HAWK: A Domain Specific Language for Expressing Scientific Models

A Tool for Testing AMELIE
Honors Thesis

Ryan Marcus

May 26, 2014

Abstract

HAWK and SHAWK are a software package that enables an AMELIE developer to test a subset AMELIE's of abilities, specifically AMELIE's ability to run experiments on a stated subset of possible experimental domains.

Acknowledgments

This thesis represents the work of many individuals. I would like to thank Dr. Richard Snodgrass and Dr. Clayton Morrison for providing excellent mentoring and advice throughout the thesis process. David Sidi's initial work on the conceptualization of the validation domain, along with his understanding of TETRAD, proved invaluable. Qiming Shao was instrumental in developing the early AMELIE prototype shown in this thesis, and has generally been exceptionally tolerant of many late-night emails and rushed deadlines.

I would also like to thank the University of Arizona Honors College for their role in giving me this opportunity, despite the generally antagonistic relationship previously present.

Contents

Chapter 1

Introduction

HAWK and SHAWK are a software package that enables an AMELIE developer to test a subset AMELIE's of abilities, specifically AMELIE's ability to run experiments on a stated subset of possible experimental domains.

1.1 Motivation

The AMELIE project [13] seeks to provide tools to aid scientists during the scientific process. AMELIE seeks to aid in model creation, hypothesis generation, experiment proposal, data verification, and model validation. Eventually, AMELIE seeks to automate the entire process of casual inference.

Since the real-world is often complex, developing AMELIE using only real-world problem domains would introduce far too much complexity in the early phases of development. To alleviate this problem, HAWK (Heuristic Artificial World Kreator) provides an AMELIE developer with the capacity to define "worlds" which represent a simplified and well-defined version of some real-world phenomena, or an entirely artificial one.

Thus, HAWK and SHAWK (Scientist HAWK) enable an AMELIE developer to test AMELIE's abilities to execute experiments on a stated subset of possible experimental domains.

1.2 Related Works

While there are a few projects with goals similar to AMELIE [9], these tools lack (or do not discuss) any sort of testing framework. Many of these projects likely utilize unit testing and other quality assurance tools, none of them present a tool for the creation of artificial models that can be accessed using the same methods as one would access "real" data.

1.3 AMELIE

Since HAWK is a testing tool for AMELIE, it is appropriate to give a brief overview of AMELIE's structure.

1.3.1 Overview

An unpublished paper by Snodgrass et al. describes the structure of AMELIE as follows:

> Figure 1.1 is a schematic of the functional components of AMELIE. At the center is the *Cyber Workspace* that provides computational support of the SCM modeling language. The stretched ovals denote data structures. These include SCMs, the hypotheses generated from them, explicit representation of explanatory goals, and experiment workflows that can be executed in the environment of a study domain.
>
> All of the instances represented by these data structures will eventually be stored and maintained in a database, depicted on the far left of the workspace. The database will also stores any data resulting from experiment outcomes, as well as results of exploratory data analysis and other data manipulations. The database will include provenance metadata describing how data or SCM structures have been generated and tested.
>
> The five vertical grey columns represent the *thematic* roles of the workspace data structures and the processes that operate on them at particular stages of the empirical investigation cycle. In the *Model* column, SCMs are the focus of activity; here, SCMs are constructed and their structure and parameters revised. The *Hypothesis* column involves the representation, generation, selection and testing of hypotheses expressed in SCMs. The *Design* column is the locus of experiment design and planning. Throughout this presentation, we use the term "experiment" to refer to a plan to analyze existing data or a specification of one or more interventions that affect aspects of the environment.
>
> The experiment design process produces a workflow that can be executed in the study domain environment and also specifies the plan to analyze the results of the experiment outcome, including reliability analysis, hypothesis testing, and updating the model.
>
> The *Experiment* column represents the details of running the actual experiment in the study domain environment. These could be observational or interventional experiments.
>
> The study domain environment provides an interface to databases storing already available data, may include access to external data sources (far right of Figure 1.1), and may also provide an interface to an experiment apparatus that can perform interventions on the study subject, generating new observations.
>
> Finally, the *Lifecycle* column on the far left of Figure 1.1 represents the overall management of AMELIE's processing throughout the empirical investigation cycle.

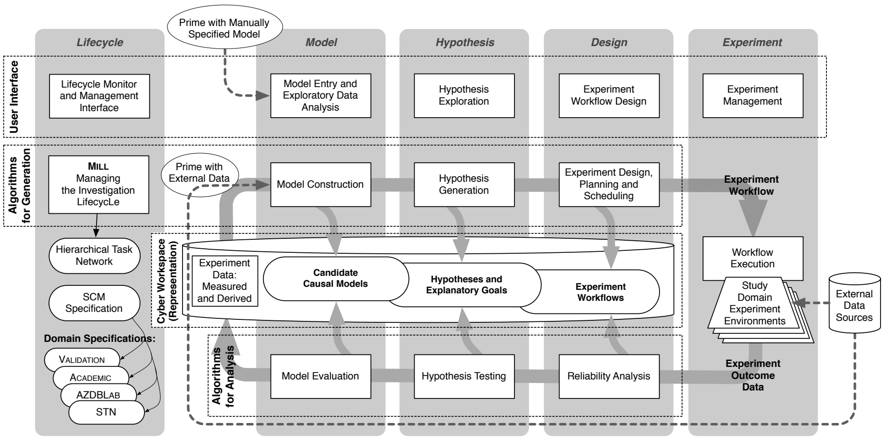


Figure 1.1: AMELIE Overview

1.3.2 Experimental Domains

Before exploring the specifics of HAWK, it is helpful to understand the current semantics of an experimental domain within AMELIE. An experimental domain is a set of experiments, each identified by a name. For example, the "quantum physics domain" might contain experiments called "the gold foil experiment" and "the prism experiment". Each experiment defines a set of independent variables (variables on which an intervention may be performed) and a set of dependent variables (variables that may only be observed).

Additionally, each experiment defines a subject class tree representing various pools of non-overlapping subjects. For example, the "the prism experiment" might have a subject class tree like this:

- Prisms
    - Glass prisms
        - Low quality
        - High quality
    - Plastic prisms

AMELIE assumes that all experiments within an experimental domain have the capacity to be executed. Executing an experiment requires specifying:

- a set of interventions (independent variables and the values to set them to),
- a set of observations (dependent variables to be recorded), and
- the number of subjects to draw from each subject class.

AMELIE assumes that the result of executing an experiment will be a list of independent trials, with data for each requested observation.

1.4 S/HAWK's Validation Strategy

HAWK defines an experimental domain, as described above, where a small, domain-specific language (DSL) defines each experiment. While an experimental domain would normally provide experiments that utilize real-world data, HAWK provides an experimental domain where each experiment is fully determined by the DSL, which will include a structural casual model and will indicate which variables are independent or dependent. However, HAWK does not expose the model itself to AMELIE; HAWK only exposes data generated from the model. This allows an AMELIE developer to validate that AMELIE enables a scientist to derive a model from data. The general validation workflow would look something like the following:

1. Alice creates a model M in the HAWK DSL which specifies the relationship between three independent variables, $I = \{a, b, c\}$, and two dependent variables, $D = \{d, e\}$.

2. Alice compiles this model into an experiment, E, which exposes a subset E_I of I and a subset E_D of D

3. Bob, if he desired, could use SHAWK to inspect the experiment E (possibly generating a few samples) before using AMELIE.

4. Bob, using AMELIE, sees that E is an available experiment declaring independent variables E_I and dependent variables E_D.

5. Bob uses AMELIE's tools to construct a model M' to represent the relationship amongst the variables exposed by E.

6. Once confident, Bob shows M' to Alice, who can compare M' to M and confirm if AMELIE (and Bob) were able to derive the correct relationship amongst the variables.

This testing workflow has several advantages. First, the isolation between Alice and Bob mirrors the relationship between a scientist and a domain. Because Bob can only see data generated from Alice's model, *Bob does not know the precise details of the model.* Second, Bob's ignorance of Alice's model ensures that any correct model derived by AMELIE and Bob was derived *exclusively from information that did not include Alice's model.*

Chapter 2

HAWK DSL

HAWK provides a small, extensible DSL (domain specific language) that can be utilized to implement different types of models. This chapter discusses the HAWK DSL and many new techniques used to create an easily extendable DSL.

