Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

A FRAMEWORK FOR FORMAL AUTOMATED ANALYSIS OF SIMULATION EXPERIMENTS USING PROBABILISTIC MODEL CHECKING

Kyle Doud
Levent Yilmaz

Department of Computer Science and Software Engineering
Auburn University
3101 Shelby Center
Auburn, AL 36849, USA

ABSTRACT

Simulation experiments contribute to scientific discovery due to the degree and extent of reproducibility that simulation systems provide. On the other hand, domain scientists may lack expertise in simulation programming and the use of effective methods for instrumenting, evaluating, and comparing models. By utilizing formal automated verification methods, we aim to improve the process of evaluating model assumptions against evidence, and to facilitate selection of new hypotheses to maximize information gain while reducing information processing requirements. To this end, to evaluate the results of a simulation experiment against expected regularities, a probabilistic model checking system is coupled with a Domain-Specific Language that expresses abstract finite state verification properties. These specification patterns are evaluated against the run-time Discrete-Time-Markov Chain model abstracted from the data obtained through aspect-driven automated instrumentation.

1 INTRODUCTION

As simulation software development practices are becoming prevalent in various scientific disciplines (Greenwald 2004, Sargent 2004), it is 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 are used in a diverse array of fields, ranging from ground-water analysis (Hunt, Anderson, and Woessner 2015) to biomedical systems, socioeconomic forecast, and astrophysics (Simon, Zacharia, and Stevens 2007), 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 and evaluating the results of 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 manually 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. In (Honavar, Hill, and Yelick 2016), computation is viewed as a powerful formal framework and exploratory apparatus for the conduct of science. The claim is supported by the observation that computation, mathematics, and science are often used ubiquitously to provide a structured way of answering questions. For the purposes of this paper, 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 explain observations. Once a hypothesis is rigorously tested under varying conditions, it can be upgraded to an 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 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" (Sprenger 2011). 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 hypothesis' 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 formulation 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 (Robinson 1997). 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 pointed out in (Savory and Mackulak 1994) 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.

We propose to address the above challenges by providing a framework for hypothesis testing that is simulation platform independent. This framework for **F**ormal **A**utomated Analysis of **S**imulation **E**xperiments (FASE) provides a domain specific language for hypothesis specification and automated model checking to evaluate hypotheses using a statistical model checker. By incorporating the results of model checking into learning networks, revealing experiments can be developed at a faster rate to increase knowledge gain.

2 BACKGROUND

In this section, we overview the state of the art in three major areas that contribute to the development of the formal analysis framework.

2.1 Model-Driven Software Development

Model-Driven Engineering (MDE) (Beydeda, Book, Gruhn, et al. 2005, Bettin 2004, Liddle 2011) is a strategy of software development that focuses on building software that is correct-by-construction as opposed to construct-by-correction. MDE 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. 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 languages to describe these models are called domain-specific languages (DSL) (Gronback 2009). 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. Finally, the semantics of the DSL ensure that the model's meaning is well-defined for the purpose of facilitating transformations (Voelter, Salzmann, and Kircher 2005).

2.2 Automated Instrumentation

Analysis and evaluation of simulations require seamless observability and controllability of simulation software. For observability, instrumentation of simulation experiments can be supported by Aspect-Oriented Programming (AOP) techniques (Lee 2002). Instrumentation refers to observing data used to indicate, measure, and record quantities observed in the course of an experiment. To facilitate instrumentation, AspectJ (Griswold 2001) provides a means of recording variable values during simulation execution by AOP methods. Therefore, AOP 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” (Lee 2002). An example of this is data logging code, where data is logged in-line with model logic. These concerns are addressed by separating *components* and *aspects*, then weaving them together in a manner that untangles the problem (Kiczales et al. 1997).

2.3 Model Checking

Model checking is a formal verification technique for finite-state concurrent systems. Using model checking as a tool for formal verification is now a well-established methodology (Clarke, Grumberg, and Peled 1999). The process of model checking involves defining the model and then checking 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 (Blaskovic 2012).

