A FRAMEWORK FOR THE AUTOMATION OF DISCRETE-EVENT SIMULATION EXPERIMENTS

by

Bryan C. Ward

A Thesis

Presented to the Faculty of

Bucknell University

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering & Bachelor of Arts in Mathematics with Honors in Computer Science and Engineering

May 11, 2011

Approved:

Dr. L. Felipe Perrone
Thesis Advisor

Dr. Stephen Guattery
Chair, Department of Computer Science

Acknowledgments

This thesis would not have been possible without the support of many people in my life. Before I begin, I would like to take this opportunity to thank these people for their support through my Bucknell career:

- Dr. L. Felipe Perrone, for his help and guidance throughout my career at Bucknell. Felipe has mentored me in research and life since my freshman year. Thanks for everything Felipe.
- Peg Cronin, for her excellent help and support during the writing process. Through my regular meeting with Peg I have learned to think more critically about my own writing. Thanks Peg for teaching me so much about writing, and making the thesis writing process bearable, even fun at times.
- Andrew Hallagan ('11), for his collaboration on SAFE. Andrew's project works in harmony with my own and I have learned a great deal from Andrew through our collaboration. Good luck after graduation Andy.
- Heather Burrell for her patience and support throughout the writing process, particularly during times of elevated stress and/or frustration. Thanks for always being there for me.
- My family and friends who supported me throughout my thesis as well as my career at Bucknell.

Contents

List of Tables

List of Figures

Code Listings

Abstract

Simulation is an important resource for researchers in diverse fields. However, many researchers have found flaws in the methodology of published simulation studies and have described the state of the simulation community as being in a *crisis of credibility*. This work describes the project of the Simulation Automation Framework for Experiments (SAFE), which addresses the issues that undermine credibility by automating the workflow in the execution of simulation studies. Automation reduces the number of opportunities for users to introduce error in the scientific process thereby improving the credibility of the final results. Automation also eases the job of simulation users and allows them to focus on the design of models and the analysis of results rather than on the complexities of the workflow.

Part I

Introduction and Background

Chapter 1

Introduction

Computer simulation is a valuable tool to people in many disciplines. This thesis develops a framework which aids simulation users in conducting their simulation studies to ensure their results are accurate and reported properly. This work is best understood with a background in computer modeling and simulation, as well as proper simulation methodology.

1.1 Modeling

In numerous applications ranging from engineering to the natural sciences to business applications, people seek to quantify the behavior of different real world processes and phenomena. When such a process or phenomenon is studied scientifically, it is called a **system**. Assumptions are often made about the behavior of these systems which allow for mathematical and scientific analysis. These assumptions comprise a **model** and are composed of mathematical or logical descriptions of the behavior of the system. [26]

Models are coupled with performance **metrics** which are used to quantify different aspects of the behavior of a system. Models logically or mathematically relate metrics with input parameters called **factors**. The specific value given to a factor is called a **level**. For example, when investigating a vehicular traffic system, the rate with which cars arrive at a traffic light is a factor, while the numeric value of 10 cars per minute is

a level. Additionally, a model can be composed of many different **sub-models** which themselves describe the behavior of a smaller or simpler **sub-system**.

If a system or model is simple enough, mathematical analysis can be used to explicitly solve for the value of different performance metrics. These solutions are known as **analytic solutions**, and are the ideal way to quantify the behavior of the system under investigation. Using an analytic solution, one can more easily isolate effects of different factors and find optimal factor-level combinations. These results can inform scientific developments as well as engineering and business decisions.

Solving for performance metrics analytically can be challenging, if not impossible, for more sophisticated and complex models. In such cases, engineers often study the system by **simulating** the behavior of the system using computers. In the absence of an analytic solution, simulation can be employed to investigate the system under many different sets of inputs.

1.2 Computer Simulation

Traditionally, real world experiments are conducted to test how systems behave under different circumstances. Often times such experiments can be expensive, time consuming, dangerous, or hard to observe. With modern computers and software, these systems can be evaluated using computer simulations. Simulation results, similar to analytic solutions, can also be used to inform scientific advancements, engineering design decisions, or business strategies. Further, simulations can sometimes be executed faster than real time, helping to provide insight into future events or phenomenon.

A discipline which enjoys extensive use of simulation is the field of computer networks. Take, for example, a researcher developing new wireless networking protocols for vehicles traveling at high speeds on roads and interstates. After developing such a protocol, the researcher would want to evaluate its performance. Testing such a protocol can be very costly, particularly when investigating how the network will fare with hundreds of vehicles traveling at high speeds over great geographic distances. In such a case, testing these new protocols using computer simulation can reduce the cost of testing and reduce development time. This is just one example of how simulation allows for a more efficient engineering process.

Another application of computer simulation is in molecular biology. A project

called Folding@Home, uses computer simulation to investigate how proteins fold. The Folding@Home project distributes simulation execution across volunteer computers throughout the world to accelerate computationally expensive simulations. The results of these simulations are used to understand the development of many diseases such as Alzheimer's, ALS, and many cancers. [14]

With advancements in computer hardware over the last 50 years, computer simulation has become an increasingly powerful tool in science and engineering. Computer simulation has aided researchers in developing many of the technologies and business strategies that power our society today. As computer hardware continues to improve, simulation will become an even more powerful tool for people in a wide array of disciplines and will play a critical role in future scientific developments, particularly as engineered systems become increasingly complex.

1.3 Discrete-Event Simulation

There are many ways to create a model in computer software. One such paradigm often used to investigate time-varied systems is called **Discrete-Event Simulation**. In such simulations, the system is described and modeled through a chronological sequence of events. These events drive the behavior of the simulated system.

In a discrete-event simulation, the simulator maintains a **simulation clock** which keeps track of time in the simulated environment. Events which change the internal state of the simulation are scheduled on the **event list** or **event queue**. During the execution of an event, new events can be added to the events list. When an event is finished processing, the simulation clock is advanced to the next event in simulated time. [13]

Discrete-event simulation is often used to investigate systems with random behavior, known as **stochastic processes**. The simulation of stochastic processes requires the generation of random numbers using a **Pseudo-Random Number Generator (PRNG)**, which produce a deterministic stream of numbers which appears to be truly random. A PRNG must be **seeded** with a starting value, which is used in a mathematical algorithm that produces the subsequent values sequence. The same stream is recreated time after time with the same starting PRNG seed. Simulations of stochastic processes which employ PRNGs to model their behavior are said to be **stochastic simulators**. [26]

A classic application of discrete-event simulation lies in queueing theory, which is a well-established field of Operations Research. An example of an application of queueing theory is the study of lines in a shopping mall store. Neither the rate with which customers enter the queue nor the amount of time it takes for the cashier to check a customer out are constant, or **deterministic**. One can, however, assume that these times are described by **random variables** and construct a discrete-event simulation to produce estimates of the average time a customer waits in line. In this case, the probability distribution of arrivals is a factor, and distribution itself, say Poisson, is a level. The parameter of the random distribution, in the case of Poisson, λ, is also a level. The simulation model uses these levels for the associated factors to schedule events such as a new customer entering a line and a cashier finishing checking someone out.

1.4 Simulation Workflow

Once a computer simulation has been built, simulations can be run on many different inputs. Simulators can therefore be used to conduct **simulation experiments**, in which the factors of the simulation are varied to investigate their effect on performance metrics. Each unique input set of factors and associated levels in such an experiment is called an experimental **design point**. For example, a design point in the context of a shopping mall store line is a complete set of levels for all the factors in the model, such as customer arrival rate and service rates.

A simulation experiment is composed of a set of design points to run. The act of choosing the particular set of design points to explore is called **experimental design**. There are many experimental design techniques which users can employ to understand the effect of different factors on the performance metrics with less time spent in simulation execution. These techniques seek to constrain the **experimental design space**, or the set of design points which are executed during the experiment's execution. Several of these techniques are described in more detail in Section 2.

Once the experimental design space is defined, one can start to execute simulations to collect data. When using a **stochastic simulator**, it is best to run many simulations for each design point, each with a different PRNG seed and to compute averages, which are **point estimates** of the metrics collected. This ensures that results are not biased by a particular stream of random numbers. Using the samples of these metrics and a chosen **confidence level**, one can compute **confidence intervals**, which give

a better estimate of the true value of the corresponding metrics.

The results obtained from the simulation runs may be saved in persistent storage to be analyzed upon the completion of the simulation experiment. The analysis of this body of data may test hypotheses or lead to conclusions about the system. Simulation results are analyzed using many different statistical techniques.

There are many complexities associated with running an experimental simulation study. The aforementioned steps must be followed very carefully, and furthermore, one must take great care in reporting results. Just as in any other scientific process, simulation results must be reproducible and independently verifiable. Simulation users must therefore take precaution in reporting not only their results properly, but also details of their experimental process so that an independent third party can replicate their results. When proper simulation workflow is put in practice, and experimental methodology and results are reported properly, a simulation study is **credible**.

1.5 Common Problems in Simulation Studies

Conducting a complete and thorough simulation experiment is an extensive process. There are countless opportunities for a user to make a mistake in the proper simulation workflow. Many researchers [25, 28] have shown that these mistakes in proper simulation workflow lead to credibility issues. Furthermore, if the experimental methodology is not reported accurately such that others can reproduce the experiment, the credibility of the results are compromised even if a simulation study is conducted properly.

Once simulation results have been collected, proper statistical methods need to be applied to ensure that the statistics for the experiment accurately portray the results. Often times simulation users make naïve assumptions in their statistical analysis and methodology which can lead to biased results. For example, users often assume that their results are **Independent and Identically Distributed (IID)**, which allows them to use simple, standard formulas to calculate the mean and variance. It is not always the case though that the samples are IID, and consequently, the reported results are often biased. [25]

Just as in any other statistical study, it is best to observe many observations in a sample to estimate different values more accurately. Consequently, in simulation experiments it is best to run many simulations with different PRNG seeds to collect

many observation. Managing the results from hundreds to thousands of simulations can be a daunting task. Furthermore, particularly in large experiments, a great deal of time can be spent running these simulations and managing their results. Only running a single simulation run for each design point is a very common mistake in simulation methodology seen in past and current literature [25].

Another common oversight in the analysis of simulation results is the lack of computed confidence intervals. Simulations are a means to estimate certain population statistics for complex systems, and it is important in any statistical study to report the confidence of computed results. This problem is compounded in the case in which a single simulation is executed per design point, and there is a single statistical sample. To ensure highly credible results, many simulation runs should be executed per design point, and the associated confidence interval should be reported with any statistics.

Simulation users often also forget to consider the **transient** or "warm up period" of a discrete-event simulation. Many of the initial results collected during the transient are biased as the system approaches its **steady state**, which is most often what simulation users are most interested in studying. Therefore, the results collected during the transient should be discarded. This process is called **data deletion**, and is important to ensure that results are not biased. In network simulation, many studies do not include data deletion, and those which do rely on arbitrary choices for the length of the transient. According to Kurkowski et al. [25], the vast majority of the past and current literature using simulation to study Mobile Ad-Hoc Networks (MANETs) do not include any discussion of data deletion.

These problems in simulation workflow and analysis are further compounded by improper reporting of experimental results and methodology in many simulation studies. This has led Pawlikowski [29] to describe the current state of the network simulation community as a "crisis of credibility."

1.6 Enhancing Usability and Credibility

Kurkowski et al. [25] explained that many of the steps necessary in proper simulation methodology and statistical analysis are often skipped or conducted carelessly, thereby compromising the credibility of the results. Many of these steps in proper simulation workflow can be automated through computer software **automation tools** to ensure that results have a higher level of credibility. Perrone et al. [30] claimed that *"The level*

of complexity of rigorous simulation methodology requires more from [the simulation user] than they are capable of handling without additional support from software tools."

Mistakes in statistical analysis can be easily avoided through the use of software tools. Statistically inclined simulation developers can develop tools which walk a simulation user through all of the steps in proper statistical analysis. In this manner, all statistical results which the tools help the user to discover are ensured to be correct. For example, tools can ensure that confidence intervals are always provided for each of the metric estimators. These tools can be extended to help users generate figures ensuring all axes are labeled, and confidence intervals are plotted.

Large simulation studies can include thousands of simulations which need to be executed. Such simulation studies can take thousands of processor hours to execute. To accelerate this process, independent simulations can be executed concurrently on many processors on different physical computers. While this can reduce the simulation time, it also incurs more administrative overhead to partition the simulations to run on many processors and aggregate results. Furthermore, this process introduces opportunities for the human user to compromise the integrity of their results. Automation tools can be used to manage the execution of simulation runs across a network of computers to reduce simulation time.

Automation tools ensure the credibility of simulation results while easing the simulation workflow. This allows users to focus their efforts on modeling the system or understanding results instead of managing simulation execution. Computer simulation automation tools make computer simulation more valuable to the research community. My thesis is thus:

Thesis Statement

The current state of the simulation community has been described as a crisis of credibility. Automation tools address this issue by automating the processes in which common mistakes are made to ensure the credibility of results. Furthermore, automation tools can ease the simulation workflow for users to allow them to focus on their science instead of the simulation workflow. I have developed a framework which can be used to automate many of the requisite steps in proper simulation workflow, thereby ensuring the credibility of collected results. This framework represents a significant contribution to the simulation community which will help users produce more credible results.

This thesis is organized as follows. The remainder of Part I, discusses background information relevant to my project. Part II describes the Simulation Automation Framework for Experiments (SAFE), which represents my main contribution. Part III, looks at applications of SAFE and concludes the thesis.

Chapter Summary

Computer simulation is a valuable tool in many fields. To use a simulator properly requires careful attention to detail when conducting a simulation experiment. When users are not careful, their results can easily be compromised, leading to results which are not credible. To fully realize the utility of computer simulations, automation tools are required to guide a user through the steps in proper simulation workflow to ensure credible results. Chapter 2 describes several ways in which experiments can be designed to investigate relationships between performance metrics and factors.

Chapter 2

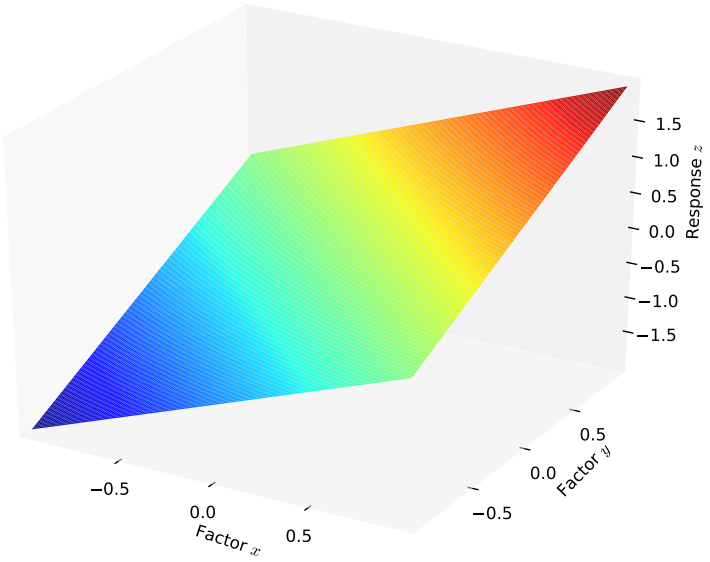
Design of Experiments

Simulation experiments are often conducted to evaluate relationships between factors and performance metrics, sometimes called **responses**. The set of responses for many design points is known as the **response surface**. Response surfaces can take on many shapes and forms as can be seen, for example, in Figure 2.1. Experiments can be designed to investigate relationships between factors and their effect on a response surfaces. Many experimental design techniques exist to help users evaluate the differences in responses from different factors. These techniques can reduce the number of simulations needed to understand these relationships.

