MODEL BASED VALIDATION FOR IMPROVING AVAILABILITY OF INTERNET SERVICES

BY ANDREW TJANG

A dissertation submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Graduate Program in Computer Science

Written under the direction of
Thu Nguyen

and approved by

______________________________

______________________________

______________________________

______________________________

New Brunswick, New Jersey
October, 2014
ABSTRACT OF THE DISSERTATION

Model Based Validation for Improving Availability of Internet Services

by Andrew Tjang

Dissertation Director: Thu Nguyen

In studies separated by decades, operator mistakes have been identified as a significant source of unavailability in computer systems. Such mistakes can range from static misconfiguration to physical misplacement of wires and machines. Detecting and repairing these mistakes can often be time consuming and for many of today’s Internet services, unavailability results in significant loss of revenue and/or clients.

In this dissertation we present a series of tools that assist those who are charged with designing and creating the infrastructure that support those Internet services to mitigate the results of operator mistakes. Specifically, we first propose an assertion based language, A, that is a formalized specification of correct behavior and can be used to bolster system understanding, as well as help to flag operator mistakes in a distributed system. We look at examples of these mistakes, their effects and manifestations in both an academic environment and in real-world applications. With a good understanding of these mistakes and behaviors, we design a process to validate operator actions called Model Based Validation. We then explore methods to simplify the assertion writing process using machine learning techniques. With such a large attribute space and such small data sets, we investigate a variety of optimizations including a refinement loop and various filtering algorithms.
Acknowledgements

This dissertation would not have been possible without the support, encouragement, and hard work of many people.

I would like to thank my primary research advisor, Thu Nguyen, for his incredible patience, expertise, and advice he has imparted on me for the tenure of my studies at Rutgers University. He has often gone beyond his responsibilities as a mentor to ensure my well being as both a student and an educator. I am honored to have been able to work with him in both capacities.

Over the course of my studies, I have also had the privilege of having Rich Martin and Ricardo Bianchini serve as co-advisors. They have shared much of their wisdom selflessly. Their ideas and discussions have undoubtedly shaped much of the body of work that I present here. While working on the last parts of this dissertation, I have been equally lucky to have been able to work with Vinod Ganapathy, whose ideas have sparked the development of ADIVA.

I can recount all the sleepless nights spent debugging and pontificating with my colleague and friend, Fabio Oliveira. Our collaboration over the many years has been an extraordinary experience for me. His work ethic and meticulousness has often inspired me to be a better student and researcher. He was never short on words of encouragement.

To the wonderful people of Dark Lab - Fabio, Rekha, Wei, Ana Paula, and Qingyuan, thank you so very much for adopting me and accepting me as one of your own. From the humble beginnings of what really was a darkroom, Dark Lab was never hesitant to offer me a seat. I am forever grateful for your kindness.

For a year and a half, my colleagues at Ask.com have provided me with great insight into the workings of a real commercial search engine. Tao Yang, Hong Tang,
and Srinath Rao welcomed me into their organization and gave me the freedom and resources I needed to succeed. This work would not have been possible without them.

This long journey, from beginning to end, would certainly not have been possible without the love and encouragement of my family and friends.

To Kourtney, Tanuj, Vera and Max: my sanity is only here because you were there for me.

To my parents: you have always given me everything - supported every one of my endeavors, and have loved me unconditionally. I can never repay you for all that you have done for me.

To the love of my life, Jessica: my wife, my rock, my bear. Your never-ending patience and love have been essential to my life. Words are not sufficient to express my gratitude and love, but I will spend the rest of my days trying to show you.
# Table of Contents

Abstract ................................................................. ii  

Acknowledgements ......................................................... iii  

List of Tables .............................................................. ix  

List of Figures ............................................................. x  

1. Introduction .......................................................... 1  
   1.1. Cost as a Motivating Factor ........................................ 1  
   1.2. Current Monitoring and Verification Tools ....................... 3  
   1.3. Motivation For Our Approach ..................................... 4  
   1.4. Introducing $A$ ..................................................... 5  
   1.5. Evaluating Model-Based Validation ............................... 6  
   1.6. A Case for Machine Learning ................................... 7  
   1.7. Roadmap ........................................................... 8  
   1.8. Contributions ..................................................... 9  