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, non-deterministic and real-time characteristics” (Kwiatkowska, Norman, and Parker 2010a). *Probabilistic model checking* (Norman and Parker 2014) is defined as a method for modeling and analysis of systems with stochastic behavior. It is a variant of *model checking* (McMillan 1993), 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 (Kwiatkowska, Norman, and Parker 2010b).

In this study, simulation system behavior is modeled as a Discrete-Time Markov Chain (DTMC). 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.

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 (Dwyer, Avrunin, and Corbett 1998). 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 (Corbett, Dwyer, and Hatcliff 2000).

3 THE CONCEPTUAL BASIS FOR FASE

In this section, we overview the process model of the formal analysis strategy and illustrate its application on a simple, yet informative example.

3.1 The Goal-Hypothesis-Experiment Framework

In this section, we delineate the overarching infrastructure of FASE, which is the Goal-Hypothesis-Experiment (GHE) framework (Chakladar 2016, Yilmaz, Chakladar, and Doud 2016). 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 MDE 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) (Yilmaz, Chakladar, and Doud 2016). 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 within each space. 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.

In the context of the GHE framework, the Explanatory Coherence Theory (Thagard 1989, Thagard 1997) offers a set of principles and methods for studying the degree of coherence existing between a set of hypotheses based on the kinds of formal analyses presented in this paper. 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. Therefore, 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.

3.2 FASE Process Model

Figure 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. In the first 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 (Chakladar 2016, Yilmaz, Chakladar, and Doud 2016). Once the specification is saved, the transformations take place, and the next activity is executed.

For model transformation, 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. To convert hypotheses to Linear Temporal Logic (LTL), the hypotheses from the specification are 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 the time when the PRISM model is built.

To facilitate code weaving with AspectJ, the system generates pointcuts and advice to record the variables from the hypotheses. Once the data recording elements are generated, the execution of experiments starts to collect the data into a table formatted file. Here, each row of the table represents a change in one of the variables' value. The data from the experiments are then used to build a Markov Chain. Transitions in the chain are determined upon the ranges defined in the hypotheses. Repeat states increase the likelihood, while not adding the same state twice. Next, 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.

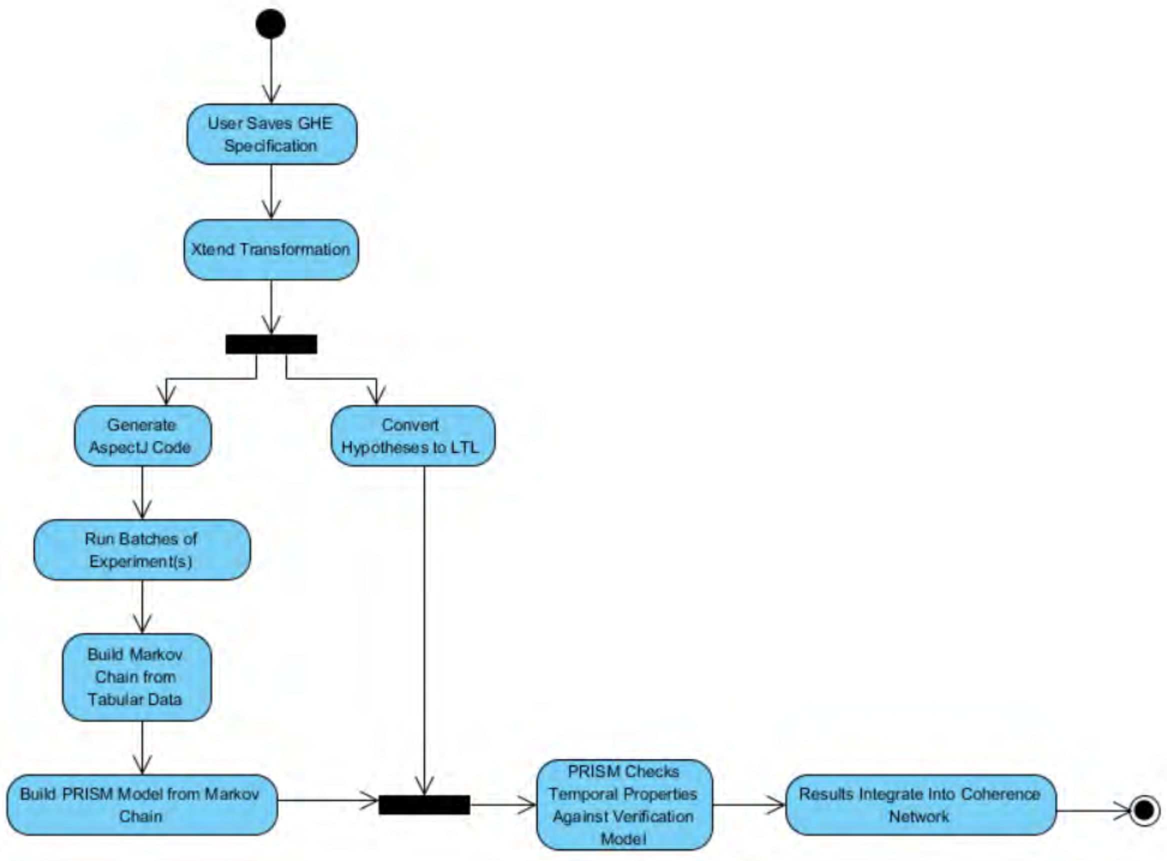
User Saves GHE Specification
Xtend Transformation
Generate AspectJ Code
Convert Hypotheses to LTL
Run Batches of Experiment(s)
Build Markov Chain from Tabular Data
Build PRISM Model from Markov Chain
PRISM Checks Temporal Properties Against Verification Model
Results Integrate Into Coherence Network