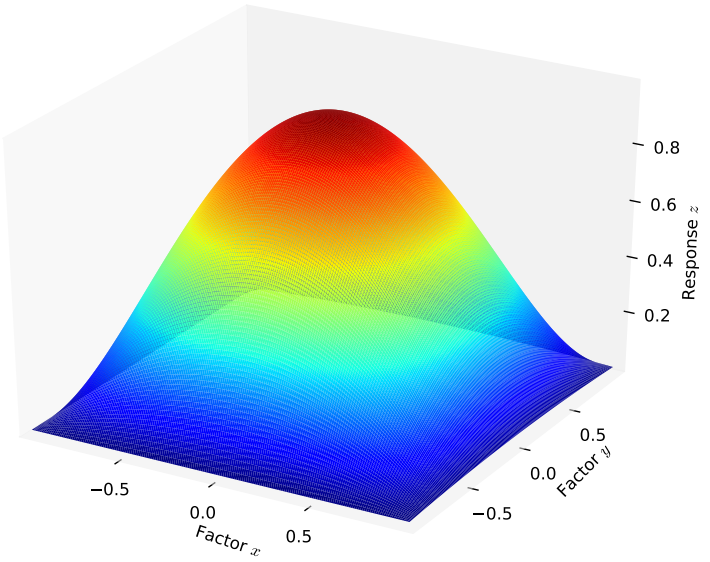
2.1 2^k Factorial Experimental Design

A simple technique often used to evaluate which factors have the largest effect on the response is the 2^k factorial experimental design. In this design, a low and a high level are chosen for each factor and permuted to compute all of the design points in the experiment. These low and high values are often coded $+1$ and -1 respectively. An example of a 2^k factorial design with 3 factors can be seen in Table 2.1. In a 2^k factorial design, there are 2^k design points in the experiment where k is the number of factors under investigation.

Using this experimental design technique, one can isolate which factors play the largest role in the response. For example, to calculate the effect of factor 1, denoted



(a) An example response surface for two factors, x and y with response $z = x + y$.



(b) An example response surface for two factors, x and y with response $z = (x + 1)(x - 1)(y + 1)(y - 1)$.

Figure 2.1: Examples of different response surfaces.

Design Point	X_1	X_2	X_3	Response
1	-1	-1	-1	R_1
2	$+1$	-1	-1	R_2
3	-1	$+1$	-1	R_3
4	$+1$	$+1$	-1	R_4
5	-1	-1	$+1$	R_5
6	$+1$	-1	$+1$	R_6
7	-1	$+1$	$+1$	R_7
8	$+1$	$+1$	$+1$	R_8

Table 2.1: 2^3 Factorial Design example.

e_1, in an 2^3 factorial experiment, we can compute the following function of responses $R_1, \ldots, R_8$

$$e_1 = \frac{(R_2 - R_1) + (R_4 - R_3) + (R_6 - R_5) + (R_8 - R_7)}{4}.$$

Similar techniques can be applied to evaluate the effect of other factors on the response. [26]

A 2^k factorial design is best suited for models where the response can be well-fit with a linear model. For example, in Figure 2.1a, there is a linear relationship between the response, z, and each of the factors x and y. This response surface can be easily investigated with a 2^k factorial experiment. By contrast, the response surface in Figure 2.1b does not exhibit a linear relationship between the factors and the response. In this case, a 2^k factorial design can yield misleading results. For example, if the points $\{(-1,-1),(-1,1),(1,-1),(1,1)\}$ were chosen, the perceived effect of both x and y would be 0 as can be seen in Figure 2.2a.

In a 2^k factorial design, as the number of factors under consideration grows, the experimental design space grows exponentially. For example, with only 10 factors, there are over 1000 design points which would need to be run. If each simulation takes a minute to run and 30 replications of each design point are executed, this experiment could take three weeks.

2.2 m^k Factorial Design

A natural extension to the 2^k factorial design is what is known as an m^k factorial design. In this case, m levels are chosen for each factor, and all permutations of factor level pairs are computed to determine the experimental design space. In such an experiment, there are m^k design points where again k is the number of factors under investigation.

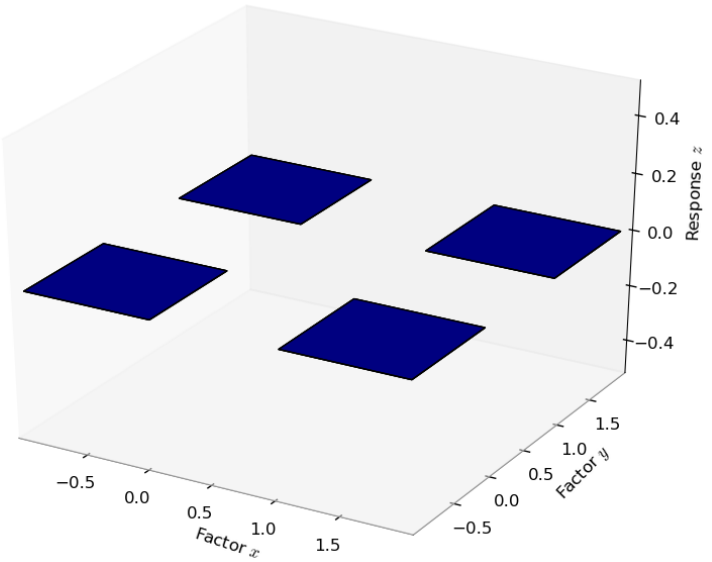
An m^k factorial design is used to investigate relationships between factors and their responses with a higher degree of **granularity**, or extent to which the response surface is subdivided to be sampled in the experiment. This can mitigate the effects of poor level value choices. An illustrative example of the benefits of increased granularity can be seen in Figure 2.2.

While this experimental design can provide further insight into more complex relationships between factors in the response surface, when the number of factors or levels is increased, the amount of time spent in simulation can increase very quickly. Extending the example in Section 2.1 where $k = 10$, if we use $m = 10$ instead of $m = 2$, we would have 10^{10} design points. With 30 replications of each design point each taking a minute, this experiment would take over 500 millennia.

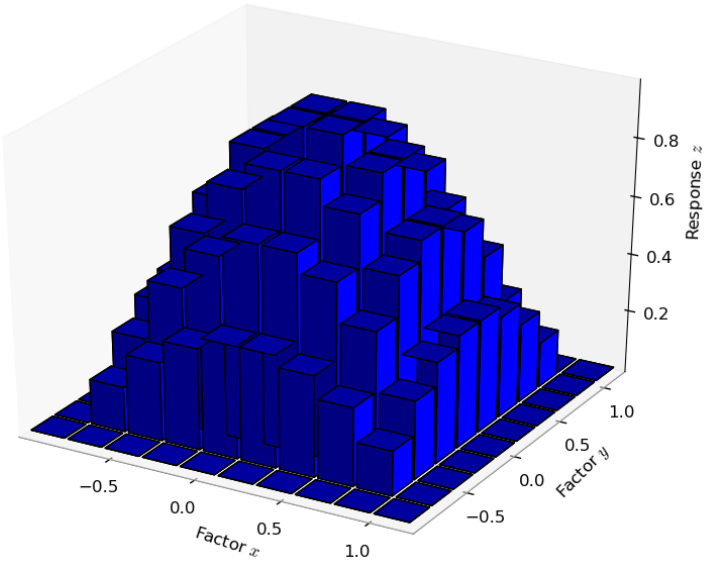
2.3 m^{k-p} Fractional Factorial Design

Fractional factorial designs offer a way to prune larger experimental design spaces to estimate more easily the effects of different factors and their interactions. These fractional experimental designs are subsets of the full factorial designs. For example, if we wish to prune a 2^4 factorial design, we could halve the number of design points, and we would have a $\frac{2^4}{2} = 2^{4-1}$ design points. As with our previous factorial designs, there are again m^{k-p} design points in such an experimental design where m is the degree of granularity, k is the number of factors, and $\frac{1}{m^p}$ is the fraction of the full factorial design investigated.

When using a fractional factorial design, there are many ways to choose the subset of the full factorial design. Some choices of subsets are more useful than others. For example, one could choose a subset of a 2^4 factorial design with a constant value for factor 4, and then a full factorial design for the other 3 factors. This design provides



(a) An example response surface for two factors, x and y with response $z = (x + 1)(x - 1)(y + 1)(y - 1)$ as observed using a 2^2 factorial design.



(b) An example response surface for two factors, x and y with response $z = (x + 1)(x - 1)(y + 1)(y - 1)$ as observed using a 10^2 factorial design.

Figure 2.2: Two examples demonstrating how more granularity can provide important insight into the shape and form of the actual response surface.

no insight into the effect of factor 4. Generally, a variety of design points should be chosen so as to have more data points to compute the effects of different factors and their interactions. For example, see Table 2.2.

Design Point	X_1	X_2	X_3	X_4
1	-1	-1	-1	-1
2	$+1$	-1	-1	$+1$
3	-1	$+1$	-1	$+1$
4	$+1$	$+1$	-1	-1
5	-1	-1	$+1$	$+1$
6	$+1$	-1	$+1$	-1
7	-1	$+1$	$+1$	-1
8	$+1$	$+1$	$+1$	$+1$

Table 2.2: 2^{4-1} Fractional Factorial Design example.

2.4 Latin Hypercube and Orthogonal Sampling

One of the more sophisticated experimental design techniques is called **Latin hypercube Sampling (LHS)**. This method is a special case of a fractional factorial design where $p = k - 1$. Each of these techniques greatly reduce the number of design points over a full factorial design but the choice of design points can help provide insight into more complex interactions in the response surface with fewer design points to investigate.

To understand a Latin hypercube, it is easiest to discuss first a Latin square. In a Latin square, a point is chosen in each row and each column such that there is only one point in each row and column. For an example of a Latin square, see Figure 2.3. A Latin hypercube is the logical extension of the Latin square as the number of dimensions is increased beyond two.

X			
	X		
			X
		X	

Figure 2.3: An example of a Latin Square.

There are many possible Latin hypercube experiments for a given set of factors and levels. One particular way to construct a Latin hypercube experiment is called **Orthogonal sampling**. This particular design places additional restrictions on the choices of design points in a Latin hypercube sampling. In Orthogonal sampling the hypercube is divided into separate regions of equal size, and a design point placed in each region. For an example of orthogonal sampling on a Latin square, see Figure 2.4.

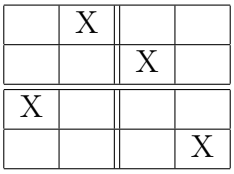


Figure 2.4: An example of Orthogonal Sampling.

Chapter Summary

Executing large simulation experiments can be computationally expensive. There are several experimental design techniques which can be used to investigate response surfaces. A 2^k factorial design can be used to investigate the effect of many factors, while an m^k factorial design can be used to investigate the shape and curvature of a response surface. Fractional factorial designs can used to reduce the number of the design points in an experiment while investigating a larger design space. There are several ways to execute these simulations to speed up the computation of the experiment which are discussed next in Chapter 3.

Chapter 3

Parallel Simulation Techniques

Simulation users often have access to computational resources such as servers, computer clusters, and other high performance workstations. These systems can have different architectures; most often they have multiple processing cores, allowing them to run programs concurrently. This allows users to run tasks in parallel, and consequently there are many ways to harness the computational power of these systems. This chapter discusses how one might harness these computational resources to accelerate the execution of large scale simulation experiments.

3.1 Fine-Grained Parallelism

One approach to utilize all of the processors available is to distribute a single simulation across all of available processors. This is called **fine-grained parallelism** [27]. In this case, different parts of the execution of the simulation must be separated to run on the individual processors.

There are many challenges associated with fine-grained parallelism. The developer of the simulator must be very careful during implementation to distribute the work in the simulation to each of the processors evenly. In many simulations, this is especially challenging due to inherent **data dependencies** in the simulation execution, in which one processor must wait on another processor's result before it can proceed. There is also overhead in communicating the result of some computation from one processor

to another.

The performance of a fine-grained parallel simulation does not scale linearly with the number of processors. The maximum theoretical speedup gained by parallelizing a process across n processors can be approximated by **Amdahl's law** [21]. Let p be the fraction of the process which can be parallelized and run on multiple processors, then Amdahl's law states that the maximum speedup with n processors goes as

$$\text{speedup} \ = \frac{1}{(1-p) + \frac{p}{n}}$$

The result of Amdahl's law can be seen in Figure 3.1. For example, if $p = 0.95$, then even using thousands of processors, there will only be a speedup of 20x. Fujimoto and Nicol [18] discussed several techniques to increase the value of p such that simulations can scale better with more processes.

3.2 Coarse-Grained Parallelism

An often simpler approach to balancing the work between many processors is called **coarse-grained parallelism** [27]. Coarse-grained parallelism distributes the work of the entire simulation experiment across many processors by assigning a single, sequential simulation to each processor. This allows processors to work independently of one another thereby eliminating overhead in synchronization and communication.

In coarse-grained parallelism, no processor ever needs to wait for the result of a computation performed by another processor. Furthermore, the processes are independent, which eliminates all synchronization overhead. This is therefore an **embarrassingly parallel** problem and thus $p \approx 1$ and by Amdahl's law:

$$\text{speedup} \ \approx \frac{1}{(1-p) + \frac{p}{n}} = \frac{1}{\frac{1}{n}} = n$$

This result is only applicable where the number of simulations which need to be run is a multiple of the number of processors available or greatly exceeds the number of processors available.

For example, assume we have one design point with 30 simulations to run on 4 processors, each of which takes time t to run. Using coarse-grained parallelism, the

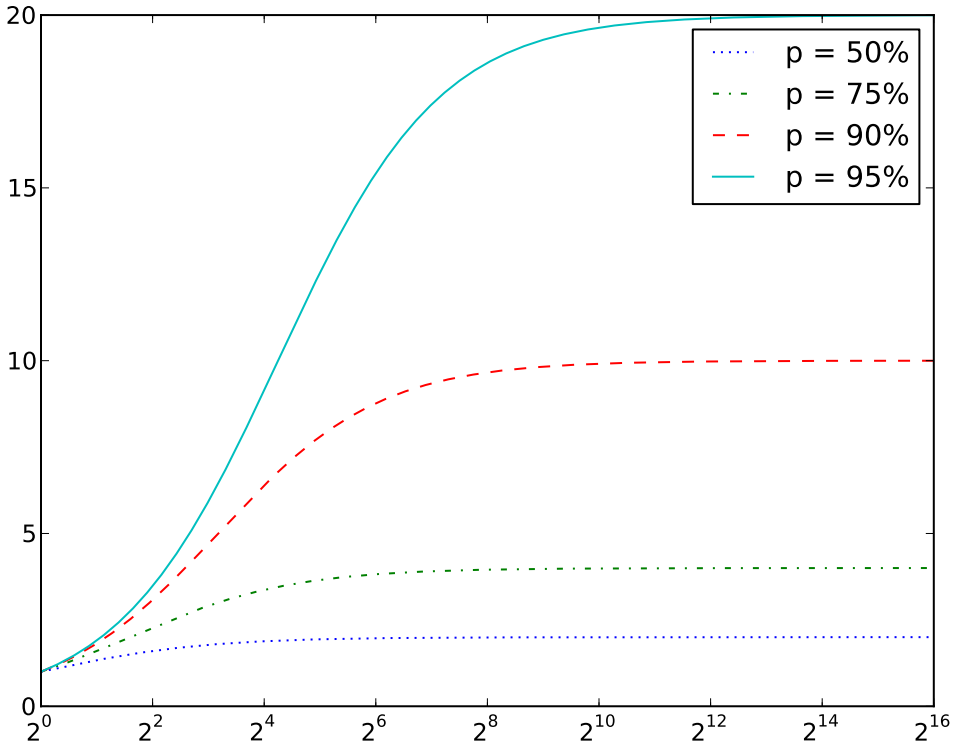


Figure 3.1: The theoretical maximum speedup possible using multiple processors as a result of Amdahl's law

first 4 simulations will take time t to run, and then the next 4 simulations will start and take time t to run. The first 28 simulations will therefore finish in $7t$ time. At this point, only 2 simulations are left for the remaining 4 processors to run, so 2 processors are left idle and the experiment takes $8t$ time. In this case, we have have a speedup of $\frac{30}{8} = 3.75$ instead of 4.