2. Related Work ......................................................... 11  
   2.1. Validation ........................................................ 11  
       2.1.1. Background .................................................. 11  
       2.1.2. Validation Strategies ..................................... 14  
       2.1.3. Why Validation? ............................................ 15  
   2.2. Formal Modeling and Monitoring ................................. 15  
   2.3. Machine Learning ............................................... 20  

3. The Model Based Validation Framework ............................ 22
3.1. The A Assertion Language ......................................... 22
3.2. The A Assertion Language ......................................... 22
  3.2.1. Overview ..................................................... 22
  3.2.2. Example System, Mistake, and Program ..................... 24
  3.2.3. Elements ..................................................... 24
  3.2.4. Assertions ................................................... 27
  3.2.5. Libraries ..................................................... 29
  3.2.6. Tasks ........................................................ 31
3.3. Model ............................................................ 32
3.4. Example Models .................................................... 32
  3.4.1. Access Control Model ....................................... 32
  3.4.2. Hierarchical Component Model .............................. 33
  3.4.3. Data Flow Model ............................................ 33
3.5. Runtime .......................................................... 36
3.6. Summary .......................................................... 37

4. Case Studies and Evaluation ......................................... 38
4.1. Ask.com Crawler ................................................... 38
  4.1.1. Crawler and Validation System ............................. 38
  4.1.2. Ask.com Runtime ............................................ 40
  4.1.3. Effectiveness ............................................... 41
  4.1.4. Performance Overhead ...................................... 44
  4.1.5. Summary ..................................................... 45
4.2. Auction Service and Validation System .......................... 46
  4.2.1. Experimental Setup ......................................... 46
  4.2.2. Auction Service Runtime ................................... 47
  4.2.3. Monitors ..................................................... 48
  4.2.4. Configuration Parameters .................................. 49
  4.2.5. Creating and Binding A Elements .......................... 50
List of Tables

4.1. A program broken down into types .............................................. 46
4.2. Mistake-injection experiments. .................................................. 54
5.1. Example Predicates for Refinement ............................................ 76
5.2. Attribute Selection in ADIVA .................................................... 83
5.3. Attributes in Manual Rules ....................................................... 84
5.4. Summary of Results ................................................................. 86
5.5. The Effect of Filtering on ADIVA .............................................. 89
5.6. The Effect of Attributes on ADIVA .......................................... 91
List of Figures

2.1. Validation with an auction service ........................................ 13
3.1. A Partial Data Flow Graph .................................................. 33
3.2. Partial flow model for the crawler. ........................................ 34
4.1. The Ask.com A runtime system. ............................................. 40
4.2. Architecture of the A runtime system .................................... 49
4.3. Erected Barriers ................................................................. 62
5.1. The Bigger Picture ............................................................... 67
5.2. The general flow for training ................................................. 68
5.3. A Generated Machine Learning Rule Set ............................. 71
5.4. Rule set after feedback ....................................................... 73
5.5. Manual Ruleset for our Migrate Database example task ........ 85
Chapter 1
Introduction

Computer users are becoming dependent on Internet services, such as search engines, e-mail, and music jukeboxes for their work and leisure. Internet services have become so pervasive in people’s lives that they have come to expect constant availability. Businesses are also reliant on Internet services for critical business functions. Their daily operations are dependent on the availability of payment processing services, user authentication services, and e-commerce portals. These services are delivered using large and complex distributed systems. Unfortunately, reasoning about correct behavior of such systems is difficult because these systems are comprised of complex conglomerates of distributed hardware and software, such as load balancers, loggers, databases, and application servers. Thus, ensuring high service availability in today’s Internet services is as challenging as it is critical([2,3]).

1.1 Cost as a Motivating Factor

Our work seeks to alleviate one important source of service failures: operator mistakes. Several studies have shown that mistakes are a significant source of unavailability [2–6]. For instance, Oppenheimer et al. [2] show that mistakes were responsible for 19-36% of failures, and, for 2 out of 3 services, were the dominant source of failures and the largest contributor to time to repair. Similarly, Oliveira et al. [6] report that operator mistakes are responsible for a large fraction of the problems in database administration. However, the problem of operators causing unscheduled downtime is not limited to those organizations in the Information Technology sector. Murphy and Levidow [5] show us that even in enterprise server environments, found in all business sectors, operator action can be a significant cause of downtime. Patterson et al. [3] show a whopping
<table>
<thead>
<tr>
<th>Sector</th>
<th>Cost Range/hr</th>
<th>Average Cost/hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brokerage Operations</td>
<td>$5.6-$7.3 million</td>
<td>$6.5 million</td>
</tr>
<tr>
<td>Credit Card or Sales</td>
<td>$2.2-$3.1 million</td>
<td>$2.6 million</td>
</tr>
<tr>
<td>Home Catalog</td>
<td>$60,000-$120,000</td>
<td>$90,000</td>
</tr>
<tr>
<td>ATM Fees</td>
<td>$12,000-$17,000</td>
<td>$14,500</td>
</tr>
</tbody>
</table>