2.1 Definition and Structure

Overall, the syntax of the HAWK DSL bears some resemblance to C [7] or to Go [2]. Examples of the HAWK DSL are made available in Appendix A.2 and A.3.

The HAWK DSL allows a programmer to specify a set of independent, dependent, and latent variables and to specify the relationship between those variables. Appendix A.2 shows an example linear model (based off of a model created to predict DBMS performance [5]) of exclusively dependent variables. Appendix A.3 shows example code that creates a Bayesian model with an independent variable (`wearing_raincoat`), a dependent variable (`average_rainfall`), and a latent variable (`average_humidity`). Both of these examples define a subject class tree, as well as the frequency of various subjects within non-leaf nodes.

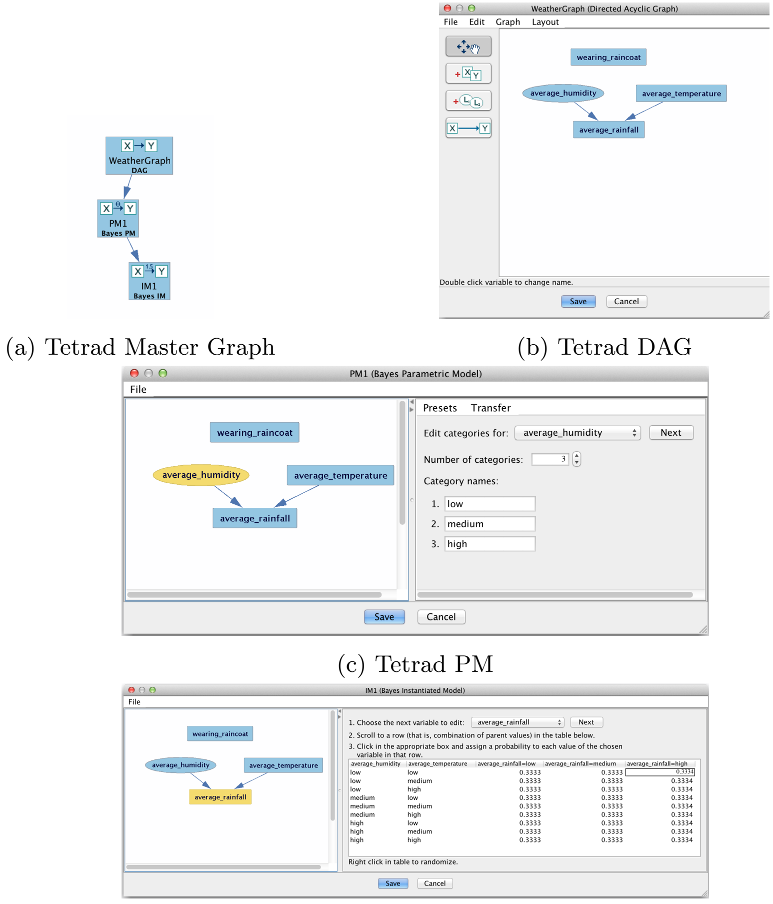
HAWK's main capabilities include the following:

- Define Bayesian networks with latent and non-latent variables
- Define structural equation models with latent and non-latent variables
- create subject class hierarchies to represent subject pools or populations

2.1.1 Bayesian Networks

Bayesian networks are commonly used to express relationships between multiple categorical variables [11], and are often used in machine learning [4].

HAWK enables a programmer to define an extended Bayesian network. Appendix A.3 shows how variables within a Bayesian network are defined. The opening `Bayes` declaration tells HAWK that the following `variable` clauses represent categorical, Bayesian variables. The first variable defined—`wearing_raincoat`—includes the line `intervene = true`, which makes the variable independent.



(a) Tetrad Master Graph

Double click variable to change name.

(b) Tetrad DAG

Edit categories for: average_humidity

Number of categories: 3

Category names:

1. low
2. medium
3. high

(c) Tetrad PM

1. Choose the next variable to edit: average_rainfall
2. Scroll to a row (that is, combination of parent values) in the table below.
3. Click in the appropriate box and assign a probability to each value of the chosen variable in that row.

average_humidity	average_temperature	average_rainfall=low	average_rainfall=medium	average_rainfall=high
low	low	0.3333	0.3333	0.3334
low	medium	0.3333	0.3333	0.3334
low	high	0.3333	0.3333	0.3334
medium	low	0.3333	0.3333	0.3334
medium	medium	0.3333	0.3333	0.3334
medium	high	0.3333	0.3333	0.3334
high	low	0.3333	0.3333	0.3334
high	medium	0.3333	0.3333	0.3334
high	high	0.3333	0.3333	0.3334

Right click in table to randomize.

(d) Tetrad IM

Figure 2.1: TETRAD representation of Appendix A.3

The next two lines describe the variable's possible values, and the probability of those values.

The variable `average_rainfall` shows a more complicated case, where the value of the variable is determined by other variables, not just a static probability table. The `parents` specifier of the `variable` clause lists two variables that affect `average_rainfall`. They are `average_humidity` and `average_temperature`. In this variable clause, the `probs` key-value pair is substantially expanded. The probabilities are listed in canonical order [11], as they are in TETRAD [12][1].

The Bayesian network represented in Appendix A.3 can easily be constructed in TETRAD. This is shown in Figure 2.1.

Appendix A.3 shows another important feature of HAWK, the `from` keyword. The second and third usage of the `Bayes` keyword uses a `from` clause as a sort of inheritance. A Bayesian network with a `from` clause specifying a parent *p* will inherit all of *p*'s variables, and has the opportunity to override properties

[1] TETRAD is a tool for creating and manipulating statistical models in a user-friendly and graphical environment [12]

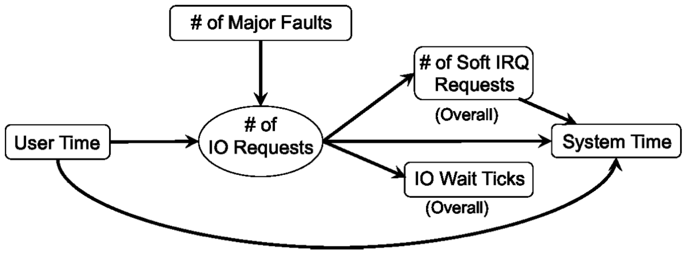


Figure 2.2: Graphical representation of linear model

of any variable. For example, the code in Appendix A.3 creates a new Bayesian network called `west`, which is exactly the same as the Bayesian network `main`, except that the probability table for the `average_temperature` variable has been overwritten. Both `main` and `west` are then assigned to subject classes.

2.1.2 Linear models

Linear models and structural equation models can be implemented in the HAWK DSL as well. HAWK defines a linear model as a collection of variables which are linear combinations of other variables and an error term E sampled from some parameterized distribution. In the example provided in Appendix A.2, the distribution is the normal distribution with mean and standard deviation parameters. The model could be represented mathematically as follows:

$$
\begin{aligned}
\texttt{user\_time} &= E(5.5, 1.0) \\
\texttt{num\_major\_faults} &= E(10.0, 2.0) \\
\texttt{num\_soft\_irq\_requests} &= 10.5 \cdot \texttt{user\_time} \\
&\quad + 29.8 \cdot \texttt{num\_major\_faults} \\
&\quad + E(1.0, 0.01) \\
\texttt{io\_wait\_ticks} &= 5.0 \cdot \texttt{num\_io\_requests} + E(1.2, 0.1) \\
\texttt{system\_time} &= 12.8 \cdot \texttt{num\_io\_requests} \\
&\quad + 2.2 \cdot \texttt{user\_time} \\
&\quad + E(0.0, 3.0)
\end{aligned} \tag{2.1}
$$

The model can also be partially represented graphically, as shown in figure 2.2 and in the paper this model is derived from [5].

The system of equations is evaluated through standard methods [4], which involves a simple algorithm that determines the value for a variable as soon as all of the variables it depends on have been evaluated. This means that there cannot be any circular dependencies. In other words, just like with a Bayesian network, the graph representation of the model must be acyclic [11].

Linear models can take advantage of the `from` keyword just as Bayesian models do, although the example in Appendix A.2 does not do so.