3.3 Multiple Replications in Parallel

Another approach to parallelizing a simulation experiment used by Pawlikowski [28] is called **Multiple Replications in Parallel (MRIP)**. This paradigm builds upon the assumption that the simulation user is following proper simulation methodology

and running multiple replications of each design point using different PRNG seeds. A **central server** dispatches independent simulation runs of the same design point with different seeds to be executed on different processors. During their execution, observations of performance metrics are reported to the central server overseeing the execution of the simulations. This process can determine when enough observations have been made to estimate performance metrics to within some tolerance specified by the user.

MRIP addresses the issue seen in coarse-grained parallelism when the number of simulations which need to be run are not significantly greater than the number of available processors. Extending the example from Section 3.2, instead of running 30 simulations on 4 processors, we instead run 4 simulations. Each of these simulations simulates more virtual time, enough time to observe the minimum number of observations required to estimate the metric to within the desired level of confidence. There is overhead both in running a separate server and communicating these observations to the server. In comparison to coarse-grained parallelism however, the amount of time spent in transient will be less, and all processors can be kept busy until the experiment completes.

Chapter Summary

This chapter describes three techniques which can be used to speed up the computation of a simulation experiment using multiple processors. Fine-grained simulation can be used to parallelize a single simulation run, while coarse-grained simulation can be used to run many independent simulations. The MRIP technique is a variant of coarse-grained simulation which can have better performance than coarse-grained execution. Next, Chapter 4 will describe how previous automation tools have integrated these experimental design and parallel simulation techniques to help users run experiments efficiently.

Chapter 4

Previous Automation Tools

Several tools have been developed to automate one or more steps of the proper simulation workflow for different simulators. In this chapter we introduce some of these tools, namely CostGlue, ns2measure, ANSWER, Akaroa, SWAN-Tools, and JAMES II. The analysis of the features, strengths, and weaknesses of these tools helped us to reach key design decisions in the construction of the framework we present in this thesis.

4.1 CostGlue

A software package called CostGlue was developed to aid telecommunication simulation users in storing and sharing their results. CostGlue provides an **Application Programming Interface (API)**, in the programming language Python, which helps one to store and access simulation results. CostGlue also has a modular architecture which allows for the development of **plugins**. These plugins can extend the original functionality offered by CostGlue without becoming part of the project's core source code. [33]

The CostGlue API exposes all of the simulation results and meta-data to plugin developers. This allows for the development of plugins which can be used to conduct statistical analysis, generate figures, or export results into a format accessible by other post-processing tools such as R [6], SciPy [7], Octave [4], or Matlab [3]. The CostGlue

developers also discuss the possibility of developing external processes which could be used to expose results stored in the CostGlue database via a publicly available web application. [33]

CostGlue, however, does not automate the process of parsing the results in the output from the simulation and therefore, does not prevent errors in this stage of the simulation workflow. Similarly, CostGlue does not provide facilities to process the results from the simulation to extract the metrics of interest. This introduces another potential area for errors to be made in the simulation workflow. Finally, users must import their results into CostGlue using custom developed scripts, which can again introduce opportunities for errors to be made in the simulation workflow. All three of these issues can lead to results which are not credible.

The CostGlue project demonstrates several desirable capabilities for handling simulation data. Though the project is no longer under active development, these lessons can be applied to future projects. The modular architecture for accessing simulation results allows developers to easily extend the tool for their needs. Users can then share the tools they have developed to help other users in the simulation community.

4.2 ns2measure & ANSWER

While the CostGlue framework provides means for storing and accessing simulation results it does not provide any facilities for collecting statistics from a simulator. Cicconetti et al. [15] has a project called ns2measure which addresses this issues and eases the process of extracting simulation metrics. Andreozzi et al. [12] also developed a tool called ANSWER which builds upon the functionality provided in ns2measure to help users automate large simulation experiments.

The ns2measure project aims to ease the process of collecting statistics during simulation execution using the network simulator *ns-2*. Ordinarily, when using *ns-2*, a trace of the network activity is written out to the file system for posterior analysis. Users must then carefully process these trace files to extract the statistics they are interested in studying. Processing these results is often conducted with unverified scripts which can produce biased or erroneous results. The ns2measure project provides a framework to collect statistics during the execution of the simulation itself. Furthermore, it provides statistical analysis tools which help users conduct more statistically sound simulation experiments. [15]

A project called ANSWER, developed by the same research group at the University of Pisa, Italy, works in harmony with ns2measure. While ns2measure aids a user in gathering accurate statistics for a single design point, ANSWER helps to automate running large scale simulation experiments with hundreds or thousands of design points. This process is accelerated by distributing independent simulations across multiple available processors using coarse-grained parallelism. ANSWER also provides web-based tools for interfacing with collected results.

These two software tools, ns2measure and ANSWER, offer many important features to simulation users. First, ns2measure provides mechanisms to extract observations of performance metrics directly during simulation execution. When ns2measure is used in conjunction with ANSWER, simulation users can easily conduct a simple, credible simulation experiment using *ns-2*. One major shortcoming of ns2measure and ANSWER is that they can only be used with *ns-2*.

4.3 Akaroa

In contrast to ANSWER, the Akaroa project developed by Pawlikowski [28] used MRIP as described in Section 3.3 to accelerate running a single design point instead of an entire experiment. The Akaroa project was originally developed for use with *ns-2*, but it has since been ported to work with other simulators such as OPNET++. Pawlikowski [28] believes it can be adapted for use with other stochastic network simulators as well.

While Akaroa demonstrates important functionality in software automation tools, it has a few shortcomings. Akaroa can only be used to execute a single design point. This requires users to manage each of the design points in their experiment manually. Because users can make mistakes when managing the execution of these design points, we would like future tools to use MRIP to automate the entire experiment. Furthermore, Akaroa does not integrate with other tools such as ANSWER which can help users manage their simulation experiments. Finally, the Akaroa project requires permission from the authors to use for any application outside of teaching and non-profit research activities.

4.4 SWAN-Tools

One of the first software projects which attempted to automate running an entire simulation experiment to ensure the credibility of results was the SWAN-Tools project developed at Bucknell University by Kenna [23], and Perrone et al. [31]. SWAN-Tools was developed for use with the Simulator for Wireless Ad Hoc Networks (SWAN). The tool guides the user through all the steps of a proper simulation experiment, and demonstrates many important functions in the automation of simulation experiments.

SWAN-Tools helps the user to create valid experiments and run independent simulations in parallel across many physical computers. Also, the tool aids the user in data analysis by presenting results to be viewed in a web browser, to be downloaded and used with a statistics package, or to be graphically presented using proper plotting techniques via a web based interface. Lastly, the tool makes the results available via a website to which any scholarly article can be linked.

The lack of flexibility in this tool is its major shortcoming. It was built exclusively for use with SWAN, and used a simulation model which was hard-coded into the tool. These constraints limit the potential uses for the tool. However, the aforementioned features which guide the user through all steps of a proper simulation experiment can be applied to future automation frameworks.

4.5 James II

The James II project takes a different approach to automating elements of proper simulation workflow. Instead of building tools which work in tandem with a specific simulator, JAMES II provides a framework upon which simulators can be built. It has a modular architecture with plugins for problem domains ranging from Computational Biology to Computer Networks. [22]

Once a simulation model has been defined using the JAMES II framework, there are tools available to help run simulation experiments. Also, there are plugins which help users use both coarse- and fine-grained parallel simulations. Furthermore, JAMES II provides facilities for storing and analyzing results.

The simulation must use the JAMES II core framework in order to take advan-

tage of the several available plugins. Additionally, since the JAMES II framework is Java-based, all JAMES II simulators must be written in Java. While JAMES II has an interesting architecture and feature set, many of the features for automating simulation experiments are not compatible with simulators which are not specifically built for this framework. The modular architecture of JAMES II, like CostGlue, allows it to be more widely applicable to different problem domains.

4.6 Lessons Learned

These tools have demonstrated several features and functions which are important for future automation tools to incorporate.

- A plugin system to allow users to customize the tool to their needs.
- A guiding user interface to help inexperienced users along.
- Parallel simulation techniques such as MRIP.
- A web interface to view the experiment configuration and results.

These ideas will be incorporated into my framework, SAFE, described next in Part II.

Chapter Summary

Several tools have been developed which automate different aspects of the proper simulation workflow. These tools demonstrate important functionality: output processing, output storage, distributed execution, rigorous statistical methods, and a guiding user interface. Lessons learned from these tools will be incorporated into my framework described next in Part II.

Part II

The Simulation Automation Framework for Experiments

Chapter 5

Architecture

The **Simulation Automation Framework for Experiments (SAFE)** addresses many of the aforementioned problems in both simulation usability and credibility. This chapter discusses SAFE's architecture and feature set which has been designed to address the following general goals. The framework should:

- Be flexible and extensible.
- Automate the simulation workflow so as to ensure the credibility of the experimental process.
- Use the MRIP methodology to accelerate the execution of experiments.
- Include a web-based component to allow for the visualization of experimental results.
- Present differentiated interfaces which meet the needs of novice and experienced simulation users.
- The framework should be flexible such that it can be extended to work with other simulators. This is not to say that SAFE can be extended to work with every possible simulator. Although existing simulators only available in binary format would be challenging to integrate with SAFE, it should be relatively straightforward to modify open source simulators to work with the framework.

SAFE consists of various components which automate different processes in the proper simulation workflow. It employs a **client-server** model in which the central server called the **Experiment Execution Manager (EEM)** coordinates the actions of numerous **simulation clients**. Each simulation client controls the execution of a single design point by a simulator. A broad overview of SAFE's architecture is presented in Figure 5.1.

5.1 The Experiment Execution Manager

The EEM coordinates the behavior of the entire simulation experiment. It handles all of the data in the experiment including, but not limited to, how it is processed, how it is stored, and how the framework responds to it. The EEM itself does not conduct complex, computationally expensive simulations or analyses, but instead coordinates and communicates with other processes which handle these tasks. All data in the experiment flows through the EEM at some point.

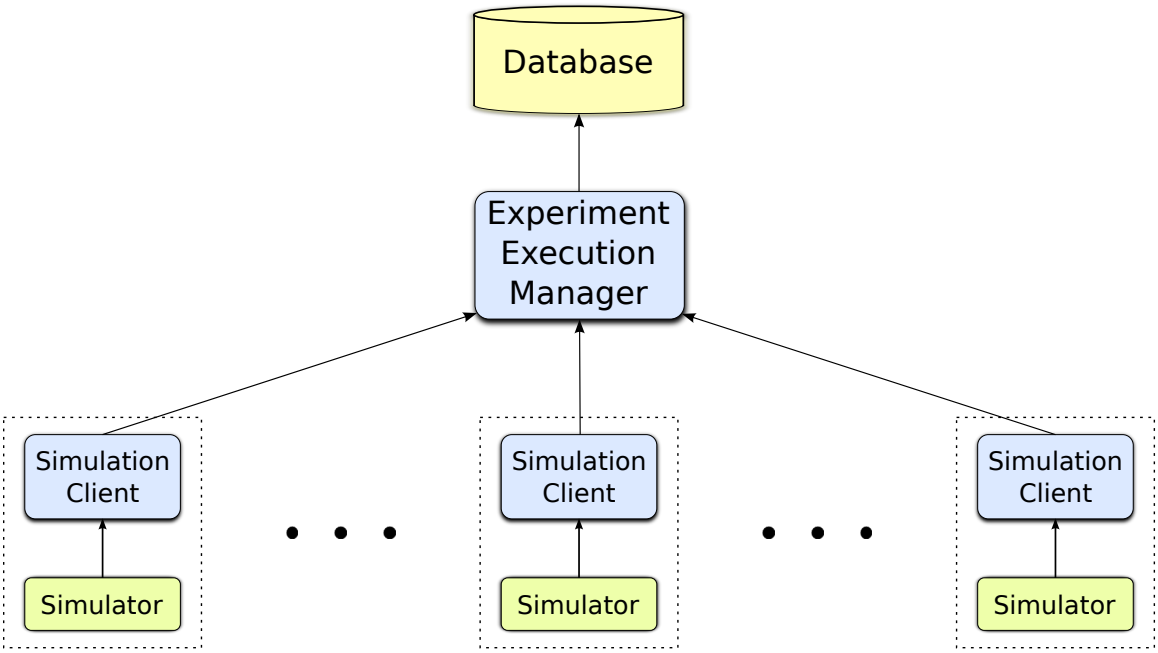


Figure 5.1: Overview of the architecture of SAFE.

The EEM accepts several input files which specify options in the EEM and define the experiment to run. The languages which describe these inputs are described later in Chapter 6. Using the information gathered from these input files, the EEM computes all of the design points in the experiment. The EEM then manages dispatching the necessary simulations to the available machines to be executed using an MRIP style parallelization technique. The Simulation Client, as described in more detail in Section 5.2, reports results back from the simulator to the EEM. The EEM must also coordinate how these results are handled after they are received from the simulation client.

One of the EEM's primary responsibilities is handling all interactions with the database where SAFE stores all of its data. (The database itself will be discussed in Chapter 8.) All results from the simulator are sent to the EEM which processes the data and stores the results in the database. When the experiment is complete and users need to conduct posterior analysis of their results, they must access their results through the EEM which helps to ensure that the results are accessed and processed properly.

The EEM is also responsible for conducting proper statistical analyses. In SAFE, the transient and the end of the design point's execution are both detected by external processes. Within this design, the EEM is responsible for forwarding the intermediate results to each of these processes in addition to the database. The EEM monitors these processes to know when the transient has passed and later when the design point should be terminated. This architecture can be seen in Figure 5.2.

SAFE has support for plugins, which extend its functionality, much like CostGlue and JAMES II. The plugins and any of their associated options are configured through the experiment configuration language described in more detail in Section 6.2. Currently, there are plugins for parsing the experiment description file, as well as generating all of the design points. There is also a hook to allow users to incorporate other plugins which manipulate data. The plugin system allows SAFE to be adapted to current and future needs of the simulation user.