Table 1.1: Summary of Costs associated with system downtime [1].

59% of blocked calls were due to operator mistakes. All of these studies corroborate an older study of Tandem systems by Gray, where mistakes were a dominant reason for outages [4]. The irony of the situation is that in the intervening years between Gray’s work and those mentioned above (nearly two decades), with all the advances in technology, operators still account for a good percentage of system downtimes.

If we then accept that operators do cause a significant percentage of the unplanned downtime, it behooves us to examine the costs associated with the downtime. Industry whitepapers [1,7] seem to agree that the costs associated with downtime vary depending on sector, but can range from $28,000 to $5,400,000 per hour for IT related industries, and can even exceed that amount for financial industries. Table 1.1 shows a summary of costs per sector due to downtime. In 2009, it was estimated that a 1 hour outage at Paypal resulted in lost transactions totaling $7.2 million dollars [8]. Some of the costs associated with this downtime is less obvious. In a survey conducted by Computer Associates [9], approximately 1800 organizations of varying sizes provided information as to how downtime affects their operations. An estimated 1.3 billion man hours were lost over the course of a year through North America and Europe. Nearly 50% of respondents indicated that the outages caused ranged from quite damaging to disastrous effects on their reputations, and 35% admitted that customer loyalty was adversely affected. A third of respondents would not be able to meet compliance and regulatory commitments due to downtime.

Clearly reducing the amount of downtime due to operator mistakes can make an impact on these costs.
1.2 Current Monitoring and Verification Tools

For years, administrators have developed individual, ad-hoc methods of managing operator mistakes and the resulting system errors. Indeed, often significant IT budgets are devoted to developing, maintaining and managing in-house software to operate large-scale systems. We believe that a language specifically tailored to the task of operator-system interaction could improve this process significantly. Instead of a collection of ad-hoc scripts tailored to a specific site and environment, a language-based approach will allow a concise, understandable, and verifiable method of specifying correctness. Codifying the process in a specific language would also ease the generalization and parameterizing of such tasks, thus allowing for the re-use of software to manage operator interactions across different organizations. For example, an organization could re-use existing parameterized libraries to manage the interactions with load balancers, databases and application servers instead of having to develop custom scripts as they do today.

Assuring correct behavior of distributed systems is often assigned to an organization’s system administrators. The Internet is littered with monitoring and alerting scripts that can assist them with the monumental task of keeping a smoothly operating system. Anecdotal evidence suggests the coverage and behavior of these tools is often random and chaotic in nature. Their usage and output leave many things open to interpretation and even the smallest of organizations may have many of them cobbled together to create a suitable solution. In any event, these system administration scripts suffer from a number of problems that show them to be sub-optimal for ensuring correct behavior of distributed systems. 1) They lack continuity: the same script used to monitor and identify problems in network traffic may vastly differ from a script used to monitor CPU utilization, even though one may be tightly related to the other. 2) Manageability/Readability: Adding new alerts or monitors can involve editing multiple scripts/tools. Determining what components, alerts, monitors, exist also involves multiple scripts/tools in multiple formats. 4) Interpretation: Many tools provide the means by which information is gathered, but interpretation of this information is largely left
to a person. Different people have differing opinions of correctness.

1.3 Motivation For Our Approach

Our approach seeks to solve these problem areas by providing a clear and concise method for system designers and administrators to formalize policies of correct behavior.

In our previous work, we proposed operator action validation as an approach for detecting mistakes while hiding them from the service and its users ([6, 10]). In that approach, a validation framework creates an isolated extension of the online service in which operator actions are performed and later validated. Before the operator acts on a service component, the component is moved to this extension. After the operator activity is completed, the correctness of the operator’s actions is validated by comparing the behavior of the component with that from either a trace or an online component. If validation succeeds, the system moves the component back online; otherwise, it alerts the operator. While this validation strategy can detect and hide a large class of mistakes, it has three important limitations: (1) it requires known instances of correct behavior for comparison; (2) it provides no guidance in pinpointing mistakes; and (3) it fails to detect latent mistakes.