2.1.3 Subject Class Trees

All HAWK models contain a subject class tree, even if the tree contains only one node. Each non-leaf node of a subject class tree is exclusively treated as a combination of its children, and the proportions are determined by the specified frequency. For example, in Appendix A.2, the `databases` non-leaf node is specified as being made up of 50% `MySQL` subjects and 50% `Oracle` subjects.

A HAWK user determines what linear or Bayesian model is used for a given subject class by specifying the `bayes` or `linear` property of a leaf subject class, as shown in both Appendix A.3 and A.2.

If a user tries to take 100 samples using the `databases` subject class, 100 random subjects will be selected, 50 from `MySQL` and 50 from `Oracle`, because the `databases` subject class is specified to contain only two children and has a `freq` (frequency) of `[0.5, 0.5]`.

2.2 Implementing an extensible DSL

Writing a DSL that can be easily extensible requires some planning. The HAWK compiler is split into five phases, some of which match up with traditional compilers, while others have been modified for extensibility. A diagram of the phases is made available in Figure 2.3.

2.2.1 Lexing and Parsing

In phase 1, HAWK performs lexing and parsing. HAWK uses the ANTLR4 parser and lexer generator [10], which allows for quick modifications to the grammar to be quickly transformed into parsing and lexing code. The formal grammar for the HAWK DSL appears in Appendix A.1. The grammar is also designed so that new types of models can be easily integrated with existing subject class trees (see the `Linear` an `Bayes` keywords). In fact, the `Linear` keyword was added in exactly this way.

ANTLR4 will automatically generate Java code to visit nodes in a parsed AST, but ANTLR4 does not generate precisely the correct class structure for use with HAWK. As a result, the ANTLR4 internal AST is translated into HAWK's AST classes. This adds a slight delay to expansion, since a new, empty class (that extends a base AST class) must be manually created for each new AST type. This is fairly trivial: for example, when adding the `Linear` features, only four new classes needed to be created. Additionally, this process could be automated.

2.2.2 Static Analysis

In phase 2, HAWK performs static analysis. While traditional compilers build symbol tables and perform a linear static analysis of an AST [1], the HAWK compiler uses a modular static analysis system. After the AST is constructed, it is traversed in order, and events are published whenever a node is entered or exited. Static analysis modules can subscribe to events for specific types of AST nodes, and throw errors at anytime during the process. This allows the static analysis routines used for Bayesian models to be cleanly and completely separated from the static analysis routines used for linear models.

For example, one static analysis module, that checks to make sure that a model has a root subject class, subscribes to the enter event for the `subjclass`

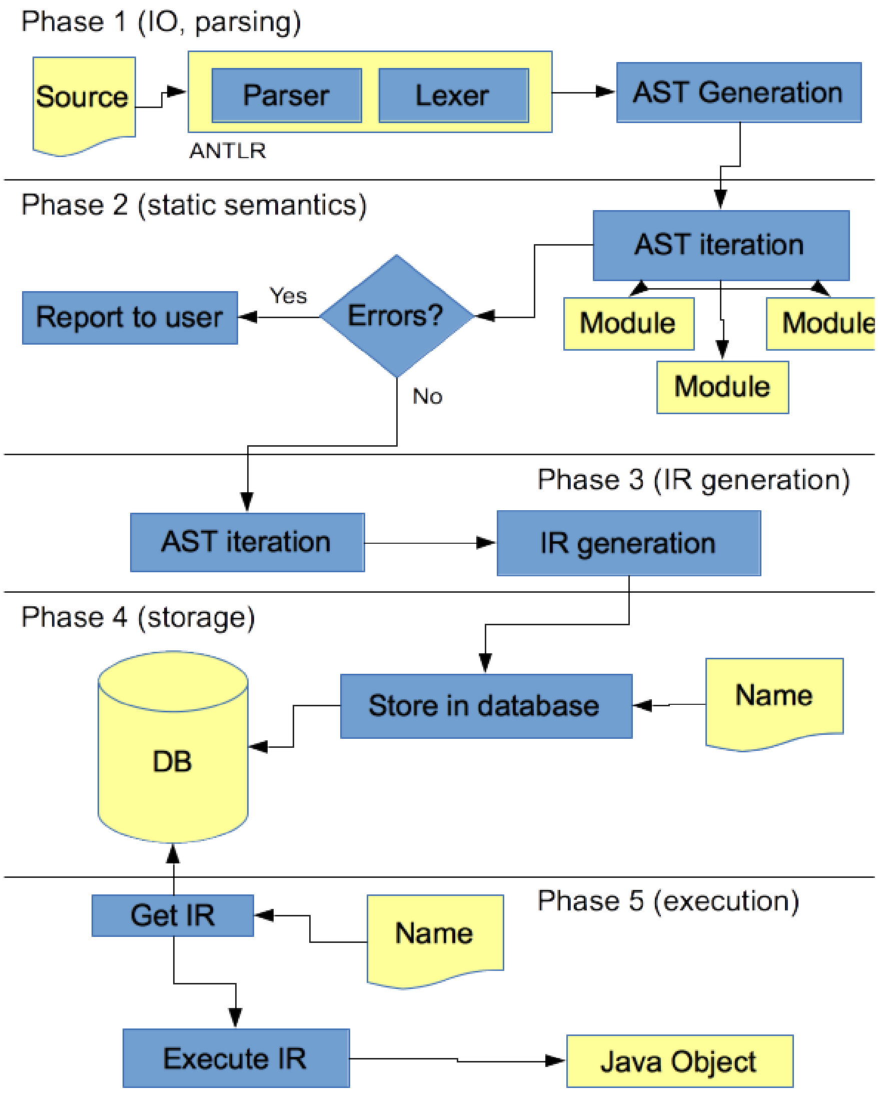


Figure 2.3: Phases of the HAWK compiler

node and ensures that it is triggered at least once.

In addition to making static analysis code modular, this approach to static analysis has the benefit of being easily multithreaded. HAWK uses Google Guava's EventBus, which can deliver events asynchronously to subscribers [3]. While HAWK is too simple to gain any performance benefits from this, this approach allows for each static analysis model to run in parallel, and thus complete much faster. It also lifts the burden of synchronization that comes with trying to multithread a traditional static analysis system because there is limited data sharing between modules and because each module will eventually process every event it subscribes to, in order.

2.2.3 IR Generation

Like many compilers [1], HAWK transforms a validated AST into an intermediate representation (IR) in phase 3. This creates isolation between the specific grammar (which is subject to be extended and changed) and the code that directly constructs and samples data. HAWK emits IR that can be executed by the HAWK virtual machine (VM), which is a stack-based interpreter that turns IR code into Java objects that support running experiments and generating data.

Generating IR code is essential to an extensible compiler because it greatly simplifies final code generation. When the grammar changes, the VM doesn't, which ensures that previously compiled models will still function. Additionally, this provides the developer with the convenience of some higher-level VM functions, as opposed to having to translate directly from AST to Java object.

2.2.4 Storage

In phase 4, HAWK stores the generated IR into a database that can be accessed by SHAWK and the HAWK domain plugin interface. Testing shows that storing the IR instead of a serialized Java object results in less data being stored in the database.

While this decision has nothing to do with making the HAWK compiler extensible, it does greatly aid in integrating AMELIE and HAWK. Having code stored in a database that is accessible to both the HAWK user and the domain interface plugins is convenient, and avoids creating multiple models with the same name.

2.2.5 Execution

Phase 5 occurs when SHAWK or the domain plugin requests an actual Java object capable of performing experiments. The IR code for a model is fetched from the database and executed in the VM, and the final Java object is returned. The final Java object returned implements an interface that the HAWK domain plugin knows about, and thus the domain plugin is able to process the request.

Some may be concerned with the performance of re-executing the IR code each time an experiment request is made. If performance becomes a problem, the domain plugin could be easily modified to cache the compiled Java object. However, testing at even very large problem sizes (over 1000 variables) shows that IR execution never takes longer than 500ms.

Chapter 3

Designing HAWK: GUI or DSL?

At first glance, a DSL may seem like an odd choice for creating a validation tool. Initially, HAWK was created as a GUI application that allowed the user to visually construct Bayesian networks using CMU's TETRAD [12]. There was immediate concern over how future versions of TETRAD would interact with HAWK, and what an upgrade path would look like. Additionally, as HAWK was further developed, it became clear that HAWK would need to be able to produce other types of models, like structural equation models and Markov chains. Since the AMELIE group decided to incrementally make HAWK's model more expressive as domains are encountered that need more expressivity, HAWK was likely to undergo a multitude of changes and evolutions.