Figure 1: FASE system activity diagram.

After the PRISM model is built, 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 kronecker product of the omega automata and the DTMC to evaluate the reachability of the strongly connected components. 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.

3.3 Example Use Case

To illustrate the process, we present a simple use case that involves determining if a biological phenomenon is satisfied by a simulation model. An example specification property is given as:

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

This property will affect which variables are recorded during the simulation run. Only the information needed to verify these properties will be recorded. These include any changes in the inflammatoryAgent, inflammatorythreshold, or cytokine variables. The LTL formula generated by the specification is as follows:

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

This property is now in the form of a temporal logic formula. It is submitted to the model checker in the last step. 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 1 is a simplified table of values for the purposes of illustration.

Table 1: Recorded simulation data values.

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

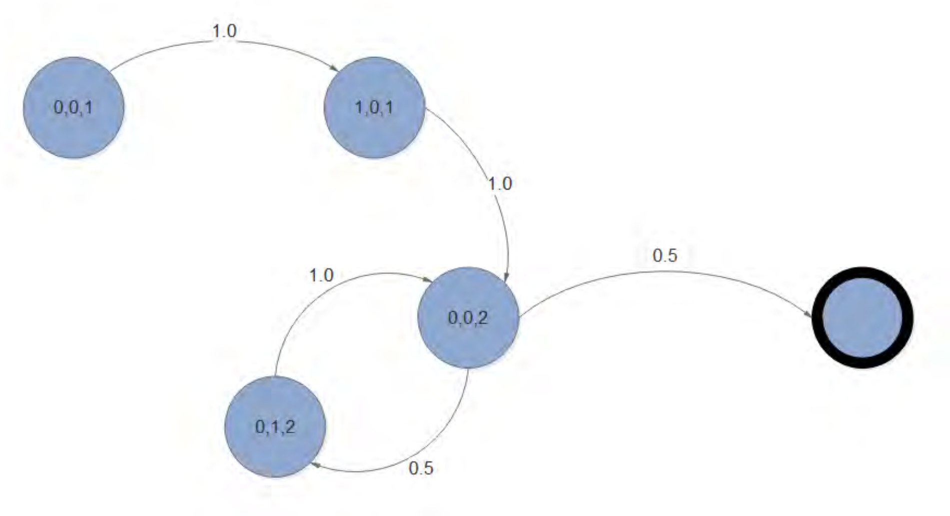

Figure 2: DTMC representation of simulation data.

