

# A Model-Driven Generative Domain Architecture for Simulation Experiment Lifecycle Management

February 19, 2018

Simulation experiments are extensively used to study the operation of a system by abstracting the complexities. Experiments conducted using the computational models are especially useful in advancing scientific knowledge. However, the current studies in these areas fail to effectively articulate all aspects of the scientific experiment process and their utility in evaluating specific questions about the model. The under-utilization of Design of Experiments (DOE), limited transparency in the collection and analysis of results, and ad-hoc adaptation of experiments and models continue to hamper reproducibility and hence cause a credibility gap. These challenges can cause incomplete or flawed analysis and wasted human and computer effort. This report introduces Model-Driven Experiment Management to make experimentation a seamless part of simulation development. Besides the explicit representation of experiment models, the integration will facilitate co-evolution of the models of simulation and experiment. The strategy leverages higher-order synergies between Model-Driven Engineering, Intelli-

gent Agent Technology, DOE, and probabilistic modeling to support development of a generative domain architecture and software system for model-driven experiment management. Experimental complexities are reduced by the inclusion of the Goal-Hypothesis-Experiment framework, which includes a domain-specific language for experiment management. Using this framework, a scientist needs to specify the goal of the experiment and give evidence and hypotheses to be tested against a simulation model. The experiment specification will be leveraged to design a family of experiments, which will be interpreted by experiment orchestration agents to execute experiments. 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. Experiment adaptation facilitates model updating as objectives shift from validation to variable screening, system exploration and evaluation. Towards this end, we discuss a potential solution of probabilistic models that can be used to predict system behavior, and active learning algorithms for efficiently updating experiment management system to reach the simulation goal earlier.

## 1 Summary

Computation is viewed as a powerful formal framework and exploratory apparatus for the conduct of science(Honavar, Hill, and Yelick 2016). The claim is supported by the observation that computation, mathematics, and science are often used ubiquitously for

augmenting scientific knowledge by answering. By experimenting on models, we can anticipate behavior and flaws that would otherwise be difficult to discover in the actual system or may prove to be cost ineffective. Determination of such characteristics about the system manually requires substantial human effort, and may still miss distinctions in content and learning. Since explicit characterization in this manner is infeasible, approaches that do not require exhaustive experimentation need to be considered (Murphy 2011). In supporting computational discovery, reliable models are necessary but not sufficient in addressing complex research problems (Kleijnen et al. 2005) as proper design and management of experiments are critical to instill confidence in the use of computer simulations (Ewald and Uhrmacher 2014). Scientific experiments are defined with a set of goals and have a purpose to answer certain questions. Its strength lies in analyzing the real effects of alternative conditions and courses of action using what if hypotheses (under certain assumptions), on a model representing the fundamental behavior of the system. The use of simulation models for scientific experiments and model discovery has been well established ((Teran-Somohano et al. 2015). In order to produce improved experimentation practices and support computational discovery, the dependencies among goals, hypotheses, and experiments needs to be considered for evaluation of computer simulation studies(Yilmaz, Chakladar, and Doud 2016). The principles of Model-Driven Engineering (MDE) aid the transformation process and to facilitate the search within the operational level of hypotheses and the tactical level of experiments. Computational models as abstract representation is widely used to solve research queries effectively (Glotzer et al. 2013). These tools are useful in conducting simulation experiments and addressing questions about the simulation model. Yet, these tools are only useful to capture few aspects of the entire scientific process and fail to include the efficient execution, analysis and management of the complete process under study, for e.g., estimation of necessary

replications (Robinson 2005) or assessing the statistical significance of the results (Lee and Arditi 2006). There is a huge gap between the ability to collect, store and process the data and the ability to make effective use of the data to advance discovery (Honavar, Hill, and Yelick 2016). Despite successful automation, the simulation experiment management and data analysis requires considerable human intervention. Cognitive tools are required to bridge the gap between the data acquisition and data analysis for developing abstractions to address research questions. This calls for intensive research study focused on development, analysis, integration and simulation of information processing abstractions of natural processes, aided with formal methods and tools for their analyses and simulation (Honavar, Hill, and Yelick 2016). It also requires the abstract representation of the different aspects of the scientific process, the development of computational artifacts (representations, processes, protocols, workflows, software) that represent the understanding of the system behavior, and the integration of the resulting cognitive tools into collaborative human-machine systems and infrastructure to advance science.

The main contribution of the research is the utilization of Model-Driven Engineering (MDE) principles and practices to assist in generating experiment models, and to manage the overall lifecycle of simulation experiments. The premise is based on the observation that proper and reliable understanding of the impact of simulation parameters on model outcomes requires efficient, high-dimensional design of experiments.

We address some of the challenges in experiment lifecycle management, by defining a framework for hypothesis testing that is simulation platform independent by using the 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. Our aim is to show in extensive computational experiments that the integration of learning method in simulation

experiments can lead to higher accuracy in prediction of system behavior with significantly fewer experiments than a random learner, in many cases with perfect accuracy without exhaustive experimentation. By incorporating the results of model checking into learning networks, better experiments can be developed at a faster rate to increase knowledge gain. Bayesian networks can be used to characterize the statistical dependencies between the system variables. The probabilistic representation of the simulation system can be used for generating an explanatory inference of the system which is easier to understand and make the decision making process quicker. We discuss the theory of explanatory coherence for evaluating and revising hypotheses while using it as a run-time cognitive model that evolves via experimentation toward an explanatory theory of the system under study.

## **2 Research Goals**

To this end, our objective is to develop an open-source MDE-enhanced Eclipse tool chain to identify, prioritize, formulate questions, and conjecture mechanisms, define or generate experiments designed to answer questions, draw inferences, and evaluate results within an incremental and iterative process. A Domain-Specific Experiment Management Language is developed towards defining families of simulation experiments, in accordance with the DOE methodology. Furthermore, the experiment model will be used to test and invalidate the assumptions or hypotheses of the simulation model. The experiment model is used to test and invalidate the evidences of the simulation model. In this process the user can devise questions to test temporal evidences about the model. Temporal properties that describe the results of the observation have a degree of acceptability of its own. This is followed by the implementation of the experiment specifications in the simulation run. The advantage of this approach is that it can be used to remove redundancy in a system by identifying and eliminating duplicate models. Experimentation becomes a

seamless part of simulation development, by explicit representation of experiment models and hypotheses for the experiments. The approach also facilitates synthesis and execution of experiments along with validation and comparison of the experimentation models. The standardization of the entire process improves reproducibility and reliability of simulation results.

The results of the experiment is analyzed and a probabilistic model checker is used as the formal method of verification to test the hypotheses against the evidences. The results of the model checking are incorporated into Bayesian networks which is used to illustrate the probability of events in the simulation model. These can be incorporated for active learning and automation in the system for selecting better experiments at a faster rate to increase knowledge gain.

### **3 Strategy**

The principles of Model-Driven-Engineering has emerged as a practical and unified methodology to alleviate the complexity of platforms and express domain concepts effectively. The use of platform independent domain models along with explicit transformation models facilitates deployment of simulations across a variety of platforms. While the utility of MDE principles in simulation development is now well recognized, its benefits for experimentation have not yet received sufficient attention.

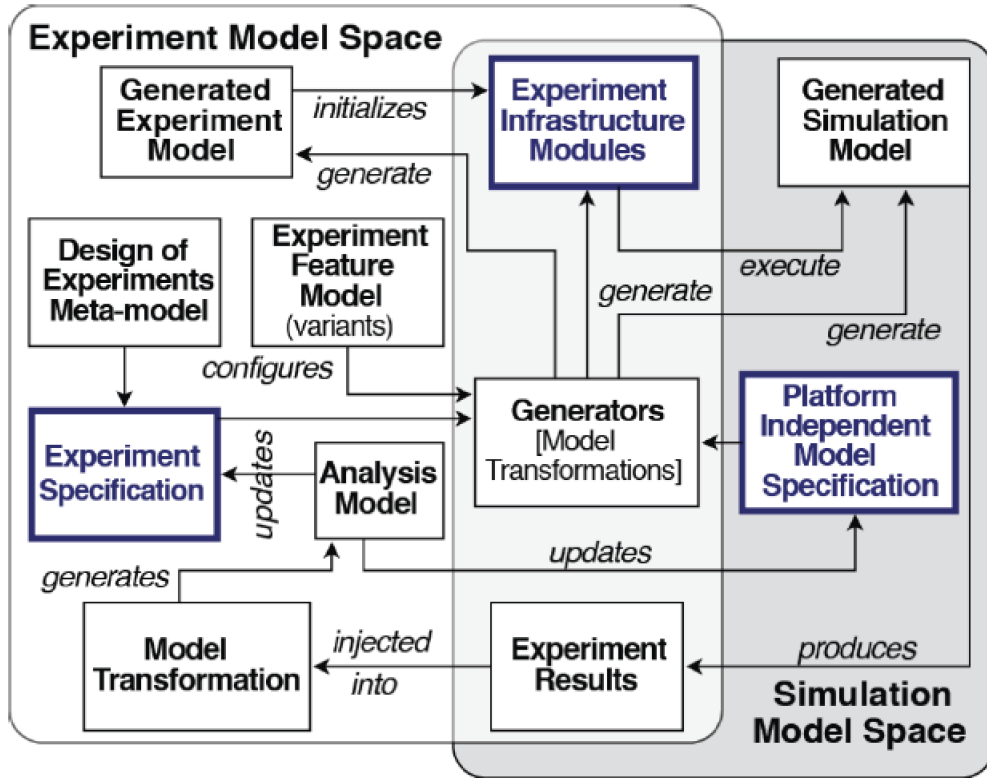


Figure 1: Experiment Management Framework

A conceptual framework that integrates MDE, agent models, and product-line engineering to manage the overall lifecycle of a simulation experiment is presented in Figure 1. In the component architecture, the experiment and simulation model spaces are tightly coupled to orchestrate the co-evolution of simulation and experiment spaces as learning takes place.