3.1 Accommodating Future Versions of TETRAD

The GUI version of HAWK used Swing components provided by TETRAD. This became immediately complex. The main TETRAD panel assumes that it is contained within a `JDesktopPane`, an assumption that prevented HAWK from having a polished look and feel. Many small hacks and tweaks were applied, but, as a design principle, the actual TETRAD code was left unaltered. This led to occasional and difficult-to-trace mouse glitches, scroll bar failures, and other UI oddities.

The primary concern was not the creation of a GUI beyond reproach. The AMELIE group did not mind a few rough spots in their testing tool. The main complications arose when trying to design HAWK such that future versions of TETRAD could simply be "dropped in". Several strategies were developed while pursuing this goal, each with notable limitations. These strategies are applicable to any Java application that includes large portions of another project.

3.1.1 JUnit Testing

Writing JUnit tests [8] for another developer's code may sound like a particularly dull experience, but a large library of tests that check every assumption made about TETRAD proved to be incredibly useful. While these tests never provided the desired “drop in replacement”, they did quickly signal where changes in the API had been made, making it substantially easier to locate and modify HAWK code to account for changes in TETRAD.

3.1.2 Reflection-based Approaches

Several attempts were made to use Java’s reflection capabilities [6] to automatically determine and account for API changes[1]. Regardless of the approach used, at least one of several assumptions must be made:

- the name of the desired method will not change,
- the signature of the desired method will not change,
- the name of a class to be extracted from a hierarchy will not change,
- if an object of a certain type T is desired, and objects of types $S = \{t_1, t_2, ...\}$ are available, a method that takes some subset of S as parameters and returns an object of type T will return the correct result.

Various approaches using some subset of the above assumptions generated various degrees of success. Searching for methods by signature proved to be effective at dealing with changes in class hierarchies. Searching for methods by name proved to be effective at dealing with changes within a class hierarchy when the hierarchy otherwise remained the same. Other techniques did not show much promise, and no technique was able to automatically transition HAWK from TETRAD 4 to TETRAD 5.

3.1.3 Limitations

Ultimately, reflection-based techniques proved to correctly compensate for changes within TETRAD about as often as they catastrophically failed. It seems that the JUnit tests that were created and maintained are invaluable, as any failure serves to quickly pinpoint changes in the TETRAD API.

However, the JUnit tests were only as useful as they were thorough. One must test that the proper buttons appear somewhere in the component hierarchy, and that clicking on one such button produces the desired result. Essentially, every action that needs to be supported within an external piece of software needs to be tested. Obviously, most developers will find a balance between test exhaustiveness and convenience, as making tests excessively detailed has diminishing returns.

3.2 Extensibility

While the ”back-end” model code (TETRAD interaction, storing subject class trees, etc.) can be easily extended with simple additions, GUI code cannot. Creating a tree structure that contains multiple TETRAD sessions is trivial, but modifying the current GUI to represent that tree is not. Many assumptions and hacks were used to integrate the TETRAD window into the right-hand-side

[1] Reflection gives a programmer the ability to *programmatically* enumerate and examine a Java class and its methods.

of the old HAWK GUI. While wrapping another panel around the TETRAD GUI seems simple, tweaking and fine-tuning mouse events and resize events would be far from simple.

Therefore, in order to choose between a GUI and a DSL, we considered the cost of extending each.

The costs of extending a GUI include the following:

- Unit tests must be completely rewritten. Adding another layer of wrapping means that any testing code that references a specific field or button must be reworked.
- Resize events and UI tweaks must be reexamined. This may seem simple, but in practice it can require many hours of developer time.
- Polish must be reapplied. Making a GUI application look nice is not easy, and making structural changes to a GUI essentially requires that all the polish be reapplied.
- Structural changes in the model may require the creation of new GUI elements. While one would not have to create a tree from scratch, one does have to implement a `TreeModel` and test it against the GUI.

The costs of extending a DSL include the following:

- Unit tests remain exactly the same. Extensions to the DSL can be added by extending the formal grammar, which should preserve backwards compatibility. If a change does not preserve backwards compatibility, then failing test cases will serve to check how far a grammar change extends.
- There are no resize events or UI tweaks. There are tools to verify that a grammar is parsable automatically and quickly. These tools are incredibly well developed and provide useful error messages.
- Polish is automatically reapplied. When one formally specifies a grammar, one gets error message generation for free.
- Structural changes in the model are cleanly and easily represented as structural changes in the grammar.

We determined that it would much easier to extend a DSL than a GUI (some of the complications of developing an extensible compiler are discussed in 2.2), thus HAWK shifted from being a GUI to being a compiler[2].

[2] The HAWK GUI code is now being transformed into a prototype of AMELIE itself.

Chapter 4

SHAWK

During the development of HAWK, it became clear that we needed some way to test HAWK itself, without using AMELIE. Thus, we developed SHAWK, which is described in the this chapter.

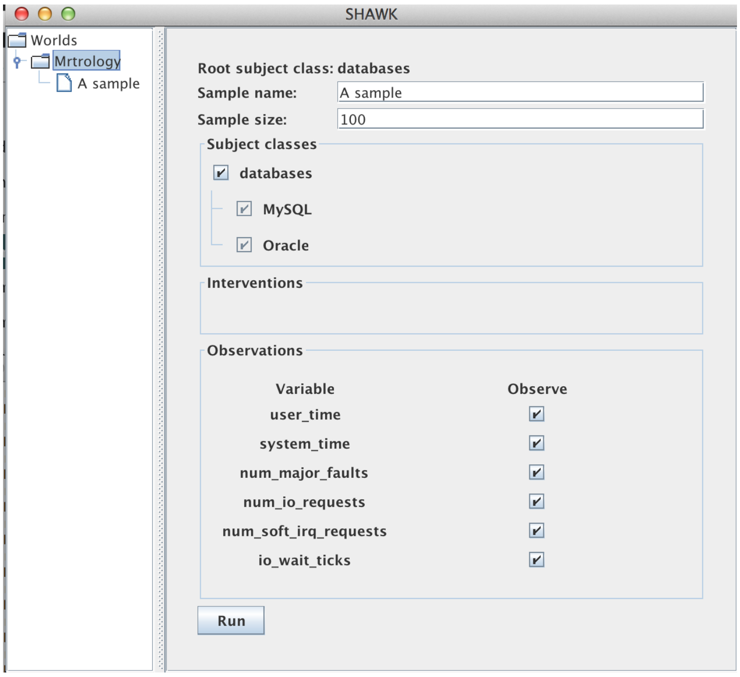
4.1 Purpose and Function

SHAWK, shown in Figure 4.1, is a companion tool to HAWK that allows a user to run experiments against a HAWK world. The main goal of SHAWK was in its development: to navigate and map the requirements for the domain plugin interface. It is worth noting that SHAWK does not actually use the domain plugin interface, but interacts directly with the HAWK compiler (specifically, phase 5).

Data sampled from the metrology model (listed in A.2) using the SHAWK tool is shown in Appendix A.4.

4.2 Implementation

SHAWK makes extensive use of Hibernate and Java Swing to provide lazy-loading and concurrency control. Unlike the domain interface plugin, SHAWK does not allow the user to specify multiple disjoint subject classes. A user can only specify one subject class at a time.



Root subject class: databases

Sample name: A sample

Sample size: 100

Subject classes

- ☑ databases
  - ☑ MySQL
  - ☑ Oracle

Interventions

Observations

Variable	Observe
user_time	☑
system_time	☑
num_major_faults	☑
num_io_requests	☑
num_soft_irq_requests	☑
io_wait_ticks	☑

Figure 4.1: The SHAWK Interface

Chapter 5

Case Study: DBMS Metrology

In order to ensure that HAWK was capable of testing AMELIE, we designed and implemented a case study using a HAWK and a prototypical version of AMELIE.

5.1 Introduction

In their paper titled “DBMS Metrology: Measuring Query Time”, Snodgrass et al. note that it is difficult to obtain both precise and accurate measurements of the time spent executing a query. Snodgrass et al. present a model supported by strong correlational analysis relating various performance measures. From this model, Snodgrass et al. developed an accurate query time measurement procedure [5].