5.1.1 Asynchronous / Event-Driven Architecture

Two of the major requirements of the EEM are responsiveness and availability: the EEM must respond or react quickly to different actions and events so as to be ready for the next event. By design, the tasks which the EEM executes are seldom ever

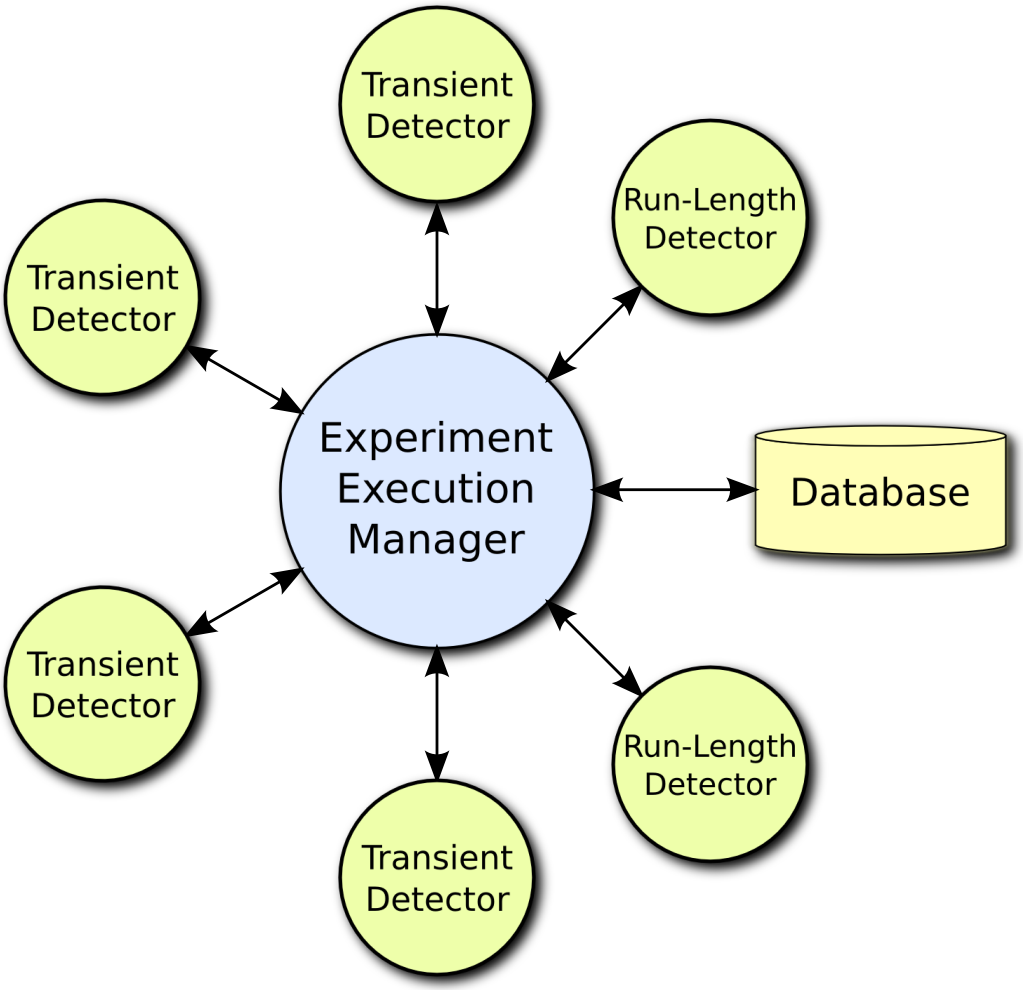


Figure 5.2: Architecture of the interactions between the EEM and transient and run length detection processes.

computationally expensive or long-running, so that the EEM can react quickly to all inputs.

This type of application is often implemented using the **reactor** design pattern [34]. This design pattern yields an **event-driven** programming model in which the application waits for an event to happen, and a method known as a **callback** is called to respond to the event and any associated data. After a callback is processed, the reactor drops back into the main loop where it waits for the next event to occur. There are many algorithms to decide when an event happens, but for networked applications such the EEM, the most common mechanism is to use the `select()` system call, which returns when a file descriptor is ready to be read from or written to.

An example of the benefits of this event-driven programming paradigm is querying a database. In most synchronous programming models, when a query is made to the database, the execution of the program **blocks**, or waits until the result of the query is made accessible. This can simplify the programming model because one is assured that when the query returns, all of the results are available. In an application which strives to achieve high availability however, this programming model is rather restrictive, because the server is unresponsive while the program waits for the result from the database. This time can be used to respond to other events, which otherwise would have to wait until after the database has returned.

This behavior can be seen in Figure 5.3. Let the blue sections represent the time spent processing a database request, the orange is a response to a request for another design point, and the green is another database request. At some time as in Figure 5.3, the request for the next design point is received. In the asynchronous case, the process is idle and can respond to the request immediately. In the synchronous case, the process is busy, and cannot begin to service the request until it has processed the results from the database. Similarly, the database request in green is dispatched before a request has been received for the blue database request. The reply from the green database request can be handled while the process waits for the result of the green database query. Consequently, the asynchronous programming model allows for the process to handle different events in the system while it waits for other events to occur.

The EEM is implemented in the programming language Python [5] and relies heavily on the asynchronous, event-driven library called Twisted [8] which implements the reactor design pattern. Using this library, callbacks can be defined to handle results from the simulation client, query results from the database, handle messages

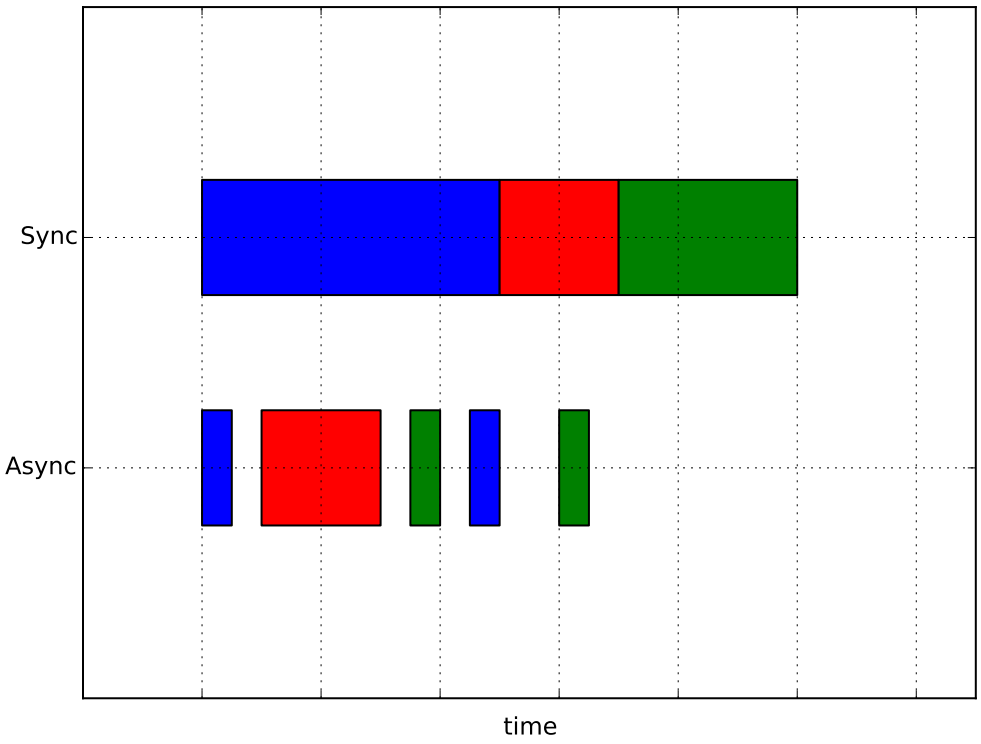


Figure 5.3: A visual depiction of the benefit of an asynchronous vs. a synchronous programming model

from external processes used to detect transients, and many other types of events.

5.1.2 Dispatching Design Points

In coordinating the execution an experiment, the EEM dispatches to different computers the simulations which correspond to various design points. We can estimate how the distributed execution of the experiment affects the time to completion. In total, we will assume N samples must be generated for each design point such that metrics can be estimated to within the desired confidence level. When these N samples have been generated, the design point's execution can be terminated. The simulation of each design point includes a number of independent replications, each of which must

incur the cost of "warming up" to the end of its transient. We say that t samples are collected during this period. The data deletion method requires that none of these first t samples be used in estimating the desired metrics.

If we have r independent simulations replicating the same design point, which combined will generate the required N samples, on average, each replication needs to generate $\frac{N}{r}$ post-transient samples. Then on average, each simulation will run until is has generated $t + \frac{N}{r}$ samples. In total, across all r replications, the total number of samples collected is then $r(t + \frac{N}{r}) = rt + N$. While the work to produce the N samples is needed, the generation of the rt samples is only productive in the sense that it allows replications to move beyond their transient.

To put this in perspective, consider the following example in which each simulation executes to 1000 s of simulated time, the transient is 100 s, and we have $r = 30$ replications of the design point. We see that $30 \times 1,000\,\text{s} = 30,000\,\text{s}$ of simulated time are executed, but only $30 \times (1,000 - 100)\,\text{s} = 27,000\,\text{s}$ of simulated time produce statistically useful samples. This means $3,000$ s, or 10% of simulated time is used to produce data which is disregarded in rigorous statistical analyses.

Although this analysis makes some simplifying, unrealistic assumptions, it demonstrates the overhead of simulating the transient. A better model would assume the transient and run length are random variables. There exist several techniques to dispatch simulations to utilize efficiently the computational resources in light of the overhead of the transient.

One dispatching algorithm is to dispatch a single design point at a time to all p available processors. Each processor continues simulating the same design point until the EEM has determined that enough observations of the metrics of interest have been collected. At this point, all of the remote hosts terminate the active simulation, and the next design point can begin execution. This is how one would conduct a simulation experiment using MRIP in the Akaroa 2 project [28]. Using this algorithm, we would simulate approximately pt units of simulated time in transient.

In a simulation experiment we have many design points which can be processed concurrently across our computational resources. Consequently, if each design point is dispatched to at most n processors where $n < p$, there will be multiple design points running concurrently. Each design point will simulate approximately nt units of simulated time in transient in comparison to pt in the previous algorithm.

I developed a design point dispatching algorithm for SAFE which is a variant

on these existing algorithms. In this algorithm, the design point which has seen the fewest results is dispatched first. This algorithm seeks to minimize the number of independent simulations which are executed for each design point (in essence, n for each design point) so as to minimize the time spent in transient. This method can be further adapted to set a minimum number of independent simulation runs n so as to reduce any bias from the choice of PRNG seed.

5.1.3 Web Manager

As a result of using an asynchronous programming model, the EEM can respond to a vast array of different types of events by installing new callbacks. This allows the EEM to expose data and parts of its state via a web-based interface, called the **web manager**. Furthermore, because the EEM and the web manager are all contained in a single process, users can control aspects of the experiments execution from the web interface.

As previously stated, one of the principal goals of SAFE is to provide a framework which both novice and experienced simulations users can employ to automate their simulation experiments. While more experienced users may gain more power through using command line oriented tools, a novice simulation user may feel more comfortable using a web browser to control their experiment. The web manager allows for this use case by allowing a user to create an experiment by uploading the necessary input files (described in more detail in Chapter 6), and using the web manager to start or stop the execution of the experiment.

Another use case for the web manager is to allow users to view their results via the web. Again, this allows the novice simulation user to analyze results without requiring more sophisticated scripts or command line oriented tools. More importantly, the web manager exposes the complete experiment setup coupled with results. This allows users to publish their results to the web such that people around the world can view them, and link back to those results from any publication. These users can also repeat the same experiment by downloading the input files for the specific experiment. This type of functionality was demonstrated by Perrone et al. [31] to significantly enhance the credibility of published results by ensuring repeatability.

A feature to be incorporated in the web manager in the future is a plotting tool. Such a tool would guide the user through the generation of a plot. This type of functionality is available in both SWAN-Tools [31], and ANSWER [12]. This not only

provides an intuitive interface to the simulation user, but also to others interested in analyzing the data after the results have been published. The web manager is a major area of future work under the continued funding of the NSF [20].

A common architecture for exposing data to clients on the web is called **Representational State Transfer (REST)**; this model was first defined by Fielding [17] in his doctoral dissertation. REST defines how web applications can be queried for different resources. This allows developers to interact with web-based applications by querying for specific information. Furthermore, it allows for external applications to be developed which can access the information made accessible through this web-based API. This architecture is applied in the web manager which allows for external developers to interface with the web manager to query for results and control other aspects of the behavior of the EEM through both the web browser as well as customized scripts.

5.2 simulation client

The EEM is responsible for orchestrating how different processors contribute to the execution of the experiment. It does so by communicating with a process called the **simulation client** running on each computer (local or remote) participating in the execution of the experiment. The simulation client manages how simulations are executed, and how results are reported back to the EEM.

The first role of the simulation client is to connect to the EEM and register as an available host for the execution of simulations. In so doing, the simulation client transmits important details about its local environment such as the operating system version, architecture, etc. (e.g. GNU/Linux Kernel 2.6.33 Intel x86_64). If for any reason problems are found which can be correlated with the collected data, then having this information can be useful in detecting any problems, or disregarding any results.

The next step in the execution of the simulation client is to request a design point to run. The EEM replies with a design point and any information required to start the simulation running (e.g. the PRNG seed). The simulation client then spawns a new process for the simulator itself. While the simulator is processing, the simulation client still has several responsibilities.

The simulation client must continue to listen for instructions from the EEM. At any point, the EEM can send a message informing the simulation client that the experiment is complete and it should terminate the simulator and gracefully shut down the process. The EEM can also inform the simulation client that the active design point is complete, in which case the simulation client should terminate the active simulation, and then request the next design point to simulate.

The simulation client must also listen for results from the simulator itself. If the simulator can output intermediate results during execution, the simulation client can forward these results to the EEM for storage and post-processing. If the simulator cannot forward intermediate results, these results must be sent to the EEM upon the termination of the simulation.

The simulation client, unlike the EEM, is not simulator specific. The simulation client abstracts the details of the simulator away from the EEM allowing for the possibility of using the EEM with different simulators. The simulation client decouples the EEM from the simulator itself allowing the EEM to be more general. Also, the functionality needed in a simulation client is largely the same between different simulators, and therefore the development of a new simulation client for a new simulator can be streamlined by following the example of previous code.

The requirement of the simulation client to be listening and acting upon data from a few data sources lends itself to the reactor design pattern described in Section 5.1.1. Currently, my simulation client implementations are all implemented in Python using the Twisted Library just as the EEM is, but it is possible to develop a simulation client in another language such as C or C++. In the development of the simulation client, one must take care to avoid **busy-waiting** and using processors cycles to stall and wait for data from a data source. The simulation client should be as light a process as possible to leave the systems resources available to the simulator itself.

Current simulation client implementations are rather simplistic. A future development which would likely provide greater performance is to develop a simulation client which is capable of managing many simultaneous simulators on a single host. This could provide slightly better performance for systems with a large number of processors which could run a single simulation client instead of a simulation client for each processor.

Chapter Summary

The SAFE project uses a client-server programming model. By abstracting the implementation details associated with integrating SAFE with a specific simulator, the SAFE framework gains flexibility. This allows for the possibility of integrating SAFE with other simulators. SAFE also defines a new way to dispatch design point to simulation clients in such a way as to minimize the aggregate amount of time spent in transient. The EEM architecture allows for the integration of external tools and libraries through other processes as well, such as through the use of the plugin system. Next, Chapter 6 describes the languages used to configure the experiment for execution.

Chapter 6

Languages

SAFE's architecture allows it to be flexible and adapt to a user's many needs in their simulation experiments. It achieves this flexibility using a modular architecture, and exposes many options to users via configuration files. These configuration files are required to specify basic options for the EEM, plugins and plugin options, the design of the experiment, and the simulation model itself. These options are provided to the EEM as documents written according to configuration **languages**, which are defined by Andrew Hallagan in his Honors Thesis [19]. The languages are extensions from XML (eXtensible Markup Language), which provides mechanisms for encoding information in formats that can be easily manipulated by computers. To fully understand the benefits of using XML, it is necessary to first describe its structure.

6.1 XML Technologies

The XML language defines a text-based document format which is designed to be simple and easy to parse/interpret with a computer program. The World Wide Web Consortium which defined the XML language standard through a set of simple rules which define how a **valid** XML document is formed and structured. [9] The simple nature of these rules allow XML to be widely applicable in many different problem domains.

An XML document is composed of three main types of content. The basic building block of XML is called an **element** which encodes some piece of data, or even a conglomeration of data. An element is separated from other pieces of the document using opening and closing **tags**. A tag is a piece of text often called a tag-name surrounded with "<" and ">" characters. A closing tag has the same tag name as the opening tag, but instead begins with "</". An example of a valid XML element is seen in Listing 6.1.

```
<tagname>contents of the element</tagname>
```

Listing 6.1: An example XML Element.

The third type of content in an XML document is called an **attribute**. Attributes can be used to specify meta-data for an element. Attributes are specified inside of the tag surrounded by quotation marks. For example, we can add an attribute to the preceding example as in Listing 6.2.

```
<tagname some_attr="attribute value">element contents</tagname>
```

Listing 6.2: An example XML Element with an attribute.