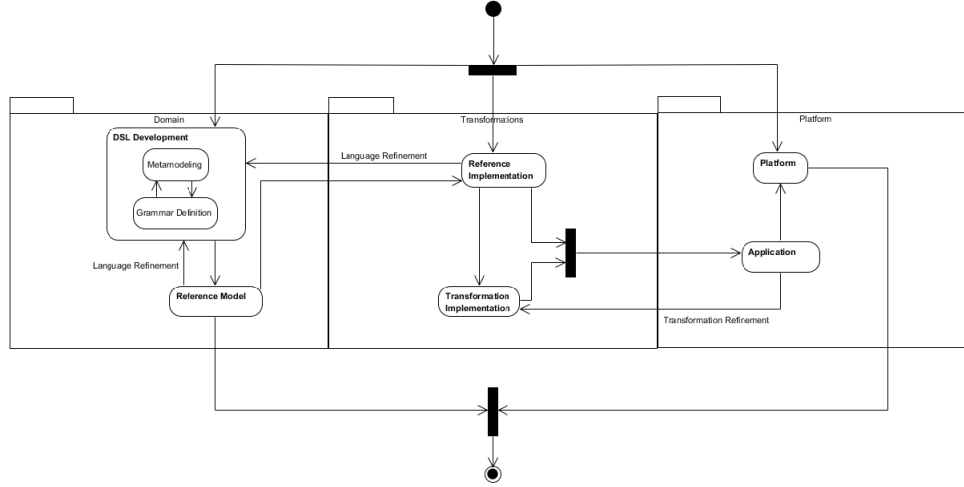


Figure 2: Development Process

First, we will give a high-level overview of the system behavior. Then we will discuss the development of the DSL 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.

### 3.1 Overview

Figure 3 presents a high-level overview of the activities that occur in the experiment lifecycle management. The only input to the process is the text of the DSL editor. After execution, the final product is a probabilistic 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.



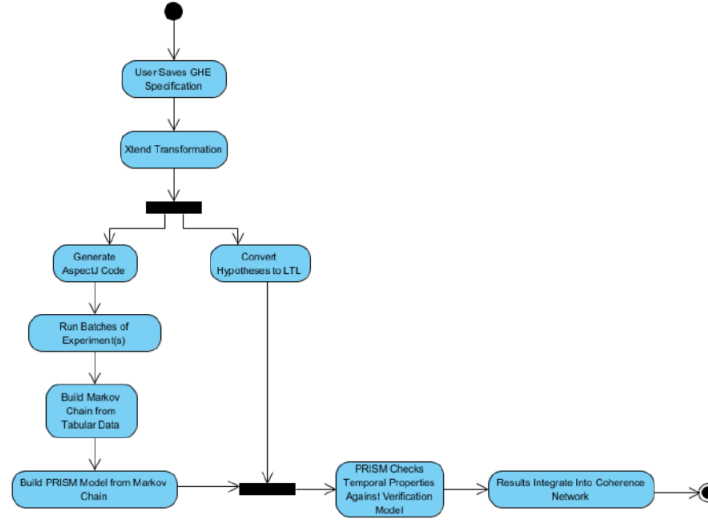


Figure 3: System Activity Diagram

- 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 the next section. Once the specification is saved, the transformations take place, and the next activity is executed.

- 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.

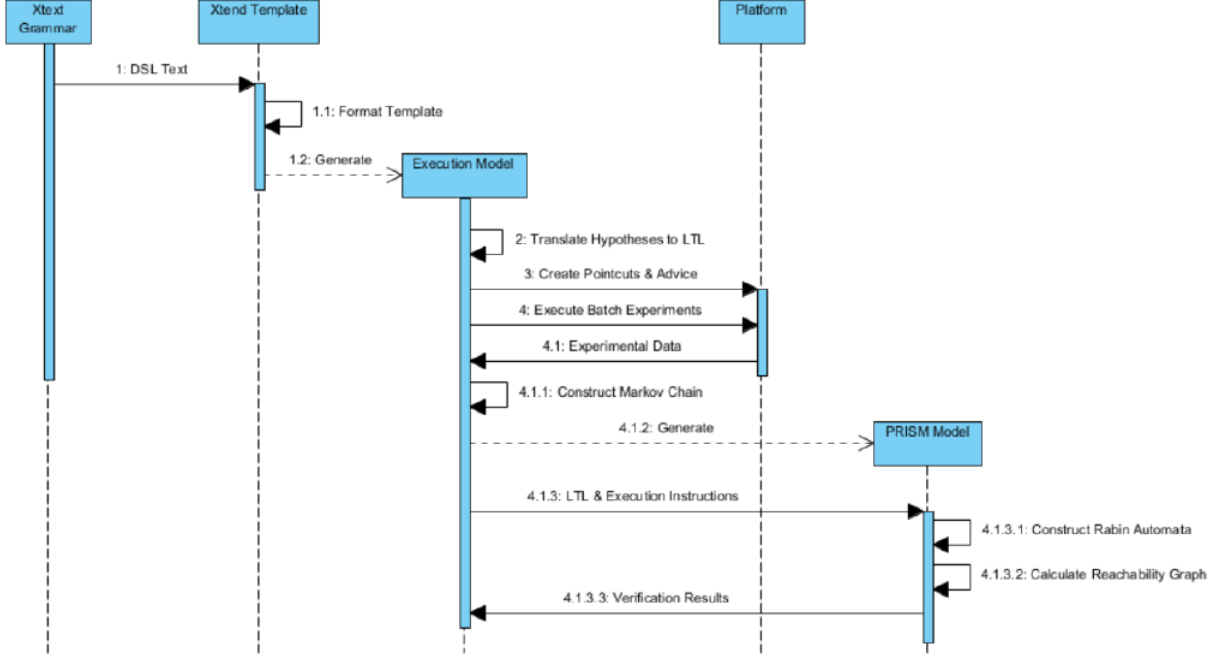


Figure 4: System Sequence Diagram

- Model Checking Results Integrate Into Bayesian Network

Bayesian Networks are effective in communicating which factors are most influential on the performance measures. They allow for combining highly dissimilar types of data (i.e., numerical and categorical), converting them to a common probabilistic framework, without unnecessary simplification; they readily accommodate missing data; and they naturally weight each information source according to its reliability. We analyze the experimental results and predict the conditional probability of events to construct the Bayesian Network. The resulting network is displayed to the user.

## 4 Development Work

The principles of model-driven software development are used throughout the development process. We started by developing a metamodel for the language in the form of a UML class model in order to facilitate understanding of the domain. We used this metamodel as a roadmap to develop a context-free grammar in Back-Naur Form (BNF). Next, we transformed this BNF grammar into an Xtext grammar and evaluated its readability in a reference model. The grammatical constructs defined in the Xtext grammar were used to identify classes and structures for a reference implementation, where a use-case for the application was developed and tested. Through development of the reference implementation, we were able to identify sections of code that were candidates for text-to-model transformation. These transformations bridge the gap between reference model, reference implementation, and platform.

The process was an effective tool for streamlining the development of a DSL-driven application. By focusing on the way the language will be used before the implementation, we were able to create a highly expressive language while providing support for platform versatility.

### 4.1 Metamodel

The metamodel encompasses all the major components of the GHE framework. It includes the goal of the experiment, model definition, hypothesis and an experiment. The metamodel for our study is shown below.