In this case study, we implement the presented model in the HAWK DSL. Then, we use the implemented model to test a prototypical version of AMELIE. The exact methodology of our case study is given in Section 5.2.

5.2 Methodology

Eventually, AMELIE will need to be able to analyze behavior exhibited by complex systems. One such system might be DBMSes, which produce data such as query processing time, index miss counts, etc. that can be used to analyze the system’s overall behavior. Ideally, AMELIE will enable a scientist to construct an explanatory model of these systems. In order to test this capability, we created a model of a DBMS in the HAWK DSL. This model enables the production of fake data that has structural similarity to the real data that would be produced by a DBMS. Then, using only the experimental domain interface exposed, we used AMELIE to construct the best model of the generated data as possible.

We followed a procedure similar to the one described in Section 1.4.

1. “Alice” constructed a structurally accurate model of DBMS query time in the HAWK DSL.

2. “Bob” uses SHAWK to to inspect the experiment which “Alice” has made visible.

3. “Bob”, with no knowledge of the model created in the HAWK DSL, attempts to model DBMS query time in AMELIE.

4. “Bob” forms hypotheses about DBMS query time, tests them, and then forms new hypotheses.

5. Once “Bob” is satisfied with his model, he shows his model to “Alice”. Looking at both the model discovered by “Bob” and the model created created by “Alice”, “Alice” and “Bob” decide if the two are close enough, or if the differences between the models indicate a bug in AMELIE.

Note that this methodology has two important properties previously discussed in Section 1.4. First, “Bob” operates with no knowledge of the model created by “Alice”. Because of this, any model that “Bob” derives from using AMELIE must have been derived without any specific knowledge of the model.

5.3 Constructing the HAWK Model

In their paper [5], Snodgrass et al. present an accurate model for estimating query execution time. Their method was derived from the basic structural casual model shown in Figure 2.2. We (“Alice”) parameterized this model using arbitrary values, being careful to preserve the correlation between variables. The model produced is represented by both the equations in Section 2.1.2 and in the HAWK DSL shown in Appendix A.2.

We (“Alice”) compiled the HAWK DSL shown in Appendix A.2 under the name “Metrology”.

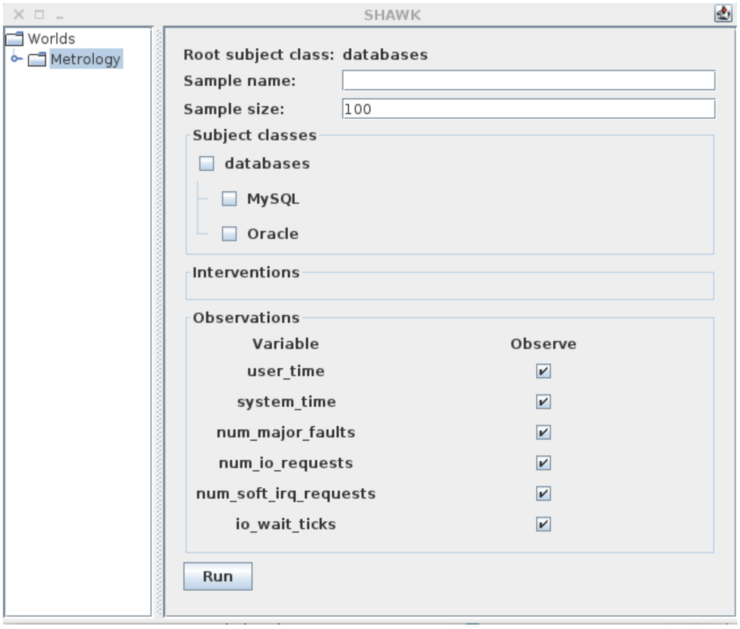
5.4 Using SHAWK

After the model was compiled, we (“Bob”) used SHAWK to inspect the experiment made available by “Alice”. SHAWK is shown in figure 5.1. “Bob” is able to see what variables and subject classes have been made available. If desired, one could choose to run a few experiments and analyze the results by hand.

5.5 Using AMELIE to Construct a Model

After the model was compiled, we (“Bob”) opened AMELIE in order to begin the scientific process. The initial, untouched AMELIE prototype window is shown in Figure 5.3. Notice that the six panels in Figure 5.3 “match up” with the panels in the AMELIE diagram (Figure 1.1) that are described in Section 1.3.1.

The upper-left panel represents model construction. In this early prototype of AMELIE, a model is limited to a listing of variables. In Figure 5.3, the list is pre-populated with values provided by the experimental domain plugin (in this case, the values specified as visible by “Alice”).



Root subject class: databases

Sample name:

Sample size: 100

Subject classes

□ databases

□ MySQL

□ Oracle

Interventions

Observations

Variable	Observe
user_time	☑
system_time	☑
num_major_faults	☑
num_io_requests	☑
num_soft_irq_requests	☑
io_wait_ticks	☑

Figure 5.1: SHAWK being used to examine an experiment

The upper-middle panel represents hypothesis generation. Currently, only simple hypotheses are supported. The panel is designed to read like a sentence. An example hypothesis, depicted in Figure 5.2, could be read as: "positive change in `user_time` causes positive change in `system_time`". After proposing a hypothesis, the hypothesis is represented textually in the box at the bottom of the panel.

The upper-right panel represents experiment design. Here, the user decides how many samples to draw from each subject class. The other panels are discussed in Section 5.6.

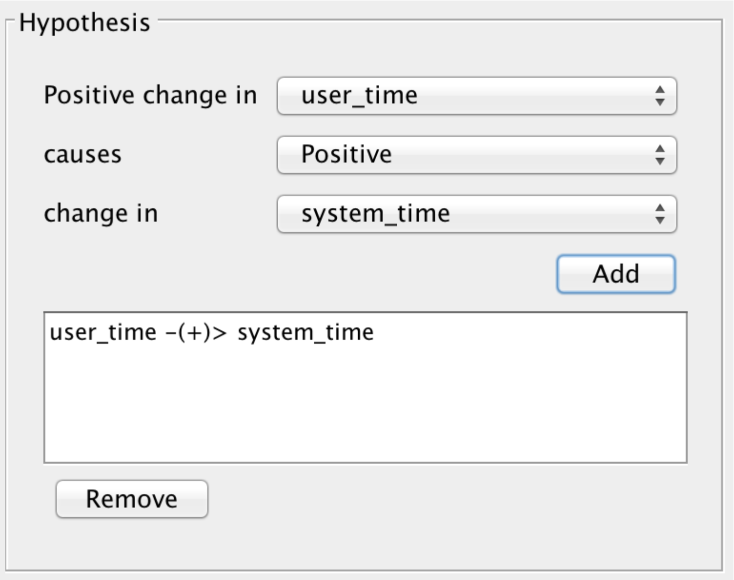
Here, we ("Bob") specify a few hypotheses about the relationship between each variable. After doing so, we can select the subject classes we want to use for an experiment testing those hypotheses. Then, we can execute the experiment by pressing the "Run" button in the upper-right panel.

5.6 Testing Hypothesis

We ("Bob") specified several hypotheses about the model, shown in the upper-middle panel of Figure 5.4. We ("Bob") then decided to take 300 samples from the `databases` subject class, and 10 samples from the `MySQL` and `Oracle` subject classes.

After clicking the "Run" button, the bottom panels will be populated, as shown in Figure 5.4.

The bottom-left panel represents data validation, which deviates from the chart (the chart specifies "model evaluation"). Here, the data from the experiment is shown in a table; this data is checked to make sure that each variable falls into an appropriate range and is an appropriate type. For our model,



Positive change in: user_time

causes: Positive

change in: system_time

```
user_time -(+)> system_time
```

Figure 5.2: An example hypothesis

AMELIE simply makes sure that the values are discrete (not categorical) and within integer bounds.

The bottom-middle panel represents hypothesis evaluation. Each hypothesis is listed, along with whether or not the hypothesis is supported or not supported with respect to the data generated. In this early prototype of AMELIE, hypotheses are evaluated based on a simple covariance test.

The bottom-right panel represents reliability analysis, which, in this early prototype of AMELIE, simply displays the total number of hypotheses and the total number of supported hypotheses.