In an XML document, elements are often nested to encode the relationships between the different pieces of data being encoded. Furthermore, the XML standard requires there to be a single root element within which all elements are nested. For example, if we wanted to encode a probability distribution and its parameters, we could nest the parameters in their own sub-elements as seen in Listing 6.3.

```
<factor distribution="Gaussian">
  <mean>5.0</mean>
  <variance>2.0</variance>
</factor>
```

Listing 6.3: An example of how elements can be nested in XML document.

The rules which define the XML language specification are very basic. They do not define the context or meaning of any of the tags or attributes in the documents themselves. This allows for the development of **XML-based** languages which further restricts the XML language by specifying what types of tags can be used and how they can be composed to form elements. In application, the specific tags and attributes are given context so that data can be easily encoded and transmitted between systems.

For example, XML based languages are one of the primary means of encoding data on the World Wide Web. The Hyper Text Markup Language (HTML) which is used to encode most of the web content viewed in web browsers is an XML-based language. The HTML specification defines a set of valid tags which web pages are encoded in. Modern web browsers understand the meaning of these tags and render the page appropriately. For a brief example of HTML see Listing 6.4.

```
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>A Sample Page Title</title>
  </head>
  <body>
    <h1>The Heading on the Page</h1>
    <p>A paragraph of text. <b>This sentence is all bold.</b></p>
  </body>
</html>
```

Listing 6.4: An example HTML document.

These specifications for the newly created language can be defined in an XML Schema. There exist languages for defining schemas including the Document Type Definition (DTD), XML Schema (XSD), and REgular LAnguage for XML Next Generation (RELAX NG). Each of these languages has a different syntax.

Such a schema file can be used to validate a document. This functionality can act as a security measure to ensure that the document is well formed before a program tries to process its contents. In the context of simulation automation tools, inputs can be validated against the associated schema to ensure that the design point or model is valid. For example, a design point can be checked to ensure a level is given for each factor. Furthermore, it can check that all levels are valid (e.g. number of wireless devices in the simulation is positive). Validation will enhance the credibility of the final results by ensuring that all simulated models contain valid inputs to the simulator.

6.2 Experiment Configuration

The first language which SAFE uses to define an input file is the Experiment Configuration Language. This language is used to define specific options and behaviors of the EEM for a specific simulator. This includes defining plugins and options specific to each of these plugins like CostGlue and JAMES II.

An important option in the execution of the simulation experiment is that every simulator used to collect results in the experiment needs to be running the same version of the simulator. The experiment configuration language provides a mechanism to specify which version of the simulator the clients should be using. This version is checked against the host specific information which each simulation client transmits to register as an available simulation client. This can be used to ensure that the version matches that which is specified in the experiment configuration language.

There exist several different algorithms to detect the end of the transient. While the technical details of these algorithms are outside the scope of this thesis, we expect users will want to be able to apply these different algorithms to different experiments. The SAFE architecture allows developers to create their own transient detection algorithms in a separate script. SAFE then manages communicating all results to these external processes. This process is managed through the plugin system, and the experiment configuration language is used to specify options to the transient detection algorithm, and setup how the communication between the two processes is handled. An analogous plugin system exists to communicate with external processes which estimate when a simulation experiment can be terminated.

Another application of plugins in SAFE is in results handling. SAFE defines default behavior for how results are handled and stored, but additional plugins can be implemented to allow for additional callbacks to be executed when results are received. This can be useful for users with more sophisticated usage patterns.

6.3 Experiment Description Language

The next language which we have developed for use with SAFE is the Experiment Description Language. This language encodes the experimental design and offers users a flexible yet succinct language with which to define their experiment. The Experiment

Description Language is also an XML-based language.

The Experiment Description Language is broken into two primary sections. The first section encodes each factor, and all of the levels which it can be associated with in the given experiment. This section alone defines a complete factorial experimental design.

The second section defines constraints on the full factorial design. It provides many mechanisms with which full factorial experiments can be pruned. For example, design points can be individually excluded from the full factorial experimental design. More useful though in application is the ability to tie specific levels for different factors together, such that any combination which does not include both (or all), of the levels will not be included in the experiment.

Parameters to random distributions form an illustrative example of this feature of the Experiment Description Language. For example, if some process can be modeled with either a Gaussian or an Exponential distribution, then the factors μ and σ^2 must be coupled with the Gaussian distribution while the factor λ must be coupled with the Exponential Distribution. If we use the factors and levels described in Table 6.1a, there are a total of $2 \times 3 \times 3 \times 3 = 54$ design points in the full factorial design. However, the vast majority of these design points are not valid based on the aforementioned constraints regarding the parameters to each distribution. The six valid design points can be seen in Table 6.1b.

Factor	Valid Levels
Distribution	Gaussian, Exponential
μ	N/A, 5, 10
σ^2	N/A, 0, 1
λ	N/A, 10, 20

(a) A list of factors and levels.

Distribution	μ	σ^2	λ
Gaussian	5	0	N/A
Gaussian	5	1	N/A
Gaussian	10	0	N/A
Gaussian	10	1	N/A
Exponential	N/A	N/A	10
Exponential	N/A	N/A	20

(b) The set of valid design points.

Table 6.1: The full factorial design in Table (a) yields many design points with invalid parameter values. The set of valid design points is seen in Table (b).

The Experiment Description Language can be used to construct any experimental design space. This allows users to customize their experiment specifically to investigate certain qualities or quantities in the system while reducing the number of design points which needs to be executed. It also allows for users to describe experimental

designs such as fractional factorial designs and Latin hypercubes or any arbitrary experimental design space.

6.4 Boolean Expression Objects

The Experiment Description Language is parsed by a SAFE plugin, thereby allowing others in the simulation community to extend the language, and adapt the parser plugin for the new language. The parser plugin must communicate the factors and levels as well as the restrictions to SAFE which in turn generates the design points through the design point generator plugin.

I have developed a standard data structure, which I have named the Boolean Expression Object, for encoding the constraints on the full factorial design. This data structure encodes a boolean expression which can be applied to a design point to determine if it is contained within the experiment. In the example in Table 6.1, we can encode the boolean expression:

> Distribution is Gaussian and μ is not "N/A", and σ^2 is not "N/A" and μ is "N/A" or Distribution is Exponential and μ is "N/A", and σ^2 is "N/A" and μ is not "N/A."

This expression is stored in a tree-like object which can be traversed to evaluate whether a design point is contained in an experiment or not. Using the Boolean Expression module which I have developed for SAFE, this expression would be encoded as in Listing 6.5.

The resulting `expression` object, can then evaluate a design point, `dp` with `expression.evaluate(dp)`. These boolean expression objects can be automatically constructed during the parsing of the Experiment Description Language and then applied during design point generation to determine which design points in the full factorial design are included in the user's experiment.

```
from safe.boolean import Term as T
na = "N/A"
t_gauss = T("Distribution", "Gaussian")
t_gauss_params = !T("mu",na) & !T("sigma",na) & T("lambda",na)
t_exp = T("Distribution", "Exponential")
t_exp_params = T("mu",na) & T("sigma",na) & !T("lambda",na)

expression = (t_gauss & t_gauss_params) | (t_exp & t_exp_params)
```

Listing 6.5: An example boolean expression object.

6.5 Design Point Generation

The experiment description language is used to encode all of the design points in an experiment at a high level. This language can then be compiled down to a boolean expression object which encodes this information and can be used to check if a specific design point is contained within an experiment. The next step is to use this information to compute all of the design points in an experiment.

We have developed two different algorithms which can be used to construct these design points. Each of these algorithms has pros and cons depending on the size of the experimental design space relative to the full factorial design space. We have therefore designed SAFE to compute design points in a plugin, thereby allowing users to decide which algorithm is best suited to their needs. Furthermore, this architecture allows other developers to create their own design point generation algorithms.

6.5.1 Backtracking Design Point Generation

The first design point generation algorithm which was developed for SAFE we have called the backtracking design point generation algorithm. It uses a backtracking algorithm with constraint propagation to build design points included in the experimental design space.

This algorithm is best described recursively. The algorithm begins with a design point without any levels assigned to factors. The next factor chosen to which to apply the next level is the factor which has the fewest valid levels given the previous factor-

level assignments. The algorithm picks one of these valid levels, applies it, and recurs down to the next most constrained factor. The set of valid levels for all remaining factors is updated based on the boolean expression object. The algorithm recurs down until all factors have had levels assigned, at which point a valid design point has been constructed. From there, the algorithm backtracks and chooses other level choices so as to construct all of the valid design points in the experiment. Pseudocode for this algorithm is found in Listing 6.6.

Require: The backtrack function is called initially with:

- $experiment \leftarrow \emptyset$.
- *remaining* as all factors in the simulation model.
- $current \leftarrow \emptyset$.
- *levels* as a mapping from each factor to all associated levels.

```
function BACKTRACK(experiment, remaining, current, levels)
    if remaining = ∅ then
        add current to experiment
        return experiment
    else
        nextFactor ← most constrained factor in levels
        for level ∈ levels[nextFactor] do
            current[nextFactor] ← level
            remove nextFactor from remaining
            newlevels ← updated levels for remaining factors
            backtrack(experiment, remaining, current, newlevels)
            add nextFactor to remaining
            current[nextFactor] ← null
        end for
    end if
end function
```

Listing 6.6: Pseudocode for the backtracking design point generation algorithm.

The benefit of this design point generation algorithm is that it only explores design points which are included in the experimental design space. For experiments which are a small subset of the full factorial design, this is particularly efficient. The price of this time efficiency is memory space. The recursive nature of the algorithm, whether it is implemented recursively or iteratively using a stack, takes additional memory. The most memory is used in cases when the experiment is a large subset of the full

factorial design.

6.5.2 Linear Design Point Generation

To address the weaknesses of the backtracking algorithm, we have developed another design point generation algorithm we call the Linear Design Point Generation algorithm. Instead of only considering valid design points the linear design point generation algorithm considers all design points in a full factorial design and evaluates whether they are included in the actual experiment.

The linear design point generation algorithm iterates through all design points. (This can be done efficiently using the inverse of the design point id function described later in Section 8.3.) Each design point, `design_point`, is evaluated using the boolean expression object `bool_exp` by calling `bool_exp.evaluate(design_point)`.

When the experimental design space is a large subset of the full factorial design, there are relatively few design points considered which are not included in the experiment. In this case, there is little overhead associated with iterating through all of the possible design points. If instead the experiment is only a small fraction of the full factorial design, then there is significant overhead iterating through the full factorial design space. Another advantage of the linear design point generation algorithm is that it requires constant space.

```
experiment ← ∅
for designPoint in factorialDesign do
    if designPoint is valid then
        add designPoint to experiment
    end if
end for
return experiment
```

Listing 6.7: Pseudocode for the linear design point generation algorithm.

6.5.3 Design Point Construction

These two design point generation algorithms compute a set of design points encoded as Python dictionaries where the factor is the key and the level is the value. Each of

these dictionaries representing a design point must be encoded into a simulation model which can be passed to the simulation client for execution. SAFE uses **templates** to accomplish this task.

A template provides the structure of the simulation model, leaving placeholders for specific factor's levels to be inserted into the model. The most simple simulation template only requires a direct string substitution to construct the model. Some models require more structure, and structure which is dependent upon the levels in the design point. This requires a more powerful template system. For this purpose, SAFE uses the template engine Cheetah which is based in Python. An example of a Cheetah template can be found in Appendix C. Cheetah allows for conditional statements to be inserted into the template which can be executed to construct parts of the simulation model. Furthermore, Cheetah can execute loops to generate parts of models. Cheetah can, in fact, generate any text-based format and can therefore generate any simulation model. [2]

Once the simulation model has been generated using the template engine, the design point can be saved to the database. This allows results which are collected by simulation clients to be linked to the appropriate design points. The design point is then ready to be dispatched to the next available simulation client.

To speed up the startup time of the EEM, SAFE generates design points one at a time, as needed, when requests are made for the next design point from simulation clients. Design points are then generated one at a time, as needed. The design point generator class is itself an iterator, and as such, the next design point can be generated with a simple call to `next()`. By constructing the next design point on an as needed basis, SAFE is able to accelerate the startup process, and accept incoming simulation client connections for faster computation. This architecture can reduce the overall time a simulation experiment takes to execute.

Chapter Summary

Libraries and utilities for validating, generating and parsing XML based languages are ubiquitous across most modern programming languages. Languages based in XML can be developed to describe input and output data necessary to conduct simulation experiments. By building these language in XML, common tools can be adapted for our uses, and different components of a larger project can easily be written in different

languages, or be executed in different environments sharing only the specification for the XML based language. We have developed languages based in XML which are used to encode the experiment and its configuration to the EEM. These languages can then be parsed to generate all of the design points in the experiment. Next, Chapter 7 describes how these design points can be communicated to simulation clients running on remote hosts.

Chapter 7

Inter-Process Communication

The architecture of SAFE defines many separate processes, many of which can run on remote computers. While this architecture allows for many features and greater flexibility, it requires careful attention to how these separate processes communicate. In SAFE there are many types of **inter-process communication (IPC)** mechanisms which are used for different applications. A broad overview of the types of IPC mechanisms used in SAFE can be seen in Figure 7.1

The SAFE project provides MRIP functionality which requires communication between simulations on networked machines and the central server. In SAFE, the communication between the simulator itself and the EEM is broken into two separate steps. The first step is to communicate results from the simulator itself to the simulation client, and the second is to communicate the results from the simulation client to the EEM.

7.1 IPC Mechanisms

The SAFE project is designed to be run on a UNIX platform. On such platforms there are several different mechanisms available to communicate data from one process to another. Each of these mechanisms has advantages and disadvantages for different applications. In this section I provide a brief description of pipes and socket based IPC mechanisms which are employed in SAFE as seen in Figure 7.1. There exist

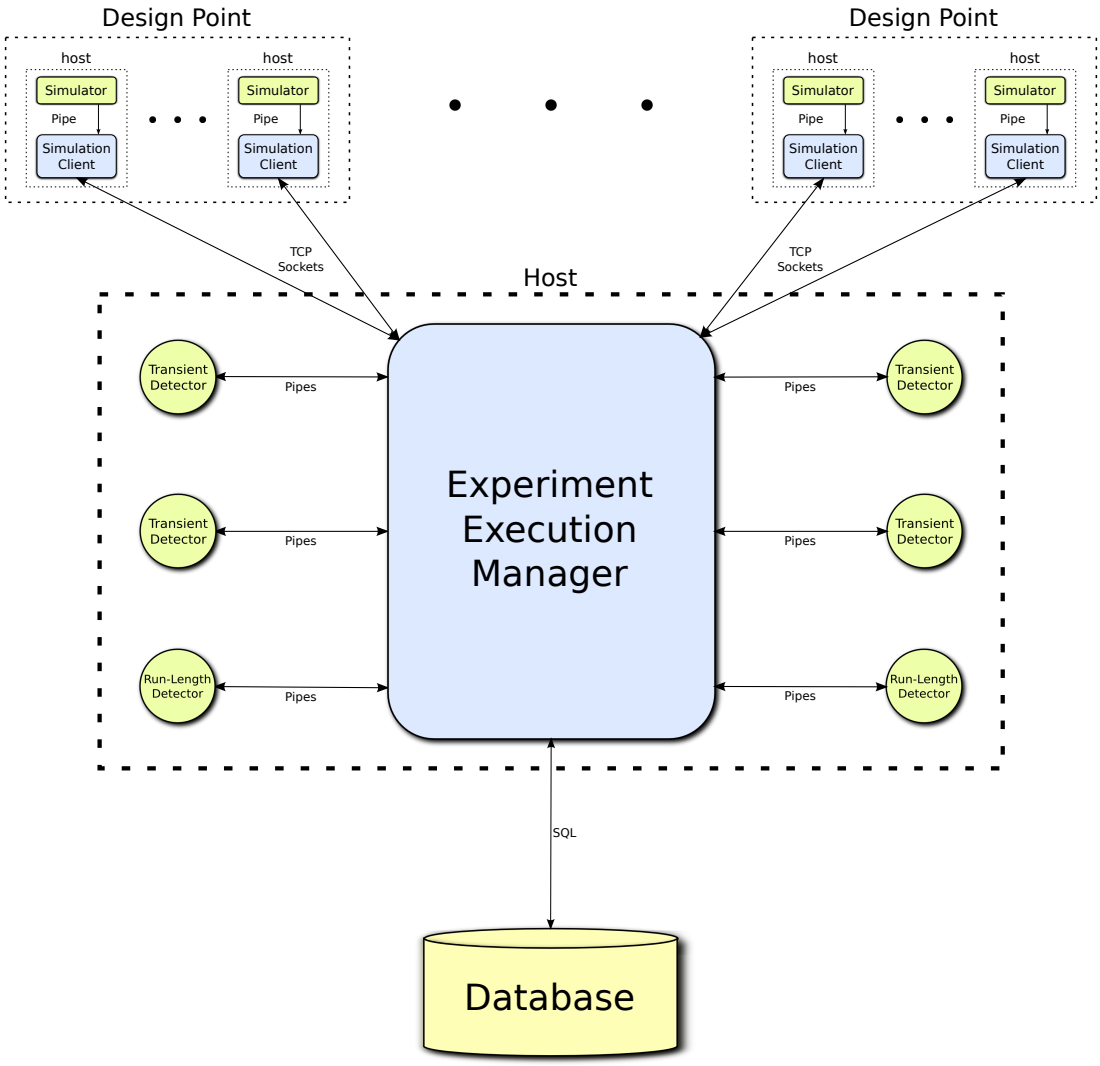


Figure 7.1: Architecture of the framework with respect to inter-process communication.

other IPC mechanisms such as **fifos**, **UNIX sockets**, **shared memory**, but these mechanisms are not used in SAFE.

