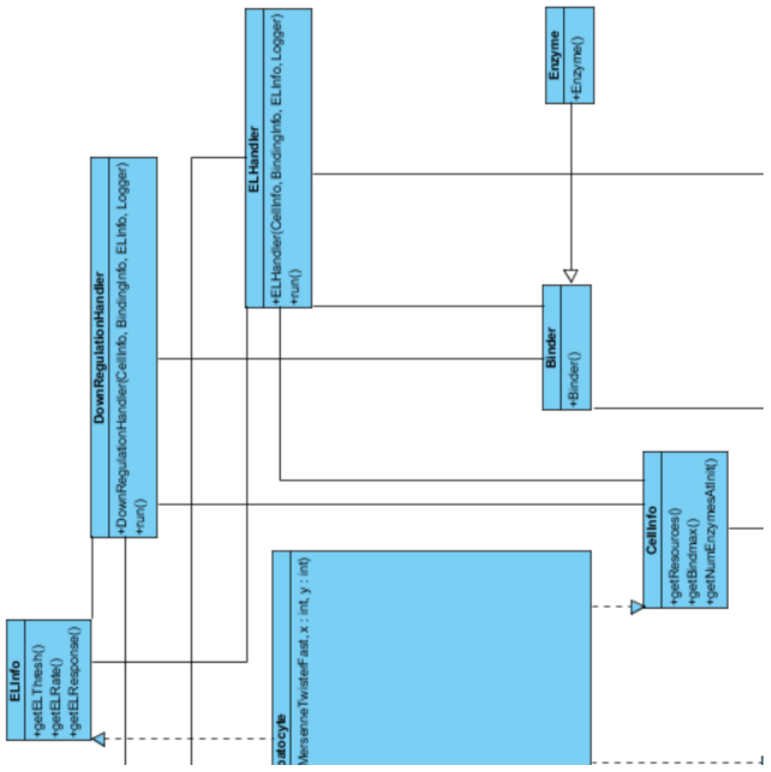
```
<Model> ::= ExperimentOntology
<ExperimentOntology> ::= ModelSection | Goals | Hypothesis |
Experiment
<ModelSection> ::=  model <id> {<Mechanism> <EventDescriptor>
<Factor>}
<Mechanism> ::= mechanism <id> = <Reaction> <GuardCondition> –>
<Reaction>
<Reaction> ::= <id> + <id>
<GuardCondition> ::= [ <id> <LinkOperator> <id> ]
<EventDescriptor> ::= event <id> = <STRING>
<Factor> ::= parameter <id> = <VariableType> <id> with values
{<Values>} properties {<properties>}
<Goals> ::= goal { object of study : <STRING> purpose :
<STRING> focus : <STRING> view point : <STRING> context :
<STRING>}
<VariableType> ::= QUALITATIVE | QUANTITATIVE | CONTINUOUS |
DISCRETE | BINARY | NONBINARY
<Values> ::= <id> <XExpression> <rangeValue> <factorLevelValue>
<properties> ::= <id> : <Values>
<rangeValue> ::= INT <dot> <OptionalInt>
<factorLevelValue> ::= <rangeValue> | , <rangeValue>
<dot> ::= "." | ""
```

```
<OptionalInt> ::= INT | ""
<Hypotheses> ::= hypotheses { mechanistic
hypotheses{<MechHypotheses>} evidence{<Evidence>} coherence
model{<CoherenceLink>} relational hypotheses
{<RelationalQuery>}}
<CoherenceLink> ::= <Coherence> (<id>)(<id>)
<MechHypotheses> ::= <id> : <TemporalPattern> <id>
<Evidence> ::= <id> : <TemporalPattern> activation weight :
<rangeValue>
<TemporalPattern> ::= <Sample> <Links> <Expression> <Operator>
<Links>
<Sample> ::= <Condition> | <Event>
<Condition> ::= <ComplexID> <LinkOperators> <Expression>
<rangeValue> <Condition>
<Event> ::= <Dispersed> | <Simultaneous>
<Dispersed> ::= <id> <Logical> <id> | <id>
<Operator> ::= <Temporal> | <Logical>
<Simultaneous> ::= [<id> <Logical> <id>] | [<id>]
<LinkOperators> ::= '+'|'-'|'*'|'/'|'%'|' = '|' ==
'|'&&'|'||'|'<'|'<='|'>'|'>='|'!'|'!='
<Expression> ::= true | false
<Links> ::= is | occurs | to | in
<Temporal> ::= precedes | between | eventually | always |
before | after | until | never | leads | absent | exists