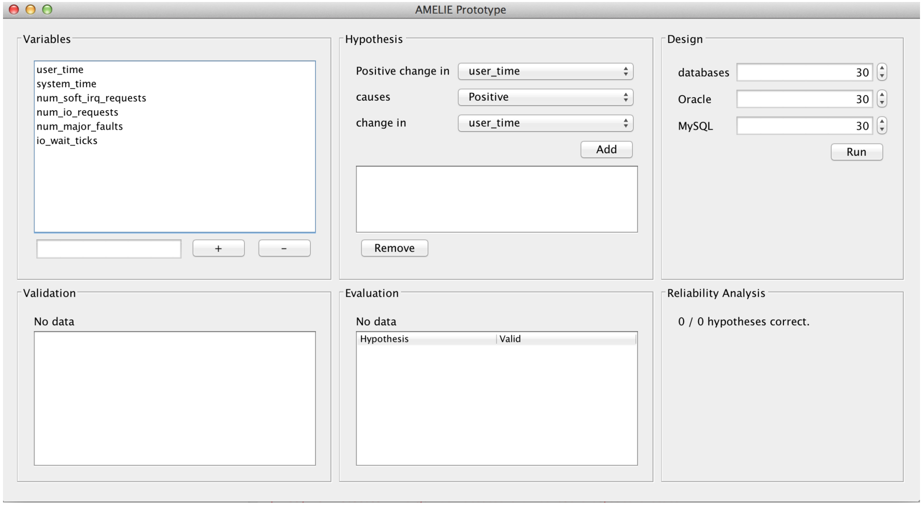
Hypotheses may be removed from the upper-middle panel by pressing the “Remove” button, and new hypotheses may be added. New data may be sampled by simply clicking the ”Run” button again. Using these simple tools, we (“Bob”) are able to quickly test hypotheses and create new ones.

5.7 Model Comparison

Because the current version of AMELIE only supports a simple covariance test, and because the model created in the HAWK DSL only has positive correlations, the model produced by AMELIE is not very advanced. Essentially, we (“Bob”) were able to learn that a positive change in each variable correlates with a positive change in every other variable.

While this may not seem like a very advanced model, it is an entirely accurate one. When we (“Alice” and “Bob”) compare the model discovered by AMELIE to the model used in HAWK, we can conclude that the model discovered by AMELIE is perfectly accurate, but did not exhibit terribly high fidelity.

The next question we (“Alice” and “Bob”) must consider is whether the lack of fidelity created by the AMELIE model indicates a bug in AMELIE.



user_time
system_time
num_soft_irq_requests
num_io_requests
num_major_faults
io_wait_ticks

Positive change in: user_time
causes: Positive
change in: user_time

databases: 30
Oracle: 30
MySQL: 30

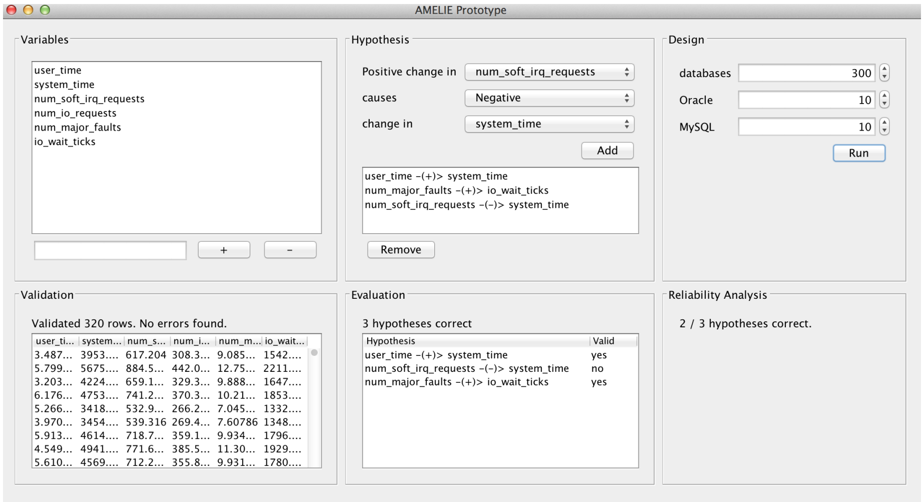
No data

No data

Hypothesis	Valid

0 / 0 hypotheses correct.

Figure 5.3: A fresh AMELIE session



user_time
system_time
num_soft_irq_requests
num_io_requests
num_major_faults
io_wait_ticks

Positive change in: num_soft_irq_requests
causes: Negative
change in: system_time

user_time -(+)> system_time
num_major_faults -(+)> io_wait_ticks
num_soft_irq_requests -(-)> system_time

databases: 300
Oracle: 10
MySQL: 10

Validated 320 rows. No errors found.

user_ti...	system...	num_s...	num_i...	num_m...	io_wait...
3.487...	3953....	617.204	308.3...	9.085...	1542....
5.799...	5675....	884.5...	442.0...	12.75...	2211....
3.203...	4224....	659.1...	329.3...	9.888...	1647....
6.176...	4753....	741.2...	370.3...	10.21...	1853....
5.266...	3418....	532.9...	266.2...	7.045...	1332....
3.970...	3454....	539.316	269.4...	7.60786	1348....
5.913...	4614....	718.7...	359.1...	9.934...	1796....
4.549...	4941....	771.6...	385.5...	11.30...	1929....
5.610...	4569....	712.2...	355.8...	9.931...	1780....

3 hypotheses correct

Hypothesis	Valid
user_time -(+)> system_time	yes
num_soft_irq_requests -(-)> system_time	no
num_major_faults -(+)> io_wait_ticks	yes

2 / 3 hypotheses correct.

Figure 5.4: An AMELIE session

Since simple covariance tests are not able to generate a more complex model, we ("Alice" and "Bob", who both paid attention in their statistics classes) can conclude that this is not a bug in AMELIE, but a feature deficiency.

5.8 Results

The case study presented above shows that HAWK can be successfully used to diagnose issues with AMELIE. In this study, HAWK and AMELIE users were able to differentiate between model accuracy issues and model fidelity issues in order to diagnose a feature deficiency. The case study also shows that the testing methodology described in Section 1.4 is viable and can successfully simulate the "knowledge gap" between a scientist and the object of study.

While the issue diagnosed may seem trivial in retrospect, the case study still demonstrates that HAWK is able to communicate with AMELIE and provide data based on user-defined models, which will doubtlessly be useful in the future.

Chapter 6

Conclusion

Overall, HAWK and SHAWK met or exceeded their requirements. The precise usefulness of either tool cannot be fully determined until AMELIE is further developed, but we can still make some observations presently.

6.1 Results

HAWK and SHAWK are shown to be fully capable of testing AMELIE at every level of development, including the early prototype presented in Chapter 5. Chapter 2 demonstrates how linear and categorical models can be defined in the HAWK DSL, and Chapter 4 shows how the SHAWK interface can sample from those models. All the tools needed to implement the validation strategy described in Section 1.4 have been created.

Because the goals of AMELIE are so broad, good testing tools are important. By enabling the development of AMELIE, HAWK represents a successful first step on the road to fully automated casual analysis. This significantly furthers AMELIE's goal of using scientific tools to analyze computer systems [9] as well as providing a tool to assist and accelerate scientific research in a multitude of domains [13].

6.2 Future Work

The experiment model discussed in Section 1.3.2 may be too constraining for some domains. For example, in a quantum physics domain, observing a variable may alter the state of that variable. We hope to expand AMELIE's capabilities as new domains are encountered, instead of trying to generalize the entire scientific process from the start.