7.1.1 Pipes

One of the most simple forms of IPC available on UNIX based systems is called a **pipe**. Pipes are implemented in the operating system itself, and therefore can only be used for communication between processes on the same physical machine. The operating system is able to synchronize the reading and writing from the pipe's buffer to ensure the consistency of the data. Pipes are also unidirectional, but two pipes, one for reading and one for writing, allow for bidirectional communication. Pipes expose data to the receiving process as a stream. [35]

Pipes are created by a single process which then calls `fork()` to spawn a **child process**. The child process shares the **open file table**, and the thus file descriptor of the pipe to the parent process is available. The parent and child processes can then communicate on this pipe using the standard system calls `read()` and `write()`.

7.1.2 Network Sockets

Pipes are a useful IPC mechanism, but they are restrictive in that they can only be used between related processes on the same host. Computer networks have been built to facilitate the passing of information from one computer to another. There are many layers of complexity in the **network stack**, which handle sending the electrical signal and routing the messages to the appropriate network node, but these technologies lie outside of the scope of this thesis. These lower layers in the network stack allow for the abstraction in the **transport layer** of **end-to-end** communication between nodes on the network via what are called **network sockets**. [32]

Application developers can interact with these network sockets to communicate with other processes on other computers. Two common transport protocols are used with sockets: the **User Datagram Protocol (UDP)**, and the **Transmission Control Protocol (TCP)**. Since each of these protocols offers a different kind of communication model, the specific needs of applications dictate which one is preferable.

UDP is a minimalist protocol in which data is encapsulated in discrete **packets**

or datagrams. A UDP packet carries a few pieces of metadata in its header, including the identification of source and destination ports to allow the multiplexing of packet flows to different applications. The UDP protocol is **connectionless** because it does not rely on the creation of a virtual circuit between sender and receiver before packets begin to flow. Additionally, UDP promises only a best-effort in packet delivery, without hard guarantees of reliability. Finally, UDP is not **order-preserving**, that is, packets can arrive at the destination in order different from that in which they were sent. [32, 37]

On the other hand, TCP implements a channel which is both reliable and order preserving. Even though TCP requires the creation of a virtual circuit from sender to receiver, it provides the abstraction of a continuous **stream** of bytes, delivered reliably and in-order [32, 37]. To provide this communication model, TCP incurs significant overhead and therefore it is not always the best protocol for every application. Time-sensitive applications, such as streaming audio or video, can tolerate packet loss much better than it can tolerate higher end-to-end delays between sender and receiver; for those applications UDP is a better choice. On the other hand, there exists another class of applications which benefit from the communication model of TCP, which is straightforward to use.

7.2 EEM ↔ Simulation Client

The first IPC mechanism used in SAFE allows for the EEM to communicate with each individual simulation client. To allow simulations to be distributed on local or remote hosts connected by a network, a network based IPC mechanism must be employed. This requires that the IPC mechanism chosen for this application be a network based IPC mechanism. For this reason, a socket based IPC mechanism has been chosen for the communication between the EEM and the simulation client.

Several types of messages must be exchanged by the EEM and the simulation client; most of these are simulation results being reported. These are small messages which contain an individual result, a double precision floating point number, and a few pieces of metadata describing the result. These results reflect the execution of a simulation run and they must be captured and stored for posterior analysis. So that none of the results would go unrecorded due to packet loss between the simulation client and the EEM, we elected to use TCP to interconnect the two processes.

The communication between the EEM and the simulation client follows a protocol with message types described as follows. Figure 7.2 illustrates how these message types are used in the course of a simulation experiment.

- `Register Message`: Sent by the simulation client to the EEM when it first connects. It provides information about the local simulation environment.
- `Next Simulation Request`: Sent by the simulation client to the EEM after the simulation of a design point terminates, or immediately following a `Register` message. Represents the simulation clients asking the EEM for a new design point to run.
- `Next Request Reply`: Sent by the EEM to the simulation client as response to a `Next Simulation Request` message. Carries an XML document describing the simulation model for a design point. The simulation client uses the XML document to setup the simulation run, which is then executed.
- `Result`: Sent by the simulation client to the EEM upon receipt of a result from the simulator. This message carries a sample of a metric generated by the simulator, which is used by detectors for transient and run-length and also stored in the database along with all data pertaining to the same experiment.
- `Finished`: Sent by the EEM to every simulation client which has been selected to collaborate in the simulation of a given design point. This message indicates that, for this design point, enough samples of the results have been collected so that the desired metric can be estimated within the user-specified confidence interval. Upon receiving this message, the simulation client terminates the current run and issues a `Next Simulation Request`.
- `Terminate`: Sent by the EEM to all simulation clients when all design points for the experiment have been completed. Upon receiving this message, the simulation client processes themselves terminate.

7.3 Simulator ↔ Simulation Client

The simulation client uses the information communicated from the EEM in the aforementioned protocol to control the initialization and communication with the simulator

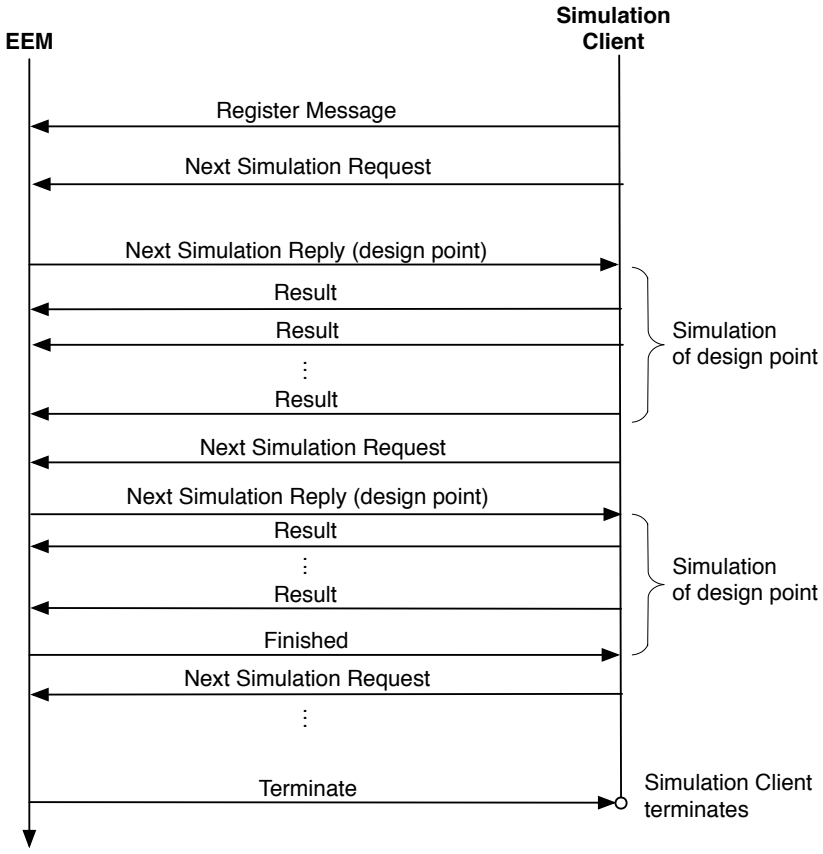


Figure 7.2: Communication protocol used by EEM and simulation client.

itself. The simulator and the simulation client are necessarily both run on the same machine. This eliminates the need to use a network based communication mechanism, but it does not rule out network based IPC mechanisms as a viable solution for this communication channel. There are two primary applications for communication between these two processes: communicating intermediate results from the simulator to the simulation client, and informing the simulator to terminate gracefully at the instruction of the EEM. For these two applications, I have chosen different IPC mechanisms.

To decide which IPC mechanism to use to pass results, I conducted several informal tests. I found pipes to be an order of magnitude faster than socket-based IPC mechanisms. The simulation client spawns a child process for the simulation itself, so pipes can be used for this form of communication. For the simulation clients I have developed, I have implemented the IPC between the simulation client and simulator itself using pipes. While this is a fast and easy form of IPC, simulation clients for other simulators could choose to use alternative IPC mechanisms. The simulation specific details of this design are described in more detail in Chapter 9.

Most simulators are not designed to be actively listening for data coming from external processes through pipes or other file descriptors. Consequently, it could be challenging to integrate the communication from the simulation client to the simulation itself using pipes. The only type of information which the simulation client needs to send to the simulator is a message to gracefully terminate the simulation. An alternative to using pipes to communicate this one simple message is to instead use a **signal**. The simulator executes the **signal handler** when the simulation client sends the specific signal. This method terminates the execution of the simulation gracefully.

Using a combination of pipes and signals, SAFE can interoperate with many different simulators, particularly those which are open source and can be easily modified for use with SAFE. For simulators in which the source code cannot be easily modified, the simulation client must be developed to interact with the simulation to extract the relevant information. In the worst case scenario, the simulation client cannot communicate directly with the simulator during its execution. In this case, the simulator can write results to the file system and the simulation client can parse all of the results to extract individual results to send to the EEM. This eliminates many of the benefits of the MRIP architecture, but still allows the simulator to integrate with SAFE to use all of its additional automation features.

7.4 EEM ↔ Transient and Run Length Detector

One of the plugins provided by SAFE allows for the computation of the transient detection and the run length detection to be offloaded to an external process. By offloading this computation to a separate process, a few things are gained. First, the EEM is made more responsive because it spends less time blocking on statistical calculations. Second, it allows SAFE to integrate with other external tools and utilities which can be used to estimate the transient and the run-length.

Much like the communication between the simulation client and the simulator, the communication between the EEM and the transient and run length detection processes can be restricted to the local machine. Furthermore, the EEM is responsible for spawning the detection processes, and therefore pipes are a natural IPC mechanism to use to communicate results to these processes. Additionally, they are a fast IPC mechanism.

The plugin which is responsible for interacting with the detection algorithm could however choose to use a different IPC mechanism. For example, one could implement a socket based solution and offload the computation of the transient and the run length to a remote host. This type of solution could be explored further by users who experience high EEM latency or find the process to be unresponsive running on a single machine.

Chapter Summary

There are several types of IPC which can be used to coordinate and communicate between many components of SAFE. This chapter explores three main IPC mechanisms: pipes, TCP sockets and UDP sockets. It is determined that a TCP based protocol is the best solution to communicate between the EEM and the simulation client. Pipes are then used to communicate both between the simulation client and the simulator, as well as between the EEM and transient and run length detection processes. These collected results which are communicated between the simulator, simulation client, and the EEM are eventually stored as described next Chapter 8.

Chapter 8

Storing and Accessing Results

Chapters 5, 6 and 7 focused on the architecture of SAFE which allows for the design and configuration of a simulation experiment, and later how the experiment is actually executed across all of the components of the SAFE framework. This architecture exists to collect data which can be stored and accessed for further analysis. This chapter focuses on the design considerations associated with storing and accessing the type and quantity of data collected during a large simulation experiment.

8.1 Databases

SAFE employs a **Relational Database System (RDBS)** to store and access all of the simulation data, as well as any associated meta-data. Before explaining how SAFE interacts with the database itself, it is best to provide a brief introduction to relational databases.

8.1.1 Theory

A relational database is composed of a set of **relations** where a relation is defined as a set of **tuples** over a fixed set of attributes. Each tuple represents a real-world object that is described through a unique assignment of values on the attributes. Each rela-

tion has a **key** used to identify the row. Most often, relations are organized as tables, where each tuple is stored in a row, and each column represents an attribute. [24]

For example, consider a table containing users for a web-based application. For every user in the system, there is a row in the table. This row stores information such as `id`, `username`, `first`, `last`, etc. My row in this database would be ⟨1, bcw006, Bryan, Ward⟩. This row is a single element in the set of all rows in the table. For an example of such a relation see Table 8.1b.

In larger, more complex systems, there are often many tables in the database. There can be complex relationships between rows in different tables [16]. These relationships are encoded through **foreign keys** which are used to relate one tuple with another. For an example of a simple relationship between two tables see Figures 8.1b and 8.1c. In this example, the foreign key `user_id` is used to reference the `Users` table from the `Purchases` table. In more sophisticated systems there can be tens or hundreds of tables which are used to store different kinds of data and complex relationships between such data.

UsersTransactions				
`id`	`username`	`first`	`last`	`price`
1	bcw006	Bryan	Ward	$10.00
2	bcw006	Bryan	Ward	$20.00
3	perrone	Felipe	Perrone	-
4	awh009	Andrew	Hallagan	$30.00

(a) A database schema which has not been normalized.

Users			
`id`	`username`	`first`	`last`
1	bcw006	Bryan	Ward
2	perrone	Felipe	Perrone
3	awh009	Andrew	Hallagan

(b) An example relation which encodes users in my research group.

Purchases		
`id`	`user_id`	`price`
1	1	$10.00
2	1	$20.00
3	3	$30.00

(c) An example table of several monetary purchases.

Figure 8.1: An example of database normalization. Redundant data in (a) can be extracted into a separate table as seen in (b) and (c). Furthermore, the database can be queried to `JOIN` the two tables to recover the data in the schema in (a).

A relational database **schema** is a formal description of the tables in the database. Database schemas are designed to store the information of a specific application. Many schemas however, require redundant data. This is problematic in that additional space is required to store the table on the physical disk, but worse yet, updates to the table must update all of the redundant data elements. The process of breaking such a schema into separate tables so as to minimize redundancy is called **normalization**. There exist several **normal forms** defined in **normalization theory**, which can be used to encode many typical types of relationships between data elements. [24]

For example, the table in Figure 8.1a contains redundant data. There are two rows for the user with `username` bcw006, one for each `Purchase`. If such a table were updated to change this user's username, this change would have to be applied to both rows 1 and 2. This schema additionally requires more storage space because the user information for bcw006 is stored twice. The process of normalizing this database would result in two separate tables as seen in Figures 8.1b and 8.1c. With this schema, a user's username can be updated, and it will automatically be applied to all purchases.