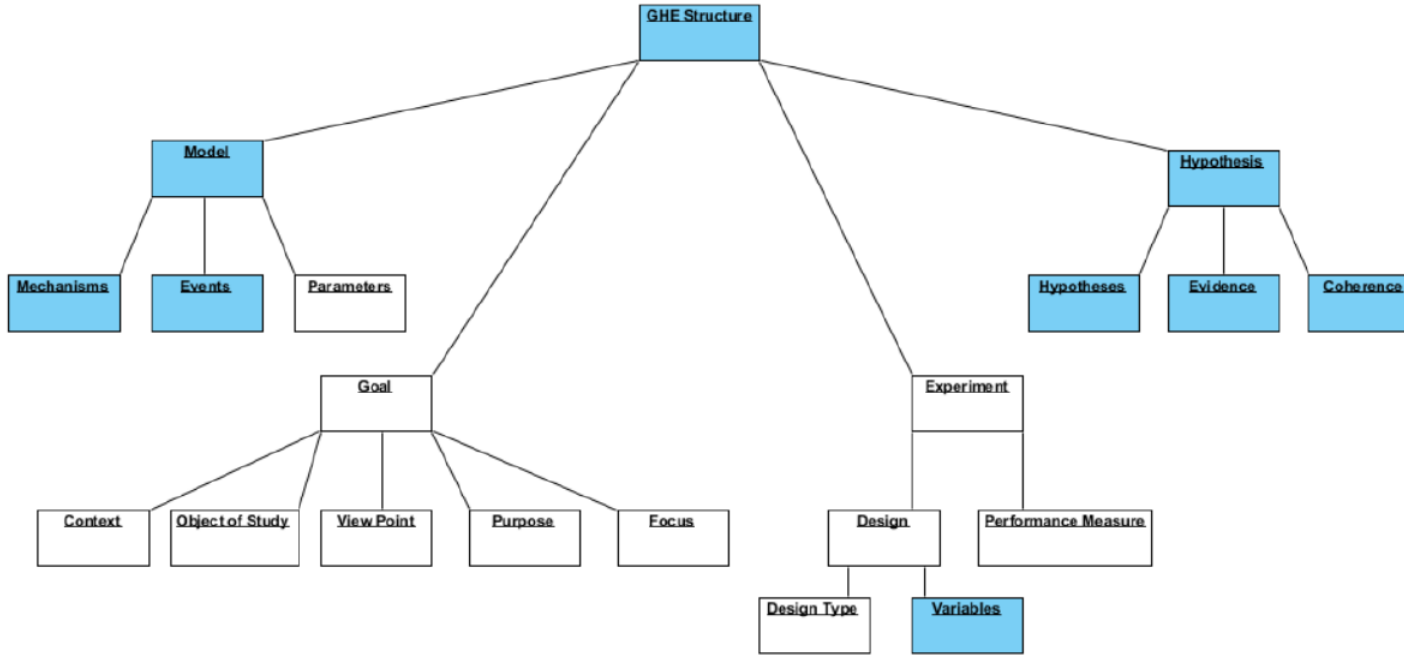


Figure 5: Goal-Hypothesis-Experiment Language Structure

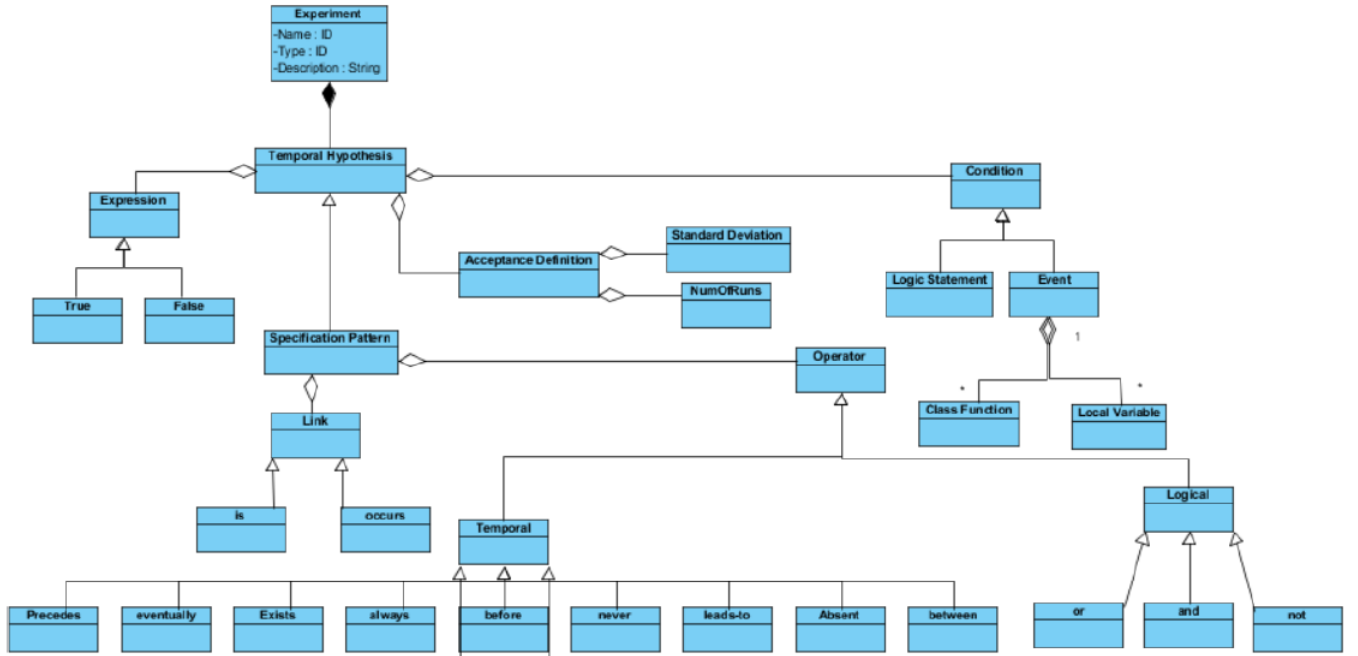


Figure 6: Metamodel representing major components of the GHE framework

## 4.2 BNF

The first step to define this new language was to describe the syntax in an easily readable/writable format. The BNF notation is used to define the syntactic grammar. In standard BNF, a grammar is defined by a set of terminal and non-terminal symbols. We defined the grammar in the standard form, without use of extended BNF symbols like \*, +, -, etc., because those aspects, while making development easier, make the grammar harder to read. Since the purpose of the BNF development was to discover how the language should look, the most easily interpreted form seemed like the best choice.

The BNF grammar definition during the early stages of development is listed below:

```
<Model> ::= ExperimentOntology
<ExperimentOntology> ::= ModelSection | Goals | Hypothesis |
Experiment
<ModelSection> ::= model <id> {<Mechanism> <EventDescriptor>
<Factor>}
<Mechanism> ::= mechanism <id> = <Reaction> <GuardCondition> ->
<Reaction>
```

The next step in development was to implement the BNF grammar using a language engineering framework. We proceed by implementing the grammar in Xtext.

## 4.3 DSL

The DSL for simulation experiment model development is developed using the Xtext DSL development environment on Eclipse Neon, by translating the experiment ontology metamodel. The DSL is also used to define a set of hypotheses for experiment model validation and verification. The simulation experiment specifications are then used for

the description of a simulation experiment. After the generation of the experiment model design with all the elements imposed by the experiment ontology, it is transformed and stored in a properties file. This file is then used to run the MASON model and collect the results of the simulation run.

The strategy is to develop a grammar to help the user specify experimentation parameters and to verify the conditions to trigger them and determine a desired plan of action. The result of the action is then translated to the user after performing a set of validation and verification.

The transition from BNF to an Xtext implementation is straightforward. Each non-terminal in BNF is treated as a grammar rule, and terminals are either IDs or new keywords. As a simple example, the following grammar rules in Xtext correspond to the BNFs transition shown in the previous section:

```

ExperimentOntology :
    ModelSection | Goals | Hypothesis | Experiment ;

ModelSection:
    'model' (modName = ID)
    '{'
    (mechanisms += Mechanism)*
    ((events += EventDescriptor)?) *
    (parameters += Factor)*
    '}' ;

Mechanism:
    'mechanism' (mechanismName = ID) ' = ' (LHS = Reaction)
    (condition = GuardCondition)? ' -> ' (RHS = Reaction) ;

```

## 4.4 Reference Implementation

The reference implementation for this model is a java program that serves as an implementation of the rules from the grammar as java classes. As we developed the program, we discovered that some aspects of the metamodel that were used in the grammar were simply textual devices for readability and served no purpose for computation. These aspects were subsumed as identifiers in text recognition algorithms for the relevant classes. Some of the use-cases are described below.