Appendix A

Code and Data Listings

A.1 HAWK DSL ANTLR4 Grammar

```
grammar HawkGram;

prog: stmt+ ;

stmt: bayes                          # BayesStmt
        | linear                     # LinearStmt
        | subjclass                  # SubjClassStmt
        ;

bayes: 'Bayes' ID fromexpr? '{' variablelist '}' ;

fromexpr: 'from' ID ;

variablelist: variable+ ;

variable: 'variable' ID parents? '{' kv+ '}' ;

parents: 'parents' '(' ID (',' ID)* ')' ;

kv: ID '=' v ;
v: ID # IDValue
 | FLOAT # FloatValue
 | '{' ID (',' ID)* '}' # IDList
 | '{' FLOAT (',' FLOAT)* '}' # FloatList
 ;


linear: 'Linear' ID fromexpr? '{' variablelist '}' ;

subjclass: 'class' ID '{' innerclass+ '}' ;
innerclass: kv # KVofClass
                  | subjclass # InnerClassOfClass
```

```
                    ;


ID: [a-zA-Z][a-zA-Z0-9_]* ;
INT: [0-9]+ ;
FLOAT: [0-9]*'.'[0-9]+ ;
WS: [ \t\n\r]+ -> skip ;
```

A.2 HAWK DSL Example: Linear Model

```
Linear main {
        variable user_time {
                observe = true
                distrib = norm
                mean = 5.5
                dev = 1.0
        }

        variable num_major_faults {
                observe = true
                distrib = norm
                mean = 10.0
                dev = 2.0
        }

        variable num_io_requests
        parents (user_time, num_major_faults) {
                observe = true
                coeffs = { 10.5, 29.8 }
                distrib = norm
                mean = 1.0
                dev = 0.01
        }

        variable num_soft_irq_requests
        parents (num_io_requests) {
                observe = true
                coeffs = { 2.0 }
                distrib = norm
                mean = 0.5
                dev = 0.01
        }

        variable io_wait_ticks
        parents (num_io_requests) {
                observe = true
                coeffs = { 5.0 }
                distrib = norm
```

```
                    mean = 1.2
                    dev = 0.1
        }

        variable system_time
        parents (num_io_requests, user_time) {
                    observe = true
                    coeffs = { 12.8, 2.2 }
                    distrib = norm
                    mean = 0.0
                    dev = 3.0
        }
}

class databases {
        freq = { 0.5, 0.5 }

        class MySQL {
                linear = main
        }

        class Oracle {
                linear = main
        }
}
```

A.3 HAWK DSL Example: Bayes Model

```
Bayes main {


    variable wearing_raincoat {
        intervene = true
        values = { yes, no }
        probs = { 0.5, 0.5 }
    }

    variable average_humidity {
        observe = false
        values = { low, medium, high }
        probs = { 0.333, 0.333, 0.333 }
        }

    variable average_temperature {
        observe = true
        values = { low, medium, high }
        probs = { 0.333, 0.333, 0.333 }
        }
    variable average_rainfall
    parents (average_humidity, average_temperature) {
        observe = true
        values = { low, medium, high }
        probs = { 0.333, 0.333, 0.333,
                  0.333, 0.333, 0.333,
                  0.333, 0.333, 0.333,
                  0.333, 0.333, 0.333,
                  0.333, 0.333, 0.333,
                  0.333, 0.333, 0.333,
                  0.333, 0.333, 0.333,
                  0.333, 0.333, 0.333,
                  0.333, 0.333, 0.333 }
    }
}

Bayes west from main {
    variable average_temperature {
        probs = { 0.2, 0.4, 0.4 }
    }
}

Bayes east from main {
    variable average_temperature {
        probs = { 0.4, 0.4, 0.2 }
    }
}

class usa {
     freq = { 0.5, 0.5 }

      class east {
            freq = { 0.3, 0.7 }
            class boston {
                  link_visible = false
                  bayes = east
            }

            class nyc {
                  bayes = east
            }
      }

      class west {
            bayes = west
      }
}
```

A.4 Metrology Data

user_time	system_time	num_soft_irq_requests	num_io_requests	num_major_faults	io_wait_ticks
5.5625677	5995.9507	935.35895	467.431	13.692482	2338.3452
3.4872932	4116.587	642.6784	321.0914	9.512383	1606.6946
6.107203	3752.416	584.8264	292.16888	7.6191006	1461.9882
4.7216887	4012.004	625.86505	312.6842	8.795301	1564.5656
3.7462072	3989.9043	621.26996	310.38702	9.06199	1553.0005
4.6200404	4076.5034	635.5762	317.54617	8.994296	1588.8329
4.0384793	3628.669	566.1139	282.81403	8.0340605	1415.2156
4.300033	4557.1655	711.69855	355.61087	10.384182	1779.1432
5.7332935	4066.2856	633.69525	316.59973	8.570033	1584.1956
4.7857704	4540.9033	708.3361	353.915	10.156842	1770.7606
4.8017826	4021.848	627.5197	313.5036	8.795212	1568.6982
5.9082108	3975.1292	620.09656	309.7947	8.280253	1550.2407
5.831276	4794.2593	747.1861	373.3463	10.440883	1867.9811
5.0885	4939.164	771.07666	385.28223	11.102222	1927.5942
4.786162	5326.9463	831.52313	415.50745	12.222858	2078.8223
6.6388345	4319.1787	673.48096	336.48685	8.91905	1683.6306
4.0768127	4914.5757	767.7597	383.62048	11.403119	1919.0034
5.4956684	4995.105	779.3058	389.40073	11.097304	1948.068
5.0446773	4555.5015	710.3521	354.92676	10.10002	1775.8998
4.854419	5041.2065	787.00916	393.25327	11.452579	1967.5436
5.3501463	5312.4062	828.7054	414.10483	11.977376	2071.819
5.8807726	3308.9163	515.2786	257.39346	6.5317283	1288.1538
6.448221	3465.6326	539.64197	269.57294	6.7411327	1349.0381
6.666109	4750.9624	740.39435	369.95178	10.031879	1850.9818
6.2450247	5300.022	825.9033	412.70142	11.615298	2064.5593
5.2850003	6154.859	961.1476	480.31784	14.222753	2402.7156
5.184474	4414.844	688.85474	344.1836	9.689387	1722.114
6.0469556	4961.51	773.3723	386.43665	10.8037405	1933.4927
6.049392	3182.123	496.15674	247.82149	6.1506243	1240.362
4.9871254	4444.0156	693.61145	346.55392	9.8385515	1734.0029
6.034366	5660.133	882.6834	441.0861	12.641324	2206.7163
4.8388605	4635.9707	723.0793	361.29672	10.38518	1807.6073
3.9859612	5050.865	788.1827	393.83813	11.777694	1970.3978
4.620126	3508.3257	547.697	273.59943	7.5198994	1369.1754
6.423861	5591.284	871.44244	435.4835	12.316348	2178.6619
6.6981397	5541.8633	864.4398	431.972	12.101948	2161.137
3.1830409	5393.265	842.3541	420.92865	12.970346	2105.8982
5.452546	5749.4395	897.3222	448.40292	13.092837	2243.359
4.292783	4576.401	713.8326	356.66257	10.421572	1784.5093
4.468278	3370.1455	525.6753	262.582	7.203528	1314.1284
5.3639364	5122.398	798.7377	399.11472	11.469025	1996.8445
4.882136	4924.534	768.5077	384.0027	11.131659	1921.0883
6.2148046	5076.373	790.8335	395.16336	11.036693	1977.237
5.764275	4454.1973	694.1575	346.822	9.57368	1735.2626
4.4891486	5520.8965	862.32465	430.90747	12.844368	2155.7986