<Logical> ::= and | or | not
<Coherence> ::= EXPLAIN | ANALOGOUS | DATA | CONTRADICT
<RelationalQuery> ::= <Query1> | <Query2> | <Query3> | <Query4>
```

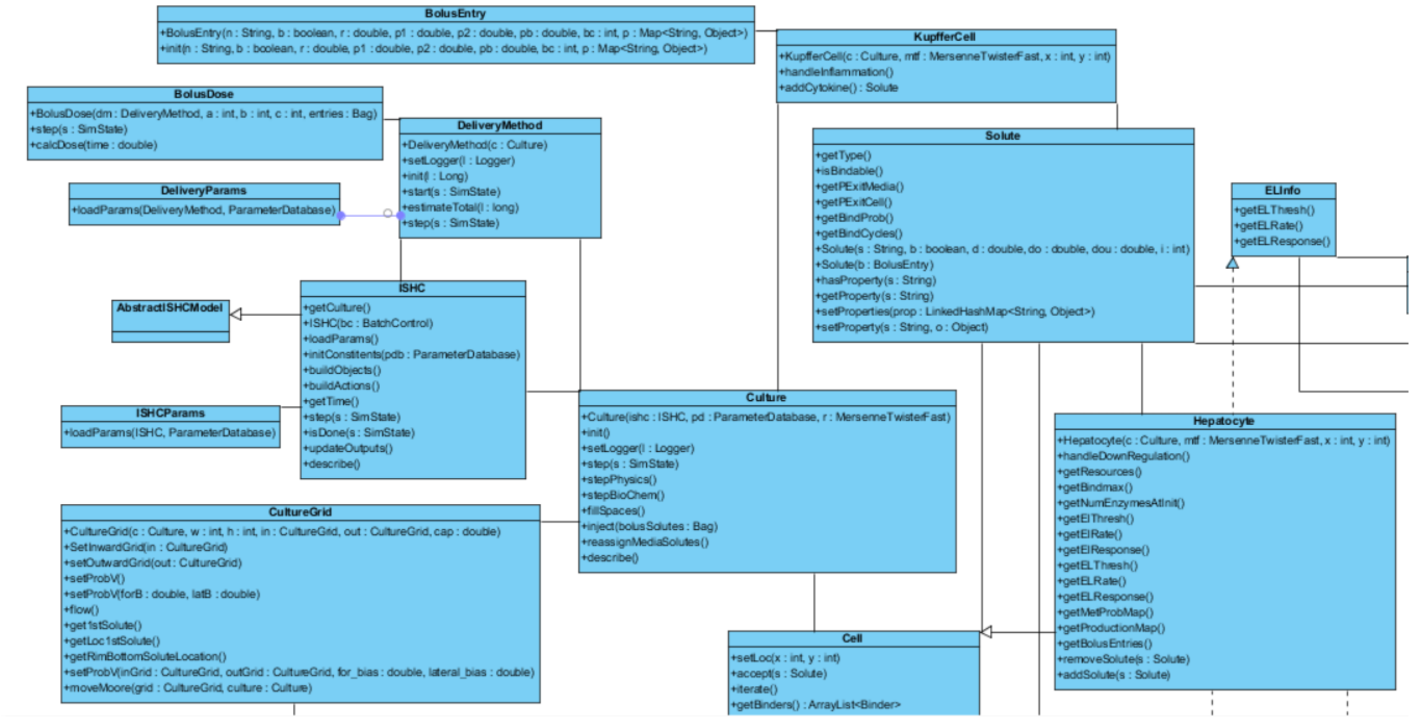
```
| <Query5>
<Query1> ::= if <id> <id> is <rangeValue> <Action> then <id> is
<Response>
<Action> ::= added | removed | in the range <rangeValue> to
<rangeValue>
<Response> ::= <rangeValue> | in the range <rangeValue> to
<rangeValue>
<Query2> ::= compare <Operand> and <Operand>
<Operand> ::= <Function> | <id>
<Function> ::= MIN | MAX | EXP | INVERSE | SIN | COS | TAN |
FACTORIAL | LOG
<Query3> ::= if <QueryCondition> then <QueryResponse> where
<Levels> for <id> <id> <id> is in the range <rangeValue> to
<rangeValue>
<QueryCondition ::= <id> <id> is <Level>
<Level> ::= at level <Levels> <rangeValue> <OptionalAnd>
<Level> | ""
<QueryResponse> ::= <id> is <Level>
<Changes> ::= CHANGED | INCREASED | DECREASED | CONSTANT
<Levels> ::= HIGH | MEDIUM | LOW
<Experiment> ::= experiment <id> { design <Design> performance
measure is <PerformanceMeasure}
<Design> ::= {designType <DesignType> constraints <XExpression>
<Iteration> variables <Variables>}
<DesignType> ::= FULLFACTORIAL | FRACTIONALFACTORIAL | OTHERS |
""
<Variables> ::= {<IndependentVariables> <ControlVariables>
<DependentVariables>}
<IndependentVariables> ::= independent variables
{<FactorLevels>}
<ControlVariables> ::= control variables {<id> : type
<VariableType> with values {<Values>}}
<DependentVariables> ::= dependent variables {<Response>}
<FactorLevels> ::= <id> are at levesl : <rangeValue> <Levels>
 where <Levels> is in the range <rangeValue> to <rangeValue>
<Response> ::= <id> : type <ResponseType>
<ResponseType ::= SIMPLE | COMPOSITE
<Iteration> ::= number of iterations : INT
<PerformanceMeasure> ::= {<id> = <rangeValue> +- <rangeValue>}
<OptionalAnd> ::= and | ""
<OptionalTo> ::= to | ""
<STRING> ::= "..."