- `Usecase1.java`

This class initializes the pieces of an experiment specification and executes the experiment with the specified parameters.

- `ToDeliveryProperties.java`

In this class we construct the `delivery.properties` file from the factors and their values specified by the user. This file is later used for the MASON simulation run.

- `ToISHCProperties.java`

In this class we construct the `ishc.properties` file from the factors and their values specified by the user. This file is later used for the MASON simulation run.

- `Query.java`

In the `Query` class, a string that represents a temporal property is passed to the constructor and is set as a global variable. The constructor calls a method, `detectEvents()`, which in turn calls `detectPattern()` and `detectPostfix()`. The objective of these functions is to discover which Linear Temporal Logic (LTL) formula matches the sentence from the grammar and to find the conditions that will be inserted into



the formula. This class subsumed most of the temporal specification keywords as well as Conditions.

- `ConvertToLTL.java`

The purpose of this class is to take the events identified from the `Query` class and replace them in a matching LTL formula that can be found in a `patterns.xml` file. An LTL formula in the XML file would be in a form like:  $\Box(Q \ \& \ !R \rightarrow (!P \ W \ R))$ , where each of the letters (aside from `W`, which represents weak until in temporal logic), represents a placeholder for a condition from the DSL. The difficulty in textually substituting these letters for their condition identifiers was due to the fact that the conditions could have the same capital letters in them as the placeholders in the formula, causing unexpected results.

- `ExperimentExecuter.java`

This class runs the model and gives the output.

- `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  $A \leq B$ . 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.

The class diagram generated from the reference implementation, shown in figure below, resembles the metamodel generated in the first step of our development process.

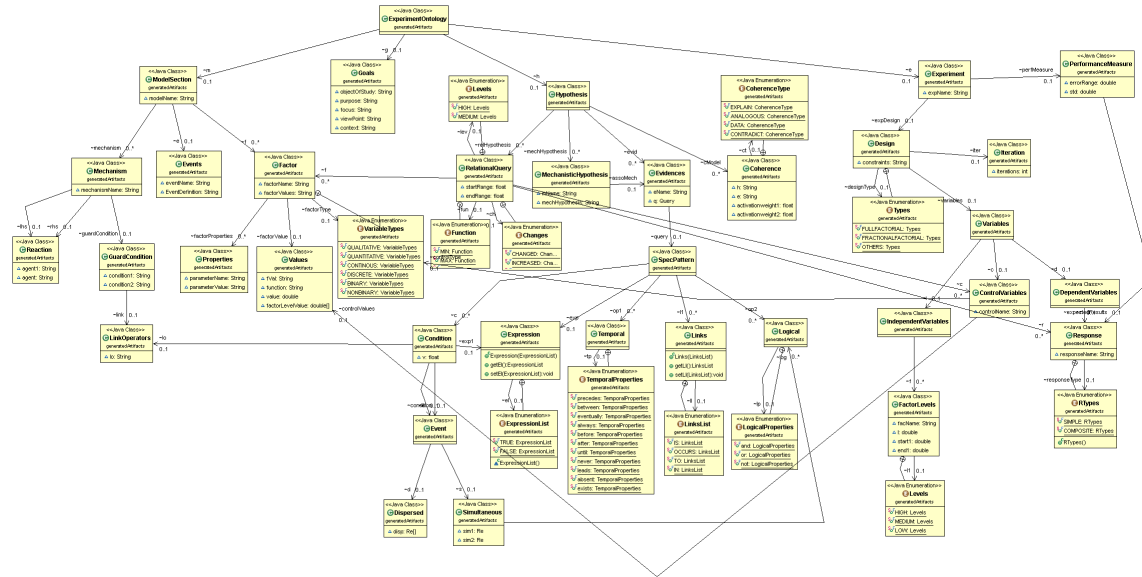


Figure 7: Class diagram generated from the reference implementation

## 4.5 Reference Model

In order to test the validity of our framework and the practical utility of the approach, we used our project to demonstrate the ISHC model. The DSL we developed is abstract and free of any technical terms. The DSL covers all relevant concepts of the domain with language elements. All schematically-implementable code fragments of the reference implementation are covered by constructs of the DSL. The reference ISHC model is an instance of the DSL. The DSL for simulation experiment model is developed by mapping the experiment ontology metamodel.

### 4.5.1 Model

Model consists of a specification about the models name, the mechanisms, the events and the factor parameters. Mechanisms consist of the processes which is assumed to take place in the simulation system. Events define the path for tracing the functions that evaluate the events that form a part of the evidences. Parameters are the inputs to the model and their properties, which have an impact in determining the response/output of the simulation run.

```
model ISHC{  
    mechanism M1 = inflammatoryAgent + Kupffercells  
    [inflammatoryAgent > inflammatorythreshold] -> Cytokines  
    mechanism M2 = inflammatoryAgent + Kupffercells [noOfCytokine  
    > cytokineThreshold] -> Cytokines  
    event inflammation = 'void  
    ishc.model.KupfferCell.handleInflammation()'  
    parameter LPS = Solute with properties {tag: LPS, bindable: true,  
    bolusRatio:1.0 , pExitMedia: 0.1 , pExitCell: 1.0 , bindProb : 0.25 ,  
    bindCycles : 1 , numProps : 8 , membraneCrossing: true, bileRatio :  
    0.5 , core2Rim : 0.50 , metProbStart : 0.3 , metProbFinish : 0.3 ,  
    metabolites: 'LPS-Metabolite_A', inflammatory : true , pDegrade :  
    0.0}  
    parameter forwardBias = DISCRETE with values {0.5}  
}
```

## 4.6 Goal

Goals define what the purpose of the experiment is. It also gives an idea about the specific field of concern and the context under which the study is performed.

```
goal  
{  
    object of study : 'Immune system influence on hepatic cytochrome  
P450 regulation'  
    purpose : 'Explain / characterize'  
    focus : 'the reason for changes in downstream drug metabolism  
and hepatotoxicity'  
    view point : 'based on the response of hepatic cytochrome P450-  
regulating mechanisms'  
    context : 'when health and/or therapeutic interventions change.'  
}
```

### 4.6.1 Hypotheses

Hypotheses consists of relational hypotheses, mechanistic hypotheses and expected regularities. Mechanistic hypotheses deal with the effect of changes in the mechanism of the model. Relational hypotheses deal with the impact of changes in inputs or outputs. In order to represent behavioral changes in the model, we focus on mechanistic hypotheses for the study. Expected regularities are the temporal properties that are to be verified in the experimental run. It is stated in terms of state of factors and their properties.

The coherence model describes the explanatory coherence relation [Thagard 1989] between the hypothesis and the evidence. The evidence can have an activation weight which indicates its reliability. This is used to establish the weightage of the link between

the evidence and hypothesis in the coherence network. We identified the explanatory coherence concept that would be relevant in our framework and would help in discovering the model mechanisms. To summarize how we will be implementing explanatory coherence theory we listed the key terms from the principles, which was used in the experiment definition.

- EXPLAIN

We use this if a coherence exists which explains or supports evidence(s) and hypothesis(es). In this case excitatory links are established between the evidences and hypotheses and an activation weight is assigned to each link. As the number of such links between the hypotheses and evidences increase, the weight on the links in the network decreases.

- ANALOGOUS

We use this if hypothesis and evidence are analogous to each other. Analogy, produces excitatory links between the similar evidences and hypotheses and an activation weight is assigned to each link.

- DATA PRIORITY

The principle of data priority is used to set up explanation-independent excitatory links to each data unit from a special evidence unit that always has an activation of 1. The data units can have activation level specified depending upon its reliability index. When the network runs, activation spreads from the special evidence unit to data units and then to the units representing explanatory hypotheses.

- CONTRADICT

We use this if there is incoherence between the evidence(s) and hypothesis(es). In this case inhibitory links are established between the evidences and hypotheses and a negative weight is assigned to each link. As the number of such links between the hypotheses and evidences increase, the weight on the links in the network changes.

The conditions are grouped under these categories which are used in designing a query based DSL to allow the user to define the hypothesis which can be used to develop a simulation model.

```

hypotheses
{
    mechanistic hypotheses
    {
        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)
    }
}

```

### 4.6.2 Experiment

The ontology for the experiment section encompasses the structural elements of an experiment which includes the experiments design and performance measure. Based on the models parameters and their levels, the hypotheses and goal of the experiment, a design is created that is used in subsequent steps of the experiment life-cycle.