4.350147	3579.1882	559.1817	279.34	7.806723	1398.1053
5.2970223	5811.075	906.58795	453.04974	13.30346	2266.3677
6.0599637	5956.624	929.2576	464.3745	13.414337	2323.0483
5.222454	3634.2668	567.5357	283.52042	7.6400843	1418.6199
5.5518084	5203.9126	811.78784	405.64517	11.622747	2029.3707
6.2367616	4475.4136	697.1454	348.3233	9.45737	1742.9364
5.7919683	3154.6086	491.48273	245.49095	6.1639657	1228.7098
6.0341587	5751.706	897.765	448.62802	12.894904	2244.4114
5.284748	4448.2314	693.32336	346.41116	9.729177	1733.3838
4.161493	3812.257	594.7264	297.1186	8.470395	1486.7919
5.5551805	4886.45	762.30505	380.90158	10.791129	1905.6251
6.853022	4624.8975	720.7458	360.12323	9.636576	1801.9186
5.7716026	6270.044	978.2657	488.8786	14.338191	2445.6436
4.188584	3593.3066	560.4972	280.00342	7.8867598	1401.2012
4.660884	4223.317	658.3129	328.9089	9.361556	1645.8992
7.720205	5380.8164	837.59717	418.55182	11.291766	2094.0742
5.1317196	4658.2217	726.76227	363.12823	10.34384	1816.9194
6.33852	5770.8823	900.35175	449.9153	12.830961	2250.6482
4.7832007	4393.0513	685.1652	342.32855	9.768505	1712.7532
3.9210694	4766.791	743.40643	371.45297	11.049798	1858.5948
5.628235	4672.9956	728.7098	364.1033	10.202037	1821.611
4.56311	4503.706	703.0745	351.28937	10.146561	1757.8237
6.6407766	6888.759	1073.8331	536.6671	15.635901	2684.5784
5.0801897	4820.681	751.70325	375.6048	10.780376	1879.1344
4.8409925	3956.747	617.07056	308.27594	8.605521	1542.5739
3.9580355	4022.099	627.43134	313.46704	9.091102	1568.4766
4.8791704	5631.843	878.6853	439.08813	12.982364	2196.5417
4.994085	4699.598	732.5044	366.00867	10.489131	1831.2871
3.9902897	4752.216	740.8228	370.16528	10.981904	1851.9745
5.038764	6646.752	1037.1356	518.32404	15.585162	2592.776
4.711552	3973.2473	620.08685	309.78973	8.7028885	1550.4182
5.6936417	4603.8384	717.85315	358.6797	9.997129	1794.4962
4.023219	3331.6252	520.26483	259.88208	7.2697945	1300.5806
5.380493	5382.625	839.127	419.32352	12.142326	2097.7192
6.7314863	5226.1655	815.03595	407.2632	11.2614	2037.6124
6.7086506	5452.8945	850.65454	425.08084	11.866732	2126.4768
4.8099294	5461.444	853.1279	426.31314	12.577367	2132.6675
5.571152	3533.9028	550.54614	275.02258	7.2329154	1376.285
5.2361307	4177.1406	651.3458	325.4284	9.041798	1628.1405
4.1578736	3178.5256	495.46	247.47739	6.805593	1238.5879
6.444685	5605.252	874.0995	436.79544	12.352834	2185.1162
4.947091	4023.0825	627.477	313.48788	8.743244	1568.5969
4.5404882	3475.2134	542.3674	270.92957	7.4584675	1355.8069
6.3780518	6267.8486	977.73267	488.62134	14.115547	2444.216
5.7979035	4025.831	626.7765	313.1375	8.430636	1566.9723
5.745349	3267.728	508.83646	254.1807	6.471661	1271.7849
6.628685	6450.912	1006.0541	502.7755	14.502126	2515.2166
6.99909	4318.118	672.2758	335.8908	8.7721815	1680.7544
5.189967	4647.563	725.51685	362.5113	10.3026085	1813.6632
6.320595	5033.9336	784.7713	392.13443	10.898609	1961.7925

6.1959357	4301.7886	670.1022	334.80392	9.017987	1675.157
6.16111	4928.4004	769.0579	384.27728	10.690654	1922.6649
7.5384045	4028.2385	627.60046	313.54996	7.831871	1569.0271
3.0970092	3584.916	559.89276	279.70203	8.261674	1399.4678
4.9084616	3404.3394	530.11554	264.8118	7.1235676	1325.1642
4.565528	3933.6692	613.50195	306.4987	8.642702	1533.7577
6.0223565	4509.918	703.70795	351.6025	9.642589	1759.1987
4.312954	4796.325	748.4698	373.99063	10.996874	1871.3391
6.3578053	4182.012	651.4134	325.45593	8.647887	1628.567
5.2935452	4477.6675	698.30835	348.9014	9.809582	1745.6863
5.147583	3997.539	623.5116	311.50867	8.606039	1558.7351
5.23452	5242.191	818.30927	408.90704	11.843852	2045.7734
5.499662	4010.1284	625.41254	312.45282	8.513612	1563.437
5.818197	2791.2156	435.1695	217.33434	5.20921	1087.8878
7.633819	3960.8953	616.3774	307.94397	7.610503	1540.8468
4.62208	5490.6636	857.7114	428.60126	12.72051	2144.1067
6.7507133	3782.931	588.73535	294.12375	7.4574547	1471.7549
6.1876884	4910.6123	766.92017	383.203	10.645349	1917.228
5.4415183	4509.305	703.2636	351.38763	9.841372	1758.0875
4.959819	3125.2678	487.49896	243.49983	6.3900566	1218.687
5.061334	5144.6367	802.19965	400.84702	11.634158	2005.3702
6.280822	4008.4956	625.06433	312.2729	8.232604	1562.7667
4.544048	3976.2952	620.45245	309.97632	8.7670765	1551.0754
4.7539244	5158.6865	803.66376	401.5732	11.766922	2009.0865
5.9666104	4863.539	758.2526	378.87796	10.578073	1895.5884
6.52931	4939.281	769.9733	384.74353	10.577084	1924.9982
5.266319	4209.068	656.05334	327.7802	9.110173	1639.8937
6.896801	5643.88	880.4135	439.9545	12.300848	2200.8394
7.0388103	3923.2026	610.83374	305.16418	7.726232	1527.0189
5.9998465	5850.703	912.79364	456.14038	13.158563	2281.85
6.506529	6618.824	1032.2866	515.8885	14.985581	2580.445
5.36405	3507.3962	547.3883	273.44366	7.2531195	1368.5981
4.4354525	3617.7227	564.1754	281.83154	7.861099	1410.2571
5.198028	3947.5654	615.8498	307.6672	8.459096	1539.526
6.780838	4672.483	728.3347	363.91544	9.789127	1820.8105
5.9427395	3766.0544	586.4145	292.9506	7.703189	1465.8761
6.2245483	3212.2583	500.23807	249.86818	6.158076	1250.3885
8.25513	4899.1323	763.7405	381.61703	9.86347	1909.3885
6.850243	4756.116	741.3033	370.40076	9.982955	1853.1348
5.6496487	4464.9927	696.08875	347.79623	9.6468115	1740.1846
6.709769	3839.7078	597.64	298.57062	7.621032	1494.1534
6.940765	3837.987	596.4509	297.97275	7.5196085	1491.0656
5.783805	4554.6665	709.7331	354.61743	9.828769	1774.3868
3.0871902	4517.446	705.5252	352.51596	10.708274	1763.8562
5.901614	3488.8215	543.9914	271.74854	7.006375	1360.0522
6.1426883	5167.1025	805.38513	402.4432	11.307048	2013.3074
2.7678938	3502.6138	547.0789	273.28043	8.161953	1367.5684
5.303328	4054.9722	632.5057	316.00354	8.702092	1581.1051
6.1400237	5733.386	893.66626	446.58405	12.789745	2234.1372
7.495324	4686.8115	730.4914	365.00342	9.573735	1826.3499

4.7115374	4206.5137	655.78046	327.64108	9.30097	1639.2588
5.891903	5438.3306	847.78015	423.64658	12.106972	2119.2732
5.105165	4780.5977	745.07764	372.28964	10.660883	1862.5881
5.9792876	5712.1533	891.14435	445.32083	12.803219	2227.7258
5.4168463	3533.6526	550.8487	275.17255	7.292448	1377.2087

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Ivo Balbaert. *The Way To Go: A Thorough Introduction To The Go Programming Language*. iUniverse, Inc., 2012.

[3] Bill Bejeck. *Getting started with Google Guava: write better, more efficient Java, and have fun doing so*. Community experience distilled. Packt Publ., Birmingham, 2013.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[5] Sabah Currim, Richard Snodgrass, Young-Kyoon Suh, Rui Zhang, Matthew Wong Johnson, and Cheng Yi. Dbms metrology: Measuring query time. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 421–432, New York, NY, USA, 2013. ACM.

[6] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action Series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[7] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[8] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[9] Clayton T. Morrison and Richard T. Snodgrass. Computer science can use more science. *Commun. ACM*, 54(6):36–38, June 2011.

[10] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[11] Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, March 2000.

[12] Richard Scheines, Peter Spirtes, Clark Glymour, Christopher Meek, and Thomas Richardson. The TETRAD project: Constraint based aids to causal model specification. *Multivariate Behavioral Research*, 33:65–117, 2009.

[13] Richard Snodgrass. Automated causal analysis. `www.cs.arizona.edu/projects/focal/ergalics/autocausalanalysis.html`. Accessed: 2014-05-03.