8.1.2 Database Management Systems

The mathematical theory of database organization has been implemented in many **Database Management Systems (DBMS)**. These systems facilitate all interactions with the database. Popular **Relational Database Systems (RDBS)** include the open source projects MySQL, PostgreSQL, and several commercial systems such as Microsoft SQL Server, and Oracal Database. These systems provide many capabilities above and beyond simply interfacing with data stored in relations.

One of the most important features of RDBS is the ability to interface with the system using the standard **Structured Query Language (SQL)**. This language allows users and programs to create and interface with tables controlled by the DBMS. SQL is a standard language which is implemented by all four of the previously mentioned RDBSs (with slight variations). A simple SQL query to find the row in the `Users` table from for my user by my username bcw006 can be seen in Listing 8.1.

```
SELECT * FROM users WHERE username = "bcw006";
```

Listing 8.1: A simple SQL `SELECT` statement used to query for a specific user in the Users table described in Figure 8.1b.

The SQL language has many additional powerful query features. For example, data can be queried from multiple tables as seen in Listing 8.2. Other features include sub-`SELECT` statements, `UNION` statements and Common Table Expressions (CTE). Details of such advanced features can be found in [16, 24] or standard texts specific to particular database engines.

```
SELECT u.*, p.*
FROM users AS u
LEFT JOIN purchases AS p ON p.user_id = u.id
```

Listing 8.2: A database query which when executed on the tables in Figure 8.1b and 8.1c would result in data formated as in Figure 8.1a.

DBMS provide a number of additional features which result in greater usability. Many SQL based RDBS include permissions systems and allow different users to query their contents. Furthermore, they allow for such users to connect to the database over a network and query the database. This can be used to help applications scale in that the database computations can be isolated on dedicated servers. Many enterprise SQL databases such as MySQL allow a cluster of computers to act as a database for applications with heavily database workloads.

Many DBMS also handle concurrency issues when multiple queries are simultaneously submitted to the database server. This allows multiple applications to interface with the database simultaneously which can be important in many use cases. Furthermore, an application can submit multiple queries to the database simultaneously, and the database will handle the queries properly.

8.2 SAFE's Database Schema

There are many data elements which must be stored in the SAFE database to allow for all of the functionality and credibility that we desire from SAFE. All of the experiment configuration information as well as the results are stored in the database such that an independent third party could replicate all aspects of the experiment. The schema used to store this information is illustrated in Figure 8.2.

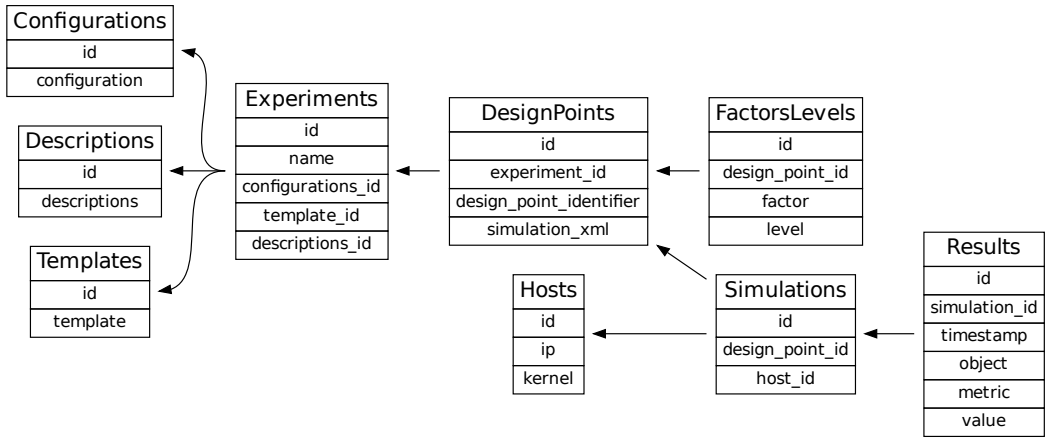


Figure 8.2: SAFE database schema.

The contents of the tables shown in this schema are as follows:

- `Configurations`: This table stores the experiment configuration files.
- `Descriptions`: This table stores the contents of experiment description languages.
- `Templates`: This table stores the contents of the templates used to generate the simulation model.
- `Hosts`: The hosts table is populated when a new simulation client registers as an available simulation host. This table stores information about each host's local environment.
- `Experiments`: This table contains information about specific experiments. It has foreign keys to `Configurations`, `Descriptions`, and `Templates` such that these files can be used for multiple experiments without data duplication.
- `DesignPoints`: An experiment is by definition composed of a set of design points. This table contains all of the design points, a unique `design_point_id` for each experiment described later in 8.3, and the XML model for the design point.
- `FactorsLevels`: A design point can have many factors, each with of which must be associated with a level. The `FactorsLevels` table stores these assignments.

- `Simulations`: For each design point in the experiment, many simulations are executed. This association is modeled with a foreign key to the `DesignPoints` table. Because each of these simulations are run on a single host, the `Simulations` table has a foreign key to the hosts table to model this association.
- `Results`: Each observation of a metric is stored in the `Results` table as a single row. For each result, we record the simulated timestamp of the result, the simulated object for which the result was gathered (e.g. which network node received a packet), the metric, and the value of that metric. Each result is then associated with the `Simulation` it was gathered from using the `simulation_id` foreign key.

The most important result of this database schema is its ability to store and access different types of simulation models. The SWAN-Tools project described in Section 4.4 hard-coded a single simulation model into both the database schema and the user interface. By normalizing the database schema and separating the factors and levels from the design points table, SAFE can be used with many simulation models.

8.3 Querying For Results

While the aforementioned database schema provides great flexibility in storing different types of models, it can also make querying for specific design points and their associated results challenging. The primary challenge is that for each factor for which a specific level is chosen, the `FactorsLevels` table must be `JOIN`ed. An example of such a query can be found in Listing 8.3. Each `JOIN` requires $O(n)$ time, and the product of all of these $O(n^j)$ `JOIN`s where n is the number of rows and j is the total number of joins. Thus this query can be slow when there are many factors and levels which are specified. Unfortunately, this is a common use case when analyzing simulation results.

One solution to this problem which was temporarily considered was to search the XML of the simulation model stored in `DesignPoints` table for specific factor's level values. While this approach does not require the query to `JOIN` the `FactorsLevels` table n times, it requires a full text search of every design point's XML in the experiment. Furthermore, the database engine cannot apply indices to accelerate such queries.

```
SELECT dp.*, r.*
FROM DesignPoints AS dp
LEFT JOIN Simulations AS s ON (s.design_point_id = dp.id)
LEFT JOIN Results AS r ON (s.id = r.simulation_id)
LEFT JOIN FactorsLevels AS f1 ON (dp.id = f1.design_point_id)
LEFT JOIN FactorsLevels AS f2 ON (dp.id = f2.design_point_id)
...
LEFT JOIN FactorsLevels AS fn ON (dp.id = fn.design_point_id)
WHERE dp.experiment_id = 1234 /* Some arbitrary experiment's id */
AND f1.factor = "param_1" /* Some arbitrary factor to filter by*/
AND f2.level = "123" /* Some arbitrary level value */
AND f2.factor = "param_2" /* Another arbitary factor */
AND f2.level = "456" /* Another arbitary level */
...
AND fn.factor = "param_n"
AND f2.level = "789";
```

Listing 8.3: An example of a query which returns results for a design point based on n levels.

The solution which I have developed for querying for results builds upon many of the ideas presented in the Design Point Generation Section, Section 6.5. The same algorithms used to prune the full factorial design to generate the appropriate design points can be applied to prune the full factorial design to find the specific design points to investigate further. These same algorithms can therefore return a set of design points for which to query the database for explicitly. One consequence of this approach is that more computation is required of the EEM. The computation of the specific design points in the result set therefore cannot be conducted by the database engine.

For this approach to be efficient, a mechanism is required to query for a specific design point. There is a unique `id` associated with each row in the database, but these values are determined by the database engine. The `DesignPoints` table therefore has another column which encodes a unique integer, the `design_point_id`, within the experiment which can be calculated based on the factors and levels in a specific design point. This integer can be computed for each design point in the set of design points for which to query. An efficient database query can then be easily constructed which returns the proper design points.

To describe further how the `design_point_id` is computed, it is necessary to establish a solid mathematical framework within which to work. Let $F_1, \ldots, F_n$ be factors of the model. Each of these factors has a set of levels denoted $L_1, \ldots, L_n$. Then the set of design points simulated, D, is a subset of the cross product of the levels, $D \subseteq L_{F_1} \times \ldots \times L_{F_n}$. Notationally, let $d \in D$ be a design point, and d_i be the level associated with factor i in the design point. The `design_point_id` is then the product of an injective function $f : L_{F_1} \times \ldots \times L_{F_n} \to \mathbb{N}$.

To construct this function, first we define the bijective function $g_i : L_{F_i} \to \mathbb{N}_{<|L_{F_i}|}$ which maps a level to the index of the level in the sorted list of levels. This function is then used in the definition of f:

$$f(d) = \sum_{i=1}^{n} \left(g(d_i) \prod_{k=1}^{n} |L_k| \right)$$

As it turns out, this function is bijective. The inverse of this function can be applied in the design point generation algorithm described in Section 6.5.2 to iterate through all design points easily.

This function is best understood through an example. In a 2^4 factorial design there are 16 design points. The function f assigns an ordering to these design points. A visual representation of how f orders design points can be seen in Figure 8.3. In this figure, each cell represents a design point with a unique level assignment for each of Factor 1 and Factor 2.

When the design points are generated and stored in the database, this function is computed and also stored. The database is setup with an index on this column, and the constraint that the unique `id` for the table and this `design_point_id` in tandem are unique. This allows for quick lookups. When querying for results, the design points of interest can be computed from $L_1, \ldots, L_n$, and then f can be applied to each tuple d. The database engine can then easily search for each of these design points.

	Factor 1			
Factor 2	1	2	3	4
	5	6	7	8
	9	10	11	12
	13	14	15	16

Figure 8.3: A visual explanation of how the function f maps design points down to $\mathbb{N}$.

Chapter Summary

This chapter briefly provides a background of relational databases, and demonstrates how they can be used. These relational databases offer many capabilities which make them very useful for storing simulation results. For these reasons, SAFE uses a relational database to store experiment configurations along with simulation results. This chapter also describes SAFE's database schema, as well as how the database can be queried to access the simulation results. Next, in Part III, we see how the SAFE architecture developed in Part II can be used in two case studies.

Part III

Applications and Conclusions

Chapter 9

Applications

Integrating SAFE with a specific simulator requires the development of a simulation client. As previously mentioned in Section 5.2, the simulation client has several important tasks which it must carry out to manage a simulation properly:

- Communicate with the EEM.
- Setup the simulation.
- Start simulator.
- Listen for results from the simulator.
- Terminate the simulator.

Depending upon the simulator being used, these tasks can be simple or complex. This chapter discusses how a simulation client can be developed to service two different simulators.

9.1 Case Study: A Custom Simulator

I developed a simple polling queues simulator which I then used to test of architecture and basic implementation of SAFE. Polling queues are a classic problem in

queuing theory. In such a system, there is a single server, and multiple queues each waiting to be processed by the server. There are many policies which can be used to determine which queue to service next, and for how long. There are many variations on this problem as well, such as queues of bounded or unbounded length, jobs or queues with different priorities, etc. [36]. Furthermore, different policies can be used in different applications to achieve different goals. For a visual depiction of a polling queues system, see Figure 9.1. Many polling queue configurations can be analyzed analytically, but this becomes increasingly difficult for complex service policies. Consequently, discrete-event simulation can be employed to more easily understand the behavior of such systems. I designed a simulator to integrate easily with SAFE, and consequently, the job of the simulation client is rather simple.

Similar to the EEM, the simulation client I developed for this language was written in Python [5] using the asynchronous, event-driven framework, Twisted [8] framework. This allowed for code reuse between the EEM and the simulation client in a few python modules. Additionally, it allows the simulation client to easily multiplex and respond to data from both the simulator and the EEM. The simulation client could have been developed in another language such as C, but Python and Twisted allowed for a faster development cycle.

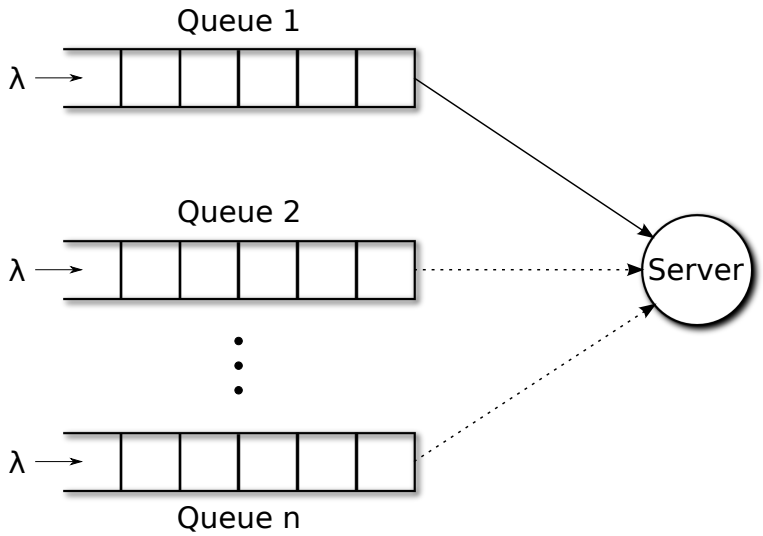


Figure 9.1: An example of a polling queues system.

The simulation client is responsible for configuring the simulator to run the desired design point. I designed my simulator to be configured through an XML based language. An example of this language can be found in Appendix A. This design

allows for the EEM to generate the XML model for the simulation, and pass it to the simulation client which can run it directly.

The simulation client must next spawn the simulation itself. In Twisted there is a method called `spawnProcess()` which is a wrapper around the necessary system calls to spawn a new process. This can be used to spawn a new process from an executable in the file system, such as the simulator. At a lower level in the Twisted library, this is implemented by calling `fork()` to spawn the new process, and `exec()` to replace the process with the simulator. Once this has been done, the simulator can begin execution and the simulation client returns to it's `select()` loop to wait for the next event to occur.

The `spawnProcess()` method also creates three standard pipes to the simulation client: Standard Input (`STDIN`), Standard Output (`STDOUT`), and Standard Error (`STDERR`). In the `spawnProcess()` library, this is done with the system calls `pipe()` to create these pipes and `dup()` to move them to file descriptors 0, 1 and 2 respectively. Either `STDOUT` or `STDERR` would be sufficient to pass results from the simulator to the simulation client; however, I created a new pipe on file descriptor 3 specifically for results from the simulator such that logging messages do not get misinterpreted as results. The simulator can then open this pipe using `fdopen(3)` and write to it with `write()`.

The simulation client can then communicate with the simulator itself over these pipes. A callback in the simulation client is installed to run whenever there is data ready to be read from the pipes from the simulator. In the simulator, when a job is serviced, its total wait time is computed, and that statistic is written to the pipe on file descriptor 3. The simulation client can then send that result to the EEM using a `Result` message.

The last responsibility of the simulation client is to properly terminate the simulation when notified by the EEM. This is done by sending a signal, which the simulator can handle to terminate the simulation. In Twisted, this can be easily done by calling `sendSignal()` on the object abstracting the simulator process.

The simulation client developed to integrate with my simple polling queues simulator demonstrates the basic functionalities that are required of a simulation client. Furthermore, when the simulator can be easily controlled by an external process than a simulation client can be developed relatively easily to allow the simulator to be used with SAFE. The development of a simulation client is made easier for future simulators in that they can follow the example set forward by this simulation client

implementation.