The experimental design is defined by the dependent variables, the control variables, the 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, that is, the combination of factor levels.



```

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
            }
        }
    }
}

```

#### 4.6.3 Performance Measure

An experiment consists of performance measure parameters which defines the criteria for successful experimental run. Basing on this measure we can decide whether additional iterations are required for satisfying the experiments objective. It is defined in terms of the expected value of the response or output of the experiment and its standard deviation.

```
performance measure is
{
    cytokines= 500 +-10
}
```

In the above example, the expected value of the cytokines after successful experiment execution is 500 with a standard deviation of 10.

## 4.7 Code Generation

We used the Xtend code generation process for mapping the DSL to platform. A set of templates were derived from the reference implementation and used for the transformation step.

```

class DOEGenerator implements IGenerator {
    override void doGenerate(Resource resource, IFileSystemAccess
fsa) {
        fsa.generateFile('ishc.properties',
            toISHCProperties(resource.allContents
                .filter(typeof(ModelSection)).head))
        fsa.generateFile('delivery.properties',
            toDeliveryProperties(resource.allContents
                .filter(typeof(ModelSection)).head ,
                resource.allContents.filter(typeof(Experiment)).head))
        fsa.generateFile("KupfferCell.java",
            toKupfferCell(resource.allContents
                .filter(typeof(ModelSection)).head))
        fsa.generateFile("Hepatocyte.java",
            toHepatocyte(resource.allContents
                .filter(typeof(ModelSection)).head))
    }
}

```

## 4.8 Generated Artifacts

The experiment specification is used for code generation by the template engine. The experiment specification defined using the DSL and the generated artifacts were used to run the ISHC simulation model in MASON to get the results.

## ishc.properties

```
# model parameters
stepsPerCycle = 1
# component parameters
forwardBias = 0.5
```

Code generated in  
response to :  
parameter forwardBias =  
DISCRETE with values  
{0.5}

## delivery.properties

```
dose.0.solute.0.tag = LPS
dose.0.solute.0.bindable = true
dose.0.solute.0.pExitMedia = 0.1
dose.0.solute.0.pExitCell = 1.0
dose.0.solute.0.numProps = 3
dose.0.solute.0.property.0.key = membraneCrossing
dose.0.solute.0.property.0.type = boolean
dose.0.solute.0.property.0.val = true
dose.0.solute.0.property.5.key = metabolites
dose.0.solute.0.property.5.type = map
dose.0.solute.0.property.5.val = Metabolite_A
=> <0.0,1.0>
dose.0.solute.0.property.6.key = inflammatory
dose.0.solute.0.property.6.type = real
dose.0.solute.0.property.6.val = true
dose.0.solute.0.property.7.key = pDegrade
dose.0.solute.0.property.7.type = real
dose.0.solute.0.property.7.val = 0.0
```

Code generated in  
response to :

parameter LPS = Solute with  
properties {tag: LPS,  
bindable: true , pExitMedia:  
0.1 , pExitCell: 1.0 ,  
bindProb : 0.25 , numProps:  
3 , membraneCrossing:  
metabolites: 'LPS-  
Metabolite\_A',  
inflammatory : true ,  
pDegrade : 0.0}

LPS are at levels : 0 , 1.0

## KupfferCell.java

```
public void handleInflammation()
{
    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(s.type.equals("inflammatoryAgent"))
        {
            if("inflammatoryAgent" > "inflammatorythreshold")
            numCytokines++;
        }
        if(s.type.equals("inflammatoryAgent")){
            if("noOfCytokine" > "cytokineThreshold")
            numCytokines++;
        }
    }
    if(numCytokines >= parent.cytokineThreshold)
    {
        return;
    }
}
```

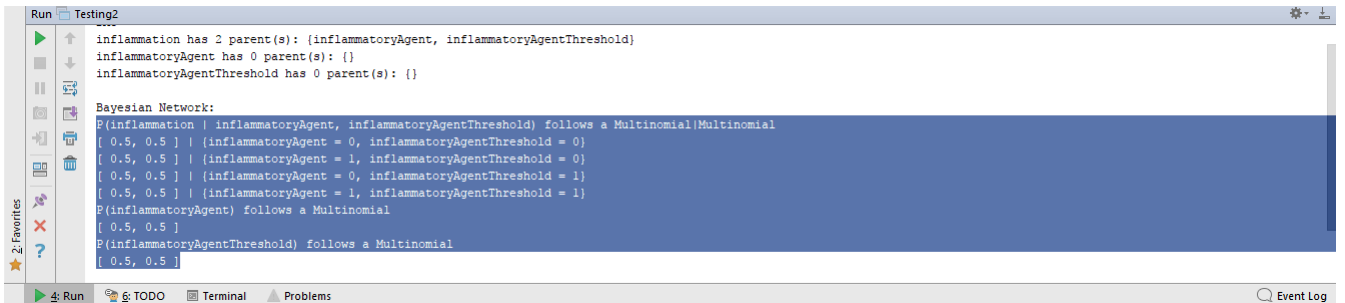
Code generated in response to  
the mechanism:

mechanism M1 =  
inflammatoryAgent + Kupffercells  
[inflammatoryAgent >  
inflammatorythreshold] ->  
Cytokines

mechanism M2 =  
inflammatoryAgent + Kupffercells  
[noOfCytokine >  
cytokineThreshold] -> Cytokines

## 4.9 Bayesian Network

Bayesian networks are used to characterize the statistical dependencies between the system variables. The probabilistic representation of the simulation system can be used for generating inference of the system which is easier to understand and make the decision making process quicker.



```
Run Testing2
inflammation has 2 parent(s): {inflammatoryAgent, inflammatoryAgentThreshold}
inflammatoryAgent has 0 parent(s): {}
inflammatoryAgentThreshold has 0 parent(s): {}

Bayesian Network:
P(inflammation | inflammatoryAgent, inflammatoryAgentThreshold) follows a Multinomial/Multinomial
[ 0.5, 0.5 ] | {inflammatoryAgent = 0, inflammatoryAgentThreshold = 0}
[ 0.5, 0.5 ] | {inflammatoryAgent = 1, inflammatoryAgentThreshold = 0}
[ 0.5, 0.5 ] | {inflammatoryAgent = 0, inflammatoryAgentThreshold = 1}
[ 0.5, 0.5 ] | {inflammatoryAgent = 1, inflammatoryAgentThreshold = 1}
P(inflammatoryAgent) follows a Multinomial
[ 0.5, 0.5 ]
P(inflammatoryAgentThreshold) follows a Multinomial
[ 0.5, 0.5 ]
```

## 4.10 Decision Tree

Our aim would be to reduce the uncertainty in the system and identify the relevance of the events. This information based approach helps in determining events that better characterize the model behavior and will be used as a reward for adapting experiment parameters to strongly prove the validity of mechanism in the system. Towards this end, we construct a decision tree by iterating through all the attributes of the dataset and calculate the entropy and information gain of each attribute.

```

37 //The following function calculates the entropy from the given data
38 private static double computeEntropy (int[] data, int c) {
39
40     double totalDataCnt = 0;
41     int count = 0;
42     double Entropy= 0;
43
44     for (int i = 0; i < 2; i++) {
45         if (data[i] == 0)
46             count++;
47         totalDataCnt += data[i];
48     }
49
50     if (totalDataCnt == 0 || count == 1)
51         return 0;
52
53     if(c==1){
54         for (int i = 0; i < 2; i++) {
55             if (data[i] != 0)
56                 Entropy += (-1) * (data[i]/totalDataCnt) * (Math.Log10(data[i]/totalDataCnt) / Math.Log10(2));
57         }
58     }
59     else {
60         if (data[0] != 0 || data[1] !=0)
61             Entropy = data[0] /totalDataCnt * data[1] / totalDataCnt;
62     }
63     return Entropy;
64 }
65
66
67

```

It then selects the attribute which has the smallest entropy (or largest information gain) value.

```

9 // The following function calculates the information gain
10 public static double computeGain (int colIndex, Tree_inst dataRows, Tree_inst resultRows, int c) {
11
12     ArrayList<int[]> dataCount = getDataCounts(colIndex, dataRows, resultRows);
13     int[] resultCountArr = getResultCountArray(resultRows);
14     double totalResultCount = 0;
15     double[] totalDataCount = new double[2];
16     double Entropy = 0;
17     double gain = computeEntropy(resultCountArr,c);
18
19
20
21     for (int i = 0; i < 2; i++) {
22         totalResultCount += resultCountArr[i];
23         for (int l = 0; l < 2; l++) {
24             totalDataCount[i] += dataCount.get(i)[l];
25         }
26     }
27
28     for (int i = 0; i < 2; i++) {
29         Entropy += (-1) * (totalDataCount[i] / totalResultCount) * computeEntropy(dataCount.get(i),c);
30     }
31
32     gain += Entropy;
33
34     return gain;
35 }
36