Figure 2 shows the result of analyzing the above data set and creating a Discrete-Time Markov Chain. With the Markov Chain constructed, the PRISM model is 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 2 is as follows:

```
 [] inflammatoryAgent = 0 & 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 (because the property is invalid for all paths of the DTMC) and a state trace from 0,0,1 to 1,0,1.

4 FASE IMPLEMENTATION

In this section, we will explain the development and design of the FASE language. Readers are reminded that the FASE framework is an extension of the Goal-Hypothesis-Experiment framework (Chakladar 2016, Yilmaz, Chakladar, and Doud 2016) that addresses the "hypothesis" section of the framework's concerns. The development of the FASE language starts with the creation of a metamodel of the concepts needed to support property specification, acceptance definitions, condition declarations, and model connection. In 4.1, the structure of the GHE framework is presented, and the parts addressed by FASE are highlighted. For more information on the experiment management terms in Figure 3, see (Yilmaz, Chakladar, and Doud 2016). The terms for hypothesis testing are discussed in detail in section 4.2.

4.1 Structure

As shown in Figure 3, 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.

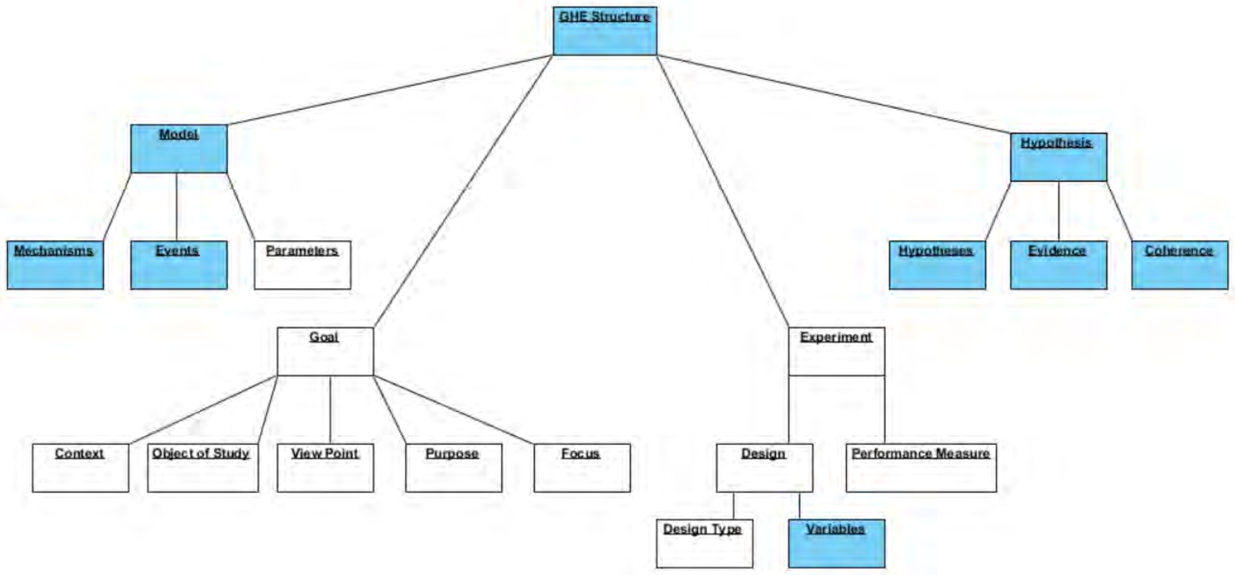

Figure 3: The goal-hypothesis-experiment language structure. Terms specific to hypothesis testing are colored, and experiment management terms are uncolored.

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. In the *Hypothesis* 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.

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 Syntax

The syntax of the FASE language is developed by first modeling the words needed and their relationships in UML. Figure 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.

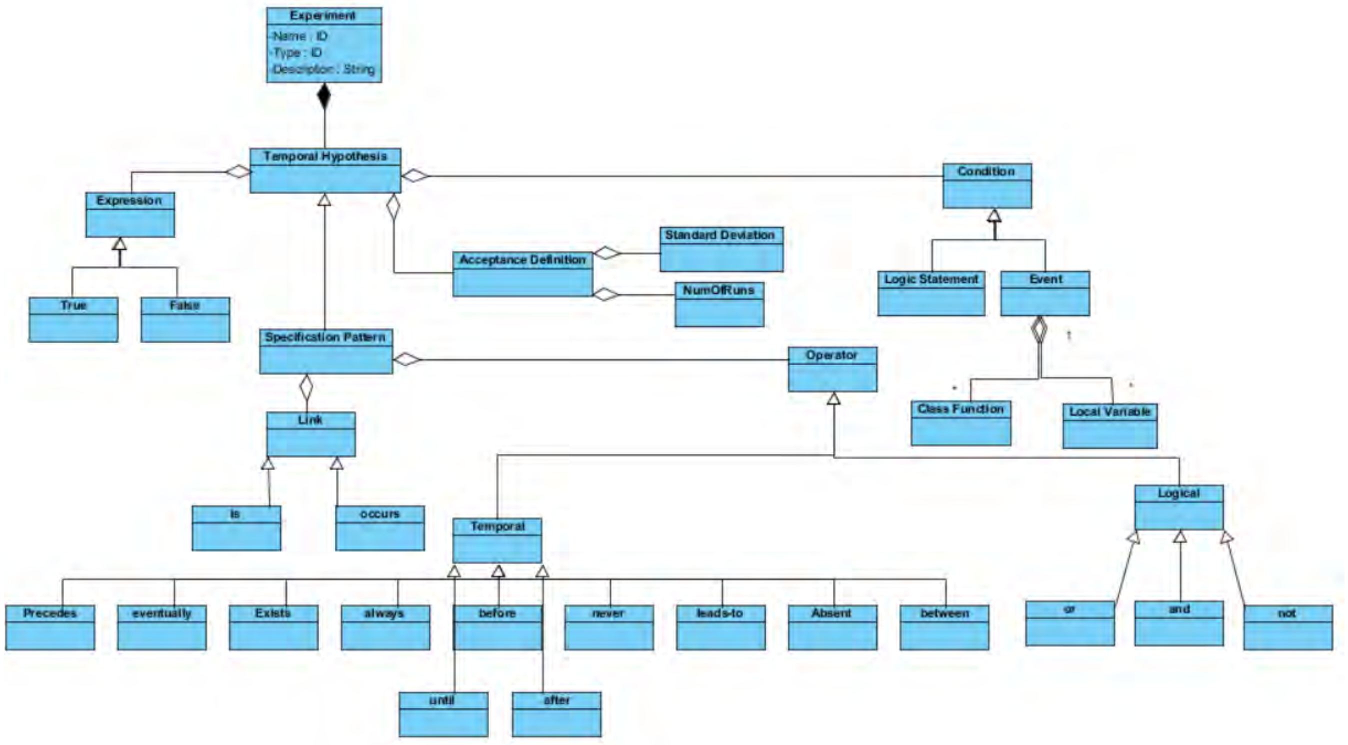

Figure 4: FASE DSL metamodel.

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.*

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 for the reference model.