9.2 Case Study: *ns-3*

The previous simulation client demonstrates the simplest case of how a simulator can be integrated with SAFE. However, not all discrete-event simulation engines can be integrated this easily though with SAFE. This section describes the basics of how one uses the popular network simulator *ns-3* and the challenges associated with building a compatible simulation client.

9.2.1 *ns-3* Architecture

Simulation models in *ns-3* are encoded in C++ or Python scripts. These scripts rely heavily on the *ns-3* core libraries and allow the simulation user to model the system at a high level. On the other hand, a simulation model cannot be encoded in a text-based configuration file which the simulator accepts as input.

There is an additional module in *ns-3* called ConfigStore which allows a static *ns-3* simulation model to be configured through an XML based language. For example, the frequency at which wireless network nodes operate may be able to be configured through ConfigStore, but not the number of nodes in the simulation. For experiments which only vary attributes to simulation objects such as wireless frequency, ConfigStore can be used in conjunction with a constant simulation model. When the experiment must vary the simulation model, such as the number of nodes in the simulation, a new simulation script must be generated for each design point. At the Fall 2010 *ns-3* developer's meeting [11], there was discussion of possibly extending ConfigStore such that *ns-3* models could be described in XML.

Another feature of *ns-3* is that most tasks are executed using a software system called `waf` [10], which claims to be similar to `automake` [1]. This helps the user in compiling and running simulation scripts in the right environment. To an automation tool like SAFE, this adds an additional level of complexity in that `waf` spawns a new process for the simulation process itself.

Most *ns-3* users gather statistics from simulations by processing packet traces

which the simulator saves after execution. Another project under development, called the Data Collection Framework (DCF), funded under the same NSF grant as SAFE [20] seeks to ease the process of collecting data and statistics from within *ns-3* simulation. The DCF should ease the process of gathering statistics from *ns-3*.

9.2.2 *ns-3* Simulation Client

The *ns-3* simulation client receives the simulation model from the EEM just as the previous simulation client does. The challenge with *ns-3* is that the simulator cannot always be configured through XML. This introduces two use cases, one in which the *ns-3* simulation model must be generated in C++ or Python, and another in which the simulation can be configured using XML and ConfigStore. If the only levels being varied in the simulation model are attributes to different objects in the simulation (e.g. bandwidth) then the XML model generated by the EEM can simply be fed into ConfigStore and executed. In cases in which this is not sufficient, a C++ or Python based simulation script can be generated from a template and executed [19].

The *ns-3* simulation is spawned similarly to my previous simulator using the `spawnProcess()` method. The difference between the two simulators is that in *ns-3*`spawnProcess()` is called on `waf` which in turn spawns *ns-3*. Because *ns-3* is a child process of `waf`, it shares the open file table with `waf`, and thus has access to the same pipes from the simulation client. This allows statistics to be computed, particularly in the DCF, and written to the pipe on file descriptor 3. The DCF will also provide mechanisms to tag results for different metrics, which will be sent along with the result to the EEM for storage in the database.

The last complication of using SAFE with *ns-3* is that calling `signalProcess()` sends a signal to the `waf` process, not the simulator itself. The simulation client must therefore terminate not only the `waf` process, but also all child processes of `waf`. Once all of these processes have been terminated, the simulation client can request the next simulation.

Chapter Summary

This chapter discusses how SAFE can be used with two simulators. The first simulator discussed is a polling queues simulator which I developed to easily integrate with SAFE. The second simulator was *ns-3* which posed several challenges to integrate with SAFE. The investigation of these two cases demonstrates the considerations one must make to integrate with SAFE. Furthermore, they demonstrate the flexibility of the SAFE architecture in that it can be adapted for use with a few types of simulators. The next chapter concludes my thesis and describes areas for future work.

Chapter 10

Conclusions & Future Work

The complexity of the execution of proper simulation methodology has led to an overall lack of credibility observed in recently published simulation articles. This problem can be addressed through the automation of proper simulation workflow. SAFE builds upon many of the features seen in previous automation tools to provide a framework with which simulation users can conduct rigorous scientific simulation experiments.

Simulation automation enhances the usability of a simulator as well as the credibility of the simulation results. Users can configure their simulation through the XML based languages and execute their experiment with the assurance that the proper simulation workflow will be applied. The execution of the simulations themselves require no user interaction. Researchers can therefore focus on their science instead of their simulations.

SAFE's extensible architecture loosely couples the EEM and the specifics of the implementation of the simulator. SAFE can be made compatible with other discrete-event simulators by customizing a simulation client to the needs of the specific simulator. This gives SAFE a broad scope of application allowing simulation users in many problem domains to reap the benefits of the framework. It is important to note that SAFE cannot necessarily be integrated with all discrete-event simulators due to the challenges associated with setting up a design point and collecting results programmatically. For example, some proprietary simulators may be configured exclusively through a Graphical User Interface (GUI) which cannot be easily controlled in the simulation client. Simulators which can be easily configured through software using, for example, a configuration file can be more easily integrated with SAFE. Finally,

if a simulator is designed from the ground up to be used with SAFE, the challenges associated with integrating with SAFE can be minimized as seen in Section 9.1.

SAFE has been released as open source software under a General Public License (GPL), and it will be packaged with *ns-3* in future releases as part of the work conducted under Dr. Perrone's recent NSF grant [20]. This will give the project visibility to *ns-3* users around the world and allow for the continued development of the software. Additionally, this visibility will hopefully promote the use of my framework with other simulators.

My work on SAFE focuses primarily on its architecture and the execution of simulation experiments. This work provides a framework upon which many new features and capabilities can be built. The follow are a few proposals for future research.

The user interfaces currently provided by SAFE demonstrate the possibilities of the architecture and the project, though they are rather simplistic. Future work could be done to enhance the usability of these user interfaces, or expose new features of the framework through these interfaces. For example, the web manager could be extended to provide more sophisticated analysis tools. Another project could be to develop graphical user interfaces to help novice or intermediate users design their simulation experiments without needing to write XML.

For the entirety of this thesis, it has been assumed that individual simulation runs are run on a single processor. Many simulators, however, such as SWAN and *ns-3* can be run in parallel using MPI. An interesting area of future research is to develop algorithms to determine how many of the available processors on a machine should be employed per simulation run. For example, if a system has eight cores, it can run eight simultaneous simulations, or it can run a single simulation across all eight processing cores. A future area of research could investigate how to allocate computational resources either by the EEM or the simulation client to speed up the simulation experiment.

SAFE's EEM is currently only capable of managing a single simulation at a time. There are many applications in which it could be useful for the EEM to manage multiple experiments. For example, in the classroom it could be useful for a professor to setup a single EEM and allow students to create and execute their own simulation experiments. This introduces additional complexity in the EEM to determine which simulation in which experiment to dispatch. There could be many algorithms to determine how and when to dispatch design points to simulation clients to achieve better performance.

An interesting application of this functionality is the classroom. A single EEM could be run by an instructor and every student could be allowed to create their own experiments. In an educational setting this streamlined workflow can make simulation more accessible particularly in an undergraduate course. Furthermore, the computational resources available for running simulations could be more evenly distributed by the EEM to students in the course.

As the SAFE project evolves, I expect that new applications will spur the development of additional plugins so that the framework's functionalities are expanded to meet the needs of its users. This will allow SAFE to facilitate scientific achievements, in addition to being the subject of future scientific advancements in simulation automation, credibility and usability.

References

[1] Automake. Available at <http://www.gnu.org/software/automake/> [Accessed April 8, 2011].

[2] Cheetah – the Python-powered template engine. Available at <http://www.cheetahtemplate.org> [Accessed March 29, 2011].

[3] MATLAB – the language of technical computing. Available at <http://www.mathworks.com/products/matlab/> [Accessed April 17, 2011].

[4] GNU octave. Available at <http://www.gnu.org/software/octave/> [Accessed April 17, 2011].

[5] Python. Available at <http://python.org> [Accessed September 1, 2010].

[6] The R project for statistical computing. Available at <http://www.r-project.org/> [Accessed April 17, 2011].

[7] SciPy: Scientfic tools for Python. Available at <http://www.scipy.org/> [Accessed April 17, 2011].

[8] Twisted. Available at <http://twistedmatrix.com/> [Accessed April 8, 2011].

[9] XML Technologies, World Wide Web Consortium. Available at <http://www.w3.org/standards/xml/> [Accessed April 17, 2011].

[10] waf: The meta build system. Available at <http://code.google.com/p/waf/> [Accessed April 8, 2011].

[11] *ns-3* developers meeting, November 2010. Washington D.C., U.S.A.

[12] Matteo Maria Andreozzi, Giovanni Stea, and Carlo Vallati. A framework for large-scale simulations and output result analysis with ns-2. In *Proc. of the 2nd Intl. Conf. on Simulation Tools and Techniques (SIMUTools '09)*, 2009.

[13] Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, fourth edition, 2004.

[14] Adam L. Beberg, Daniel L. Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S. Pande. Folding@Home: Lessons from eight years of volunteer distributed computing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1. doi: 10.1109/IPDPS.2009.5160922. URL http://portal.acm.org/citation.cfm?id=1586640.1587707.

[15] Claudio Cicconetti, Enzo Mingozzi, and Giovanni Stea. An integrated framework for enabling effective data collection and statistical analysis with ns-2. In *Proceedings of the 2006 workshop on ns-2: the IP network simulator*, WNS2 '06, 2006.

[16] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, fifth edition, March 2006.

[17] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[18] Richard Fujimoto and David Nicol. State of the art in parallel simulation. In *Proceedings of the 24th Winter Simulation Conference*, WSC '92, pages 246–254, New York, NY, USA, 1992. ACM. ISBN 0-7803-0798-4.

[19] Andrew H. Hallagan. The design of an XML-based model description language for wireless ad-hoc networks simulations. Undergraduate Honors Thesis, Bucknell University, Lewisburg, PA, 2011.

[20] Thomas Henderson, L. Felipe Perrone, and George Riley. Frameworks for ns-3, 2010. Available at <http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=0958142> [Accessed September 1, 2010].

[21] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2006.

[22] J. Himmelspach, R. Ewald, and A.M. Uhrmacher. In *Proceedings of the 40th Winter Simulation Conference*.

[23] Christopher J. Kenna. An experiment design framework for the Simulator of Wireless Ad Hoc Networks. Undergraduate Honors Thesis, Bucknell University, Lewisburg, PA, 2008.

[24] Michael Kifer, Arthur Bernstein, and Philip M. Lewis. *Database Systems: An Application-Oriented Approach*. Addison Wesley, 2nd edition, March 2005.

[25] Stuart Kurkowski, Tracy Camp, and Michael Colagrosso. MANET simulation studies: the incredibles. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(4):50–61, 2005.

[26] Averill M. Law. *Simulation Modeling and Analysis*. McGraw-Hill, 3rd edition, 2000.

[27] S. Leye, J. Himmelspach, M. Jeschke, R. Ewald, and A.M. Uhrmacher. A grid-inspired mechanism for coarse-grained experiment execution. In *Distributed Simulation and Real-Time Applications, 2008. DS-RT 2008. 12th IEEE/ACM International Symposium on*, pages 7 –16, oct. 2008. doi: 10.1109/DS-RT.2008.33.

[28] Krzysztof Pawlikowski. Akaroa2: Exploiting network computing by distributing stochastic simulation. In *Proc. of the 1999 European Simulation Multiconference*, pages 175–181, Warsaw, Poland, 1999.

[29] Krzysztof Pawlikowski. Do not trust all simulation studies of telecommunication networks. In *Proceedings of the International Conference on Information Networking, Networking Technologies for Enhanced Internet Services*, pages 899–908, 2003.

[30] L. Felipe Perrone, Claudio Cicconetti, Giovanni Stea, and Bryan C. Ward. On the automation of computer network simulators. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*.

[31] L.F. Perrone, C.J. Kenna, and B.C. Ward. Enhancing the credibility of wireless network simulations with experiment automation. In *Proc. of the 2008 IEEE International Conference on Wireless & Mobile Computing, Networking and Communications*, WiMob '08), pages 631–637, 2008.

[32] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 3rd edition, 2003.

[33] Dragan Savić, Matevž Pustišek, and Francesco Potortì. A tool for packaging and exchanging simulation results. In *Proc. of the First International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '06, Pisa, Italy, 2006. ACM.

[34] Douglas C. Schmidt. *Reactor: an object behavioral pattern for concurrent event demultiplexing and event handler dispatching*, pages 529–545. ACM

Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995. ISBN 0-201-60734-4. URL http://portal.acm.org/citation.cfm?id=218662.218705.

[35] Abraham Silverschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts with Java*. Wiley, 7th edition, 2006.

[36] Hideaki Takagi. Queuing analysis of polling models. *ACM Comput. Surv.*, 20: 5–28, March 1988. ISSN 0360-0300.

[37] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, fourth edition, 2002.

Part IV

Appendices

Appendix A

Polling Queues Example XML Configuration

```
<simulation>
  <manager>
    <seed>1027</seed>
    <transient>10</transient>
    <termination>
      <time>100</time>
      <response>2</response>
      <min_samples>20</min_samples>
    </termination>
  </manager>
  <server>
    <service>
      <random_variable>
        <distribution>gauss</distribution>
        <mean>5</mean>
        <stdev>1</stdev>
      </random_variable>
    </service>
    <switch_queue>
      <random_variable>
        <distribution>gauss</distribution>
        <mean>1</mean>
```

```
        <stdev>0.1</stdev>
      </random_variable>
    </switch_queue>
    <next_job_time>
      <random_variable>
        <distribution>gauss</distribution>
        <mean>1</mean>
        <stdev>0.5</stdev>
      </random_variable>
    </next_job_time>
    <policy><model>longest</model></policy>
  </server>
  <queues>
    <count>5</count>
    <iat>
      <random_variable>
        <distribution>exponential</distribution>
        <lambda>0.166666</lambda>
      </random_variable>
    </iat>
    <length>10</length>
  </queues>
</simulation>
```

Appendix B

Example Experiment Configuration File

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

<configuration xmlns="http://www.bucknell.edu/safe/exp">

    <transient>
        <module name="simple">
            <metric>y</metric>
            <metric>z</metric>
            <samplelimit>234</samplelimit>
        </module>
    </transient>

    <runlength>
        <module name="simple">
            <metric>y</metric>
            <metric>z</metric>
            <samplelimit>234</samplelimit>
        </module>

        <!-- possibly multiple module elements -->
        <module name="simple">
            <metric>y</metric>
```

```
            <metric>z</metric>
            <samplelimit>234</samplelimit>
        </module>
    </runlength>

    <parser>
        <expparser>expparser.py</expparser>
        <modelparser>modparse.py</modelparser>
    </parser>

    <options>
        <simulator>ns3</simulator>
        <version>3.9</version>
    </options>

</configuration>
```

Appendix C

Example Cheetah Template

```
<simulation>
    <manager>
        <seed>1027</seed>
        <transient>10</transient>
        <termination>
            <time>100</time>
            <response>2</response>
            <min_samples>20</min_samples>
        </termination>
    </manager>
    <server>
        <service>
            <random_variable>
                <distribution>gauss</distribution>
                <mean>$service_mean</mean>
                <stdev>$service_stder</stdev>
            </random_variable>
        </service>
        <switch_queue>
            <random_variable>
                <distribution>gauss</distribution>
                <mean>0</mean>
                <stdev>0</stdev>
            </random_variable>
```

```
        </switch_queue>
        <next_job_time>
            <random_variable>
                <distribution>gauss</distribution>
                <mean>2</mean>
                <stdev>0.5</stdev>
            </random_variable>
        </next_job_time>
        <policy>
            <model>longest</model>
        </policy>
    </server>
    <queues>
        <count>$queue_counts</count>
        <iat>
            <random_variable>
                <distribution>exponential</distribution>
                <lambda>0.166666</lambda>
            </random_variable>
        </iat>
        <length>$queue_length</length>
    </queues>
</simulation>
```