```

The dataset is then split by the selected attribute to produce subsets of the data. The algorithm continues to recursively on each subset, considering only attributes never selected before.

```

<terminated> Main_project [Java Application] C:\Program Files\Java\jdk1.8.0_131\bin\javaw.exe (Dec 8, 2017, 1:34:53 PM)
Printing Tree ::

pExitCell-7 = 0 :
|
| membrane_crossing-8 = 0 :
| |
| | pExitMedia-6 = 0 :
| | |
| | | InflammatoryAgent-0 = 0 :
| | | |
| | | | InflammatoryThreshold-1 = 0 :
| | | | |
| | | | | Metabolite_B-4 = 0 :
| | | | | |
| | | | | | Metabolite_N-5 = 0 : 0
| | | | | | Metabolite_N-5 = 1 :
| | | | | | |
| | | | | | | pDegrade-9 = 0 : 0
| | | | | | | pDegrade-9 = 1 :
| | | | | | | |
| | | | | | | | Bindable-2 = 0 :
| | | | | | | | |
| | | | | | | | | Metabolite_A-3 = 0 : 0
| | | | | | | | | Metabolite_A-3 = 1 : 0
| | | | |
| | | | | Metabolite_B-4 = 1 :
| | | | | |
| | | | | | pDegrade-9 = 0 : 1
| | | | | | pDegrade-9 = 1 :
| | | | | | |
| | | | | | | Bindable-2 = 0 : 0
| | | | | | | Bindable-2 = 1 :
| | | | | | | |
| | | | | | | | Metabolite_N-5 = 0 : 1
| | | | | | | | Metabolite_N-5 = 1 :
| | | | | | | | |
| | | | | | | | | Metabolite_A-3 = 0 : 0
| | | | | | | | | Metabolite_A-3 = 1 : 0
| | | | |
| | | | | InflammatoryThreshold-1 = 1 :
| | | | | |
| | | | | | Bindable-2 = 0 :
| | | | | | |
| | | | | | | pDegrade-9 = 0 :
| | | | | | | |
| | | | | | | | Metabolite_A-3 = 0 : 1
| | | | | | | | Metabolite_A-3 = 1 :
| | | | | | | | |
| | | | | | | | | Metabolite_N-5 = 0 : 0
| | | | | | | | | Metabolite_N-5 = 1 :
| | | | | | | | | |
| | | | | | | | | | Metabolite_B-4 = 0 : 1
| | | | | | | | | | Metabolite_B-4 = 1 : 0
| | | | | |
| | | | | | pDegrade-9 = 1 : 0
| | | | | | Bindable-2 = 1 : 0

```

The decision tree is constructed with each non-terminal node representing the selected attribute on which the data was split, and terminal nodes representing the class label of the final subset of this branch.

If a tree that is too large, it risks overfitting the training data and poorly generalizing to new samples. A small tree might not capture important structural information about the sample space. However, it is hard to tell when a tree algorithm should stop because it is impossible to tell if the addition of a single extra node will dramatically decrease error. This problem is known as the horizon effect. A common strategy is to grow the tree until each node contains a small number of instances then use pruning to remove nodes that do not provide additional information. Pruning should reduce the size of a learning tree without reducing predictive accuracy as measured by a cross-validation set.



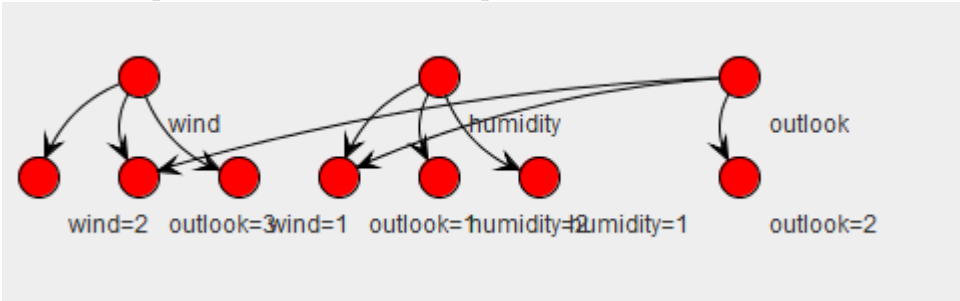
```

40
41 private int removeLeaf(Tree_nodes newTree, int count, int p) {
42
43     if (newTree.getSubTreeList() == null || newTree.getSubTreeList().isEmpty()){
44         count++;
45         if (count==p)return count;
46     }
47     if (newTree.getSubTreeList() != null && !newTree.getSubTreeList().isEmpty()) {
48         List<Tree_nodes> keyArray = newTree.getSubTreeList();
49         for (int y = 0; y < keyArray.size(); y++) {
50             count = removeLeaf(keyArray.get(y),count,p);
51             if (count==p){
52                 int c= count01new(newTree,0,0);
53                 newTree.getSubTreeList().remove(y);
54
55                 newTree.getSubTreeList().add(y,new Tree_nodes(c,null,null));
56             }
57         }
58     }
59     return count;
60 }
61
62
63 private int count01new(Tree_nodes newTree, int count0, int count1) {
64
65     if (newTree.getSubTreeList() == null || newTree.getSubTreeList().isEmpty())return (count1>=count0)?1:0;
66     if (newTree.getSubTreeList() != null && !newTree.getSubTreeList().isEmpty()) {
67         List<Tree_nodes> keyArray = newTree.getSubTreeList();
68         for (int y = 0; y < keyArray.size(); y++) {
69             if(keyArray.get(y).getValue() ==0)count0++;
70             else count1++;
71             count01new(keyArray.get(y),count0,count1);
72         }
73     }
74     return (count1>=count0)?1:0;

```

The entropy-based information gain implementation scans through the input attribute set and selects the best attribute. The selected attribute is then used to generate an experiment design with varying input levels of this attribute, in order to provide a better rationale for system's decision and discriminate competing hypotheses.

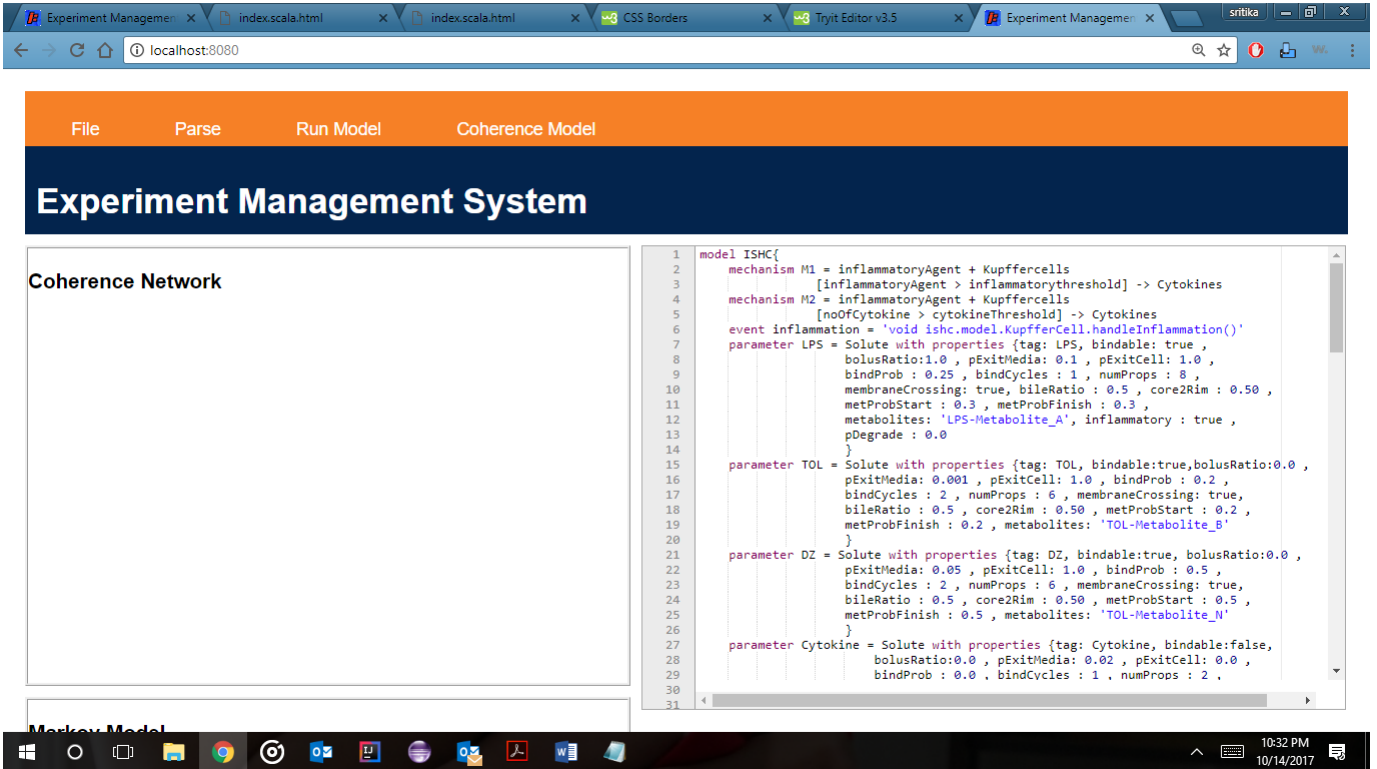
A visual representation of an example dataset is shown below:



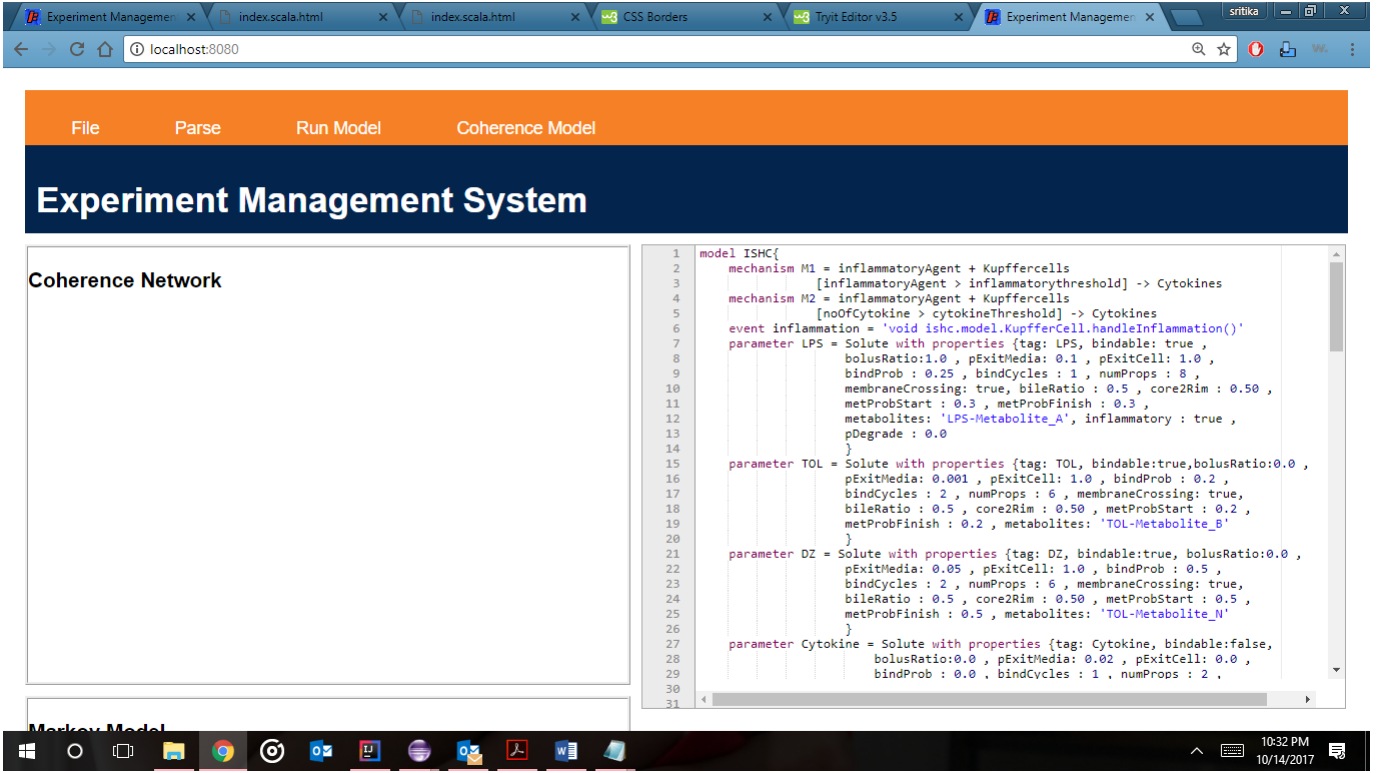
## 5 Application

We developed an application to demonstrate our framework and its functionalities. The experiment specification defined using the DSL and the generated artifacts were used to run the ISHC simulation model in MASON to get the results. The figures below illustrate

various functions supported by the application.

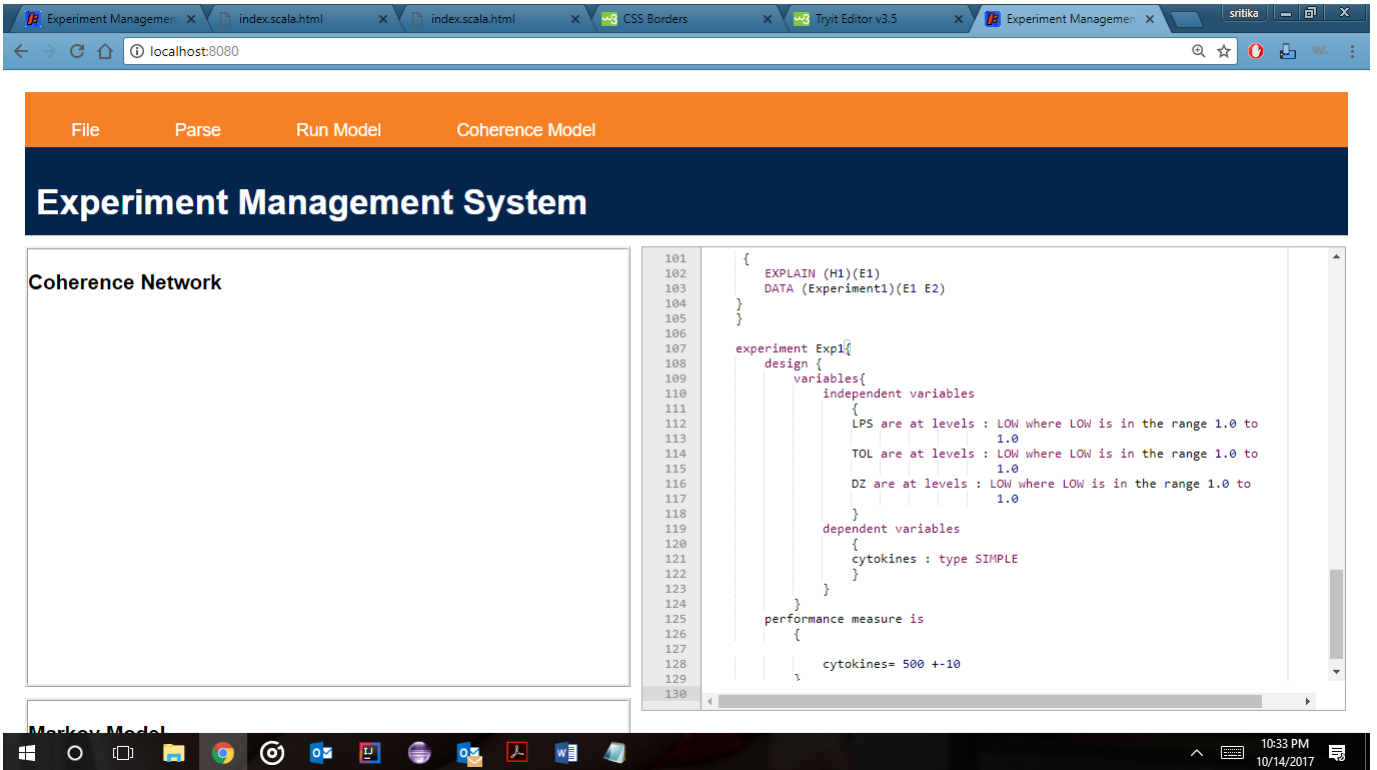


The web interface is shown above. The experiment management system shows an online DSL editor, coherence network and the Markov model. It supports functionalities like opening a reference model implemented using the DSL grammar, editing it and saving changes. The online editor also supports syntax highlighting.