```

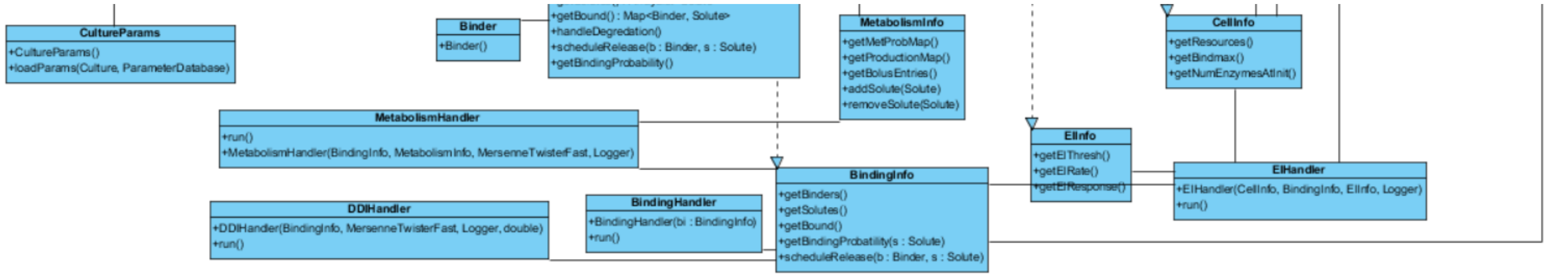
Appendix B

ISHC Metamodel



ELInfo
+getELThresh()
+getELRate()
+getELResponse()
DownRegulationHandler
+DownRegulationHandler(CellInfo, BindingInfo, ELInfo, Logger)
+run()
ELHandler
+ELHandler(CellInfo, BindingInfo, ELInfo, Logger)
+run()
Enzyme
+Enzyme()
Binder
+Binder()
CellInfo
+getResources()
+getBindmax()
+getNumEnzymesAtInit()
atocyte
MersenneTwisterFast, x : int, y : int)


BolusEntry
+BolusEntry(n : String, b : boolean, r : double, p1 : double, p2 : double, pb : double, bc : int, p : Map<String, Object>)
+init(n : String, b : boolean, r : double, p1 : double, p2 : double, pb : double, bc : int, p : Map<String, Object>)
KupfferCell
+KupfferCell(c : Culture, mtf : MersenneTwisterFast, x : int, y : int)
+handleInflammation()
+addCytokine() : Solute
BolusDose
+BolusDose(dm : DeliveryMethod, a : int, b : int, c : int, entries : Bag)
+step(s : SimState)
+calcDose(time : double)
DeliveryMethod
+DeliveryMethod(c : Culture)
+setLogger(l : Logger)
+init(l : Long)
+start(s : SimState)
+estimateTotal(l : long)
+step(s : SimState)
DeliveryParams
+loadParams(DeliveryMethod, ParameterDatabase)
Solute
+getType()
+isBindable()
+getPExitMedia()
+getPExitCell()
+getBindProb()
+getBindCycles()
+Solute(s : String, b : boolean, d : double, do : double, dou : double, i : int)
+Solute(b : BolusEntry)
+hasProperty(s : String)
+getProperty(s : String)
+setProperties(prop : LinkedHashMap<String, Object>)
+setProperty(s : String, o : Object)
ELInfo
+getELThresh()
+getELRate()
+getELResponse()
ISHC
+getCulture()
+ISHC(bc : BatchControl)
+loadParams()
+initConstitents(pdb : ParameterDatabase)
+buildObjects()
+buildActions()
+getTime()
+step(s : SimState)
+isDone(s : SimState)
+updateOutputs()
+describe()
AbstractISHCModel
ISHCParams
+loadParams(ISHC, ParameterDatabase)
Culture
+Culture(ishc : ISHC, pd : ParameterDatabase, r : MersenneTwisterFast)
+init()