```
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 and Reference Implementation

Transformations are defined using Xtend, a template-based model generation extension of Xtext. The Xtend templates are developed as a driver for the hypothesis testing framework. Grammar elements are extracted from the specification and substituted into the template. The reference implementation is the Java code that should execute as a result of running the DSL. It performs five major 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 components are described below.

The conversion process takes the hypothesis definition from the DSL as input 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. The strategy involves identifying what specification property the hypothesis belongs to and decomposes its parts, identifying conditions and events.

This *AspectJGenerator* 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

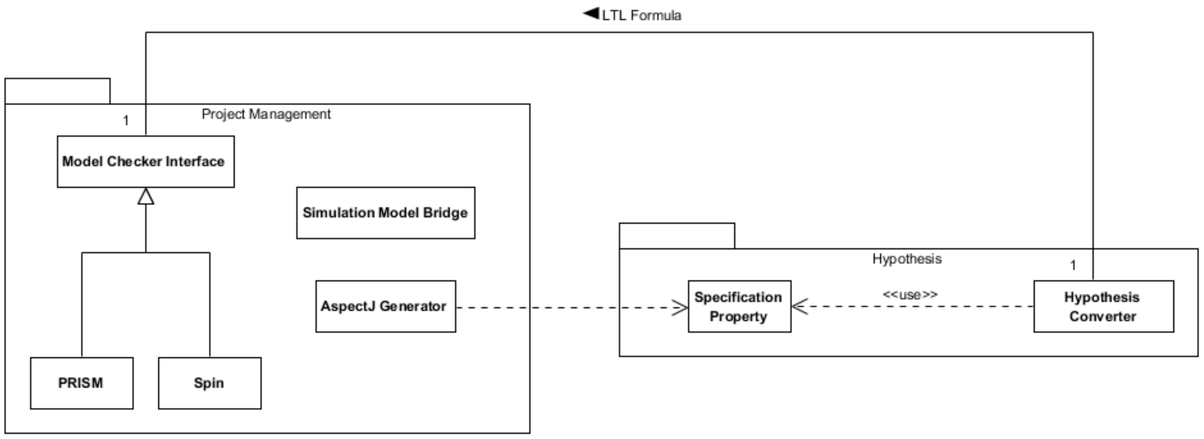

Figure 5: Hypothesis and project management metamodel.

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. The *SimulationModelInterface* 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. The *DataRecordManager* 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.This

MarkovChainBuilder constructs the verification model from the simulation data. It reads through the data file that the *DataRecordManager* 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. The *PrismInterface* 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.

5 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.

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. Improvements in instrumentation specification is needed to identify insertion points for pointcuts around critical variable changes. In regards to specification patterns, the framework is limited to only selected patterns, so arbitrary LTL properties cannot be defined in the DSL if the property cannot be expressed as a specification pattern.

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. 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

Bettin, J. 2004. “Model-Driven Software Development”. *MDA Journal* 1:1–4.

Beydeda, S., M. Book, V. Gruhn et al. 2005. *Model-Driven Software Development*, Volume 15. Springer.

Blaskovic, B. 2012. “Model Checking Executable Specification for Reactive Components”. *International Conference on Advances in System Testing and Validation Lifecycle* 4:107–113.

Chakladar, S. 2016. “A Model Driven Engineering Framework for Simulation Experiment Management”. Master’s thesis, Auburn University, Auburn, AL.

Clarke, E. M., O. Grumberg, and D. A. Peled. 1999. *Model Checking*. Edmund M. Clarke, Jr., Orna Grumberg, and Lucent Technologies.

Corbett, J., M. Dwyer, and J. Hatcliff. 2000. “A Language Framework for Expressing Checkable Properties of Dynamic Software”. *SPIN Model Checking and Software Verification* SPIN 2000:205–223.

Dwyer, M. B., G. S. Avrunin, and J. C. Corbett. 1998. “Property Specification Patterns for Finite-State Verification”. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, 7–15. ACM.

Greenwald, M. 2004. “Beyond Benchmarking – How Experiments and Simulations Can Work Together in Plasma Physics”. *Computer physics communications* 164 (1): 1–8.

Griswold, B. 2001. “Aspect-Oriented Programming with AspectJ”. *AspectJ.org, Xerox PARC*:1–101.

Gronback, R. C. 2009. *Eclipse Modeling Project: A Domain-Specific Language Toolkit*. Pearson Education.

Honavar, V. G., M. D. Hill, and K. Yelick. 2016. “Accelerating science: A Computing Research Agenda”. *arXiv preprint arXiv:1604.02006* 1:1–17.

Hunt, R., M. Anderson, and W. Woessner. 2015. *Applied Groundwater Modeling, Simulation of Flow and Advective Transport*. Academic Press.