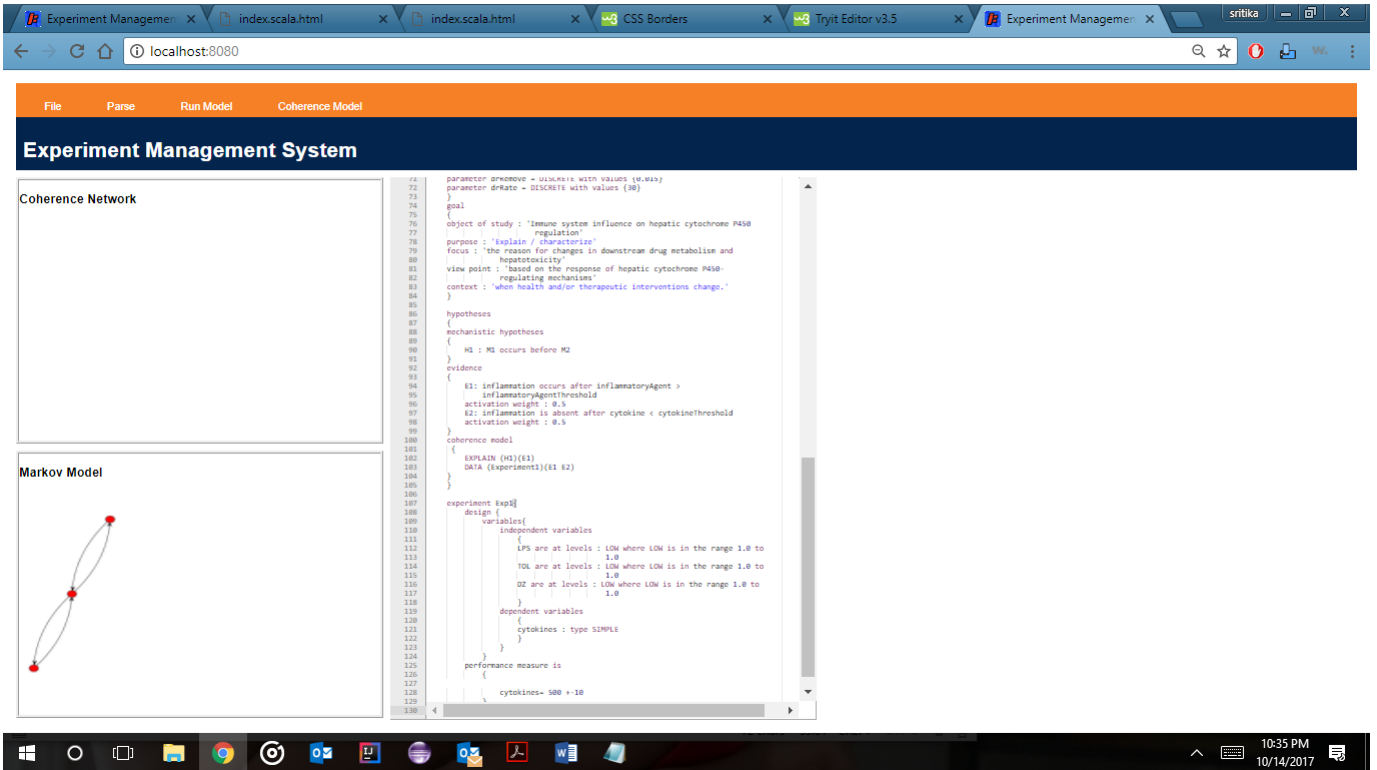


When the user selects to Parse the input, the following operations are executed:

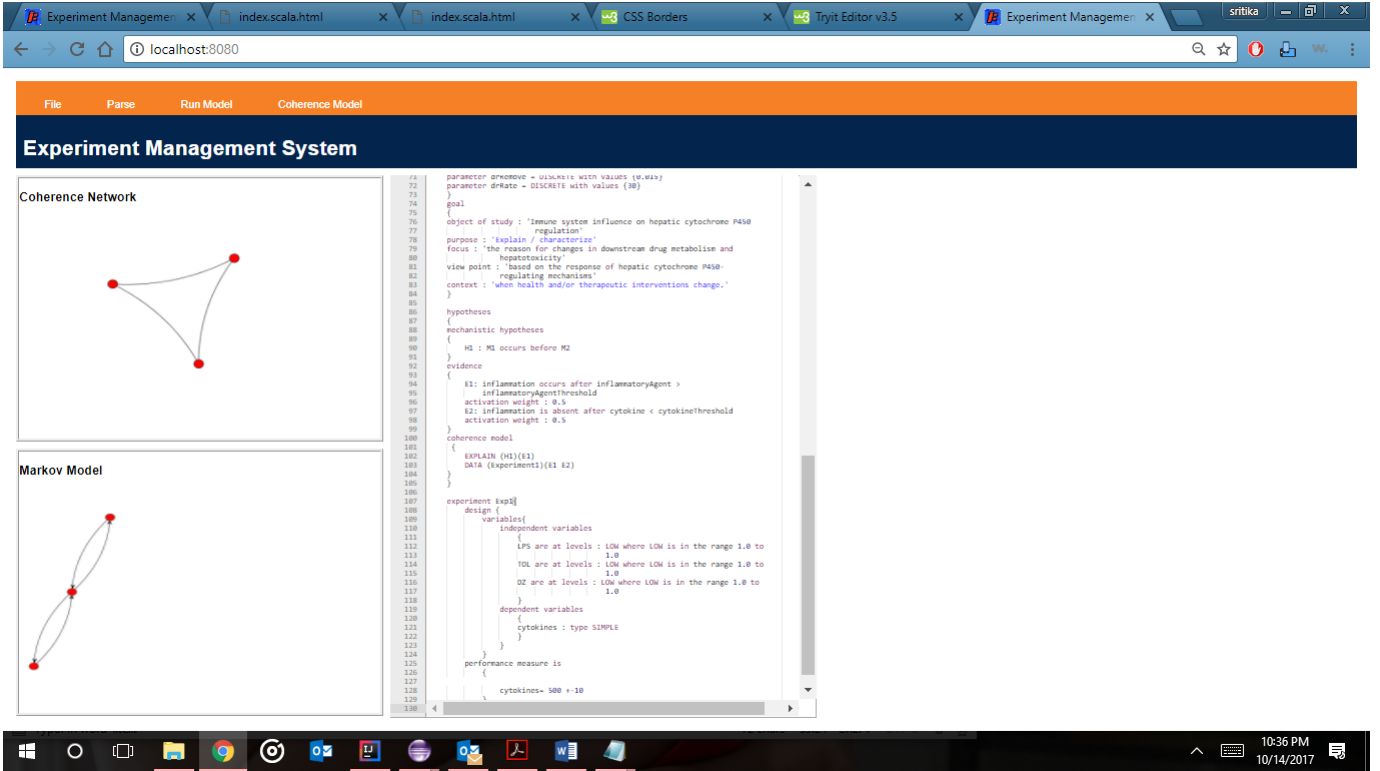
- The mechanistic hypothesis is translated and a model with the defined mechanism is generated.
- The parameters defined in the model section is used to generate the default values of the model parameters.
- The levels and values of independent variables in the experiment section are used to generate a full-factorial design. This is used to run the simulation model.
- The evidences are translated into LTL properties which is used in the model verification process.



When the user selects Run Model, the generated design matrix is used to run the simulation model which incorporates the user defined mechanisms. The results from the run are then analyzed and a Discrete Time Markov Chain(DTMC) model is constructed by carefully inspecting the results and recording the changes that trigger a transition in the output variables. This generated DTMC model is shown to the user. This is used for model verification using the PRISM interface.



When the user selects Coherence Model, the initial explanatory coherence network is generated based on the user definition in the DSL.



## 6 Future Work

### 6.1 Coherence/Model Discovery

There is a great synergy to be taken advantage of between mechanistic hypotheses and temporal properties. We can take observations made from in vitro or in vivo labs as evidences to our simulation model in the form of temporal specifications. We can use these evidences in a coherence model with mechanism changes in the program, specified by a mechanistic hypothesis to see if an evidence is invalidated with the mechanism change or supports one or more evidence. One use of this coherence model is to develop an intelligent agent that can take knowledge gleaned from the model and develop new mechanism changes that support the most evidences, in an effort to develop autonomous computing. Alternatively, or in the short term, this coherence model will be useful for

model discovery for a simulationist. If, for example, a simulationist introduces a new evidence to the system which is not supported by the current mechanisms, connections in the coherence model can help direct the user to a needed mechanism update that would not otherwise be known.

## 6.2 Mechanistic Hypotheses

Our future efforts will be directed towards generalizing this rule based definition of the hypotheses to capture various behavior of the model. Also our efforts will be directed towards identifying reaction scenarios for mechanistic hypotheses and associating the rules to a particular transformation process to facilitate computation. Transformation of these mechanism into computational code for simulation is a task in progress. Generalizing the transformation process to accommodate various scenarios is also a future goal.

The challenge lies in identifying different scenarios of mechanistic hypotheses and generalizing the DSL to allow their definition. The transformation of these hypotheses to mechanisms or expected behavior in the simulation model for the purpose of computation, is a challenge. These transformations might require additional information or assumptions on the model that must be provided by the user and that is not defined.

## 6.3 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 contradicting with in-silico experiment results.

## 6.4 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.

## 6.5 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.



## 6.6 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.