+setLogger(l : Logger)
+step(s : SimState)
+stepPhysics()
+stepBioChem()
+fillSpaces()
+inject(bolusSolutes : Bag)
+reassignMediaSolutes()
+describe()
Hepatocyte
+Hepatocyte(c : Culture, mtf : MersenneTwisterFast, x : int, y : int)
+handleDownRegulation()
+getResources()
+getBindmax()
+getNumEnzymesAtInit()
+getElThresh()
+getElRate()
+getElResponse()
+getELThresh()
+getELRate()
+getELResponse()
+getMetProbMap()
+getProductionMap()
+getBolusEntries()
+removeSolute(s : Solute)
+addSolute(s : Solute)
CultureGrid
+CultureGrid(c : Culture, w : int, h : int, in : CultureGrid, out : CultureGrid, cap : double)
+SetInwardGrid(in : CultureGrid)
+setOutwardGrid(out : CultureGrid)
+setProbV()
+setProbV(forB : double, latB : double)
+flow()
+get1stSolute()
+getLoc1stSolute()
+getRimBottomSoluteLocation()
+setProbV(inGrid : CultureGrid, outGrid : CultureGrid, for_bias : double, lateral_bias : double)
+moveMoore(grid : CultureGrid, culture : Culture)
Cell
+setLoc(x : int, y : int)
+accept(s : Solute)
+iterate()
+getBinders() : ArrayList<Binder>


CultureParams
+CultureParams()
+loadParams(Culture, ParameterDatabase)
Binder
+Binder()
+getBound() : Map<Binder, Solute>
+handleDegredation()
+scheduleRelease(b : Binder, s : Solute)
+getBindingProbability()
MetabolismInfo
+getMetProbMap()
+getProductionMap()
+getBolusEntries()
+addSolute(Solute)
+removeSolute(Solute)
CellInfo
+getResources()
+getBindmax()
+getNumEnzymesAtInit()
MetabolismHandler
+run()
+MetabolismHandler(BindingInfo, MetabolismInfo, MersenneTwisterFast, Logger)
ElInfo
+getElThresh()
+getElRate()
+getElResponse()
ElHandler
+ElHandler(CellInfo, BindingInfo, ElInfo, Logger)
+run()
BindingInfo
+getBinders()
+getSolutes()
+getBound()
+getBindingProbatility(s : Solute)
+scheduleRelease(b : Binder, s : Solute)
DDIHandler
+DDIHandler(BindingInfo, MersenneTwisterFast, Logger, double)
+run()
BindingHandler
+BindingHandler(bi : BindingInfo)
+run()