Kiczales, G. et al. 1997. “Aspect-Oriented Programming”. *ECOOP’97Object-Oriented Programming* 1241:220–242.

Kwiatkowska, M., G. Norman, and D. Parker. 2010a. “Advances and Challenges of Probabilistic Model Checking”. In *Proceedings of the 2010 Annual Allerton Conference on Communication, Control, and Computing*, 1691–1698. IEEE.

Kwiatkowska, M., G. Norman, and D. Parker. 2010b. “Probabilistic Model Checking for Systems Biology”. Citeseer.

Lee, K. W. K. 2002. “An Introduction to Aspect-Oriented Programming”. *COMP610E: Course of Software Development of E-Business Applications (Spring 2002), Hong Kong University of Science and Technology*.

Liddle, S. W. 2011. “Model-Driven Software Development”. In *Handbook of Conceptual Modeling*, 17–54. Springer.

McMillan, K. L. 1993. “Symbolic Model Checking”. In *Symbolic Model Checking*, 25–60. Springer.

Norman, G., and D. Parker. 2014. “Quantitative Verification: Formal Guarantees for Timeliness, Reliability and Performance”. Technical report, The London Mathematical Society and the Smith Institute.

Robinson, S. 1997. "Simulation Model Verification and Validation: Increasing the Users' Confidence". In *Proceedings of the 1997 Winter Simulation Conference*, edited by S. Andradttir, 53–59: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Sargent, R. G. 2004. "Validation and Verification of Simulation Models". In *Proceedings of the 2004 Winter Simulation Conference*, edited by R. G. Ingalls and M. D. Rossetti, 17–28: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Savory, P., and G. Mackulak. 1994. "The Science of Simulation Modeling". *International Conference on Simulation in Engineering Education (ICSEE 94)* 26:115–119.

Simon, H., T. Zacharia, and R. Stevens. 2007. "Modeling and Simulation at the Exascale for Energy and the Environment: Report on the Advanced Scientific Computing Research Town Hall Meetings on Simulation and Modeling at the Exascale for Energy, Ecological Sustainability and Global Security (E3)". *Office of Science, The US Department of Energy, Tech. Rep*.

Sprenger, J. 2011. "How do Hypothesis Tests Provide Scientific Evidence? Reconciling Karl Popper and Thomas Bayes".

Thagard, P. 1989. "Explanatory Coherence". *Behavioral and Brain Sciences* 12 (03): 435–467.

Thagard, P. 1997. "Probabilistic Networks and Explanatory Coherence". In *Automated Abduction: Inference to the Best Explanation*. AAAI Press. Menlo Park.

Voelter, M., C. Salzmann, and M. Kircher. 2005. "Model-Driven Software Development in the Context of Embedded Component Infrastructures". *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends* 3778:143–163.

Yilmaz, L., S. Chakladar, and K. Doud. 2016. "The Goal-Hypothesis-Experiment Framework: A Generative Cognitive Domain Architecture for Simulation Experiment Management". In *Proceedings of the 2016 Winter Simulation Conference*, edited by P. Frazier, T. Roeder, R. Szechtman, and E. Zhou, 1001–1012: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

AUTHOR BIOGRAPHIES

KYLE DOUD is a Graduate Student of the Department of Computer Science and Software Engineering at Auburn University, Auburn, AL. His research interests are in model-driven engineering and formal methods. His e-mail address is krd0015@auburn.edu.

LEVENT YILMAZ is Professor of Computer Science and Software Engineering at Auburn University with a joint appointment in Industrial and Systems Engineering. He holds M.S. and Ph.D. degrees in Computer Science from Virginia Tech. His research interests are in agent-directed simulation, cognitive computing, and model-driven science and engineering for complex adaptive systems. He is the former Editor-in-Chief of Simulation: Transactions of the Society for Modeling and Simulation International and the founding organizer and general chair of the Agent-Directed Simulation Conference series. His email address is yilmaz@auburn.edu.