We propose a novel validation strategy, called model-based validation, that addresses these limitations. Model-based validation calls for service engineers\(^1\) to choose abstract models to describe the systems and identify incorrect configurations and behaviors. These models are then used to guide the specification of assertions to check the correctness of operator actions without requiring instances of correct behaviors for comparison. The purpose of the models is to ensure a systematic and proactive approach to generating assertions, rather than an ad-hoc/reactive approach that may leave many mistakes undetected.

Our approach of model-based validation contrasts with the prevailing approach of developing a collection of ad-hoc scripts for verifying the proper behavior of a service. As its namesake implies, model based validation extends the work of Nagaraja et al.

---

\(^1\)“Service engineers” are the people who design and implement a service, whereas “operators” are those responsible for its day-to-day management.
on a technique of verifying distributed system correctness called validation. The key concept to validation techniques is that effects of operator mistakes should be contained so that they are not exposed to users, and that system properties are as close to real as possible. Chapter 2.1 describes, in detail, the strategies employed to satisfy these requirements.

Model based validation, like other validation techniques, also seeks to verify correctness in real environments without exposing operator mistakes. However, it seeks to solve a known problem with other validation techniques. In order to mimic real system behavior, previous validation techniques required instances of already correctly behaving components with which to compare those chosen to be verified. This leaves components that have no basis of comparison vulnerable - as their behavior cannot be compared. This is most often the case with non replicated components and introduction of new components. Model based validation seeks to provide system designers the tools needed to specify correctly behaving components in these specific instances.

1.4 Introducing $A$

Our technique of model-based validation is comprised of a domain specific language (DSL), $A$, its runtime system, and a set of models and assertions about correct service behavior. Although $A$ does impose a learning curve on the service designers, we believe that an assertion language specifically tailored to operational tasks (and mistakes) combined with validation can improve service operation tremendously. Studies examining the development and use of domain specific languages [11,12] describe their trade offs and benefits. We believe that a domain specific language like $A$ exemplifies many of the beneficial attributes.

In contrast with the ad-hoc scripts, assertions provide a concise, understandable, and verifiable method of specifying correctness and comparing behaviors before and after operational tasks are performed. [12] contends that one of DSL’s benefits is that: “DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate,
modify, and often even develop DSL programs.”

Furthermore, the assertions developed for a service can be parameterized and re-used by services with similar structure at different organizations, again instead of today’s site-specific scripts. The assertions in A are expressive enough to encode multi-component dependencies and simple enough to be easily and readily readable and editable. A features elements, element groups, dynamic bindings, and configuration structures - all of which make great strides in simplicity and uniformity.

1.5 Evaluating Model-Based Validation

To demonstrate and evaluate our approach, we have built a prototype model-based validation system for the Ask.com Web crawler, a system that contains a diverse set of software components replicated across hundreds of machines. While the crawler is not a part of the online search engine, it provides a meaningful evaluation platform for model-based validation because it needs to run 24x7 to keep Ask.com’s snapshot of the Web as fresh as possible. Also, the crawler interacts with the Web at large and so operational mistakes (and/or software bugs) can have undesirable business consequences.

While we were implementing the validation system, we were also recording problems encountered during the final testing runs of the crawler after an operator action (e.g., a software update) for a period of about 8 months. Using the detailed logs from these test runs, which were essentially validation runs, we show that our validation system would have quickly detected 5 of the 6 problems that could have had real-world impact.

We also implemented a prototype model-based validation system for an academic yet realistic online auction service [13]. This service is smaller in scale than the Ask.com system but allows us to evaluate model-based validation more extensively using mistake injection. This system detected 10 out of 11 plausible yet not easily anticipated mistakes injected during a variety of operator tasks. All of these would have been missed using our previous validation strategies. Model-based validation would also have detected all 28 previously observed mistake instances that our previous validation strategies were able to detect [10].
The above implementations have three main parts. (1) A small set of models that we decided were simple yet allowed clear description of system correctness. We describe these models in Section 3.4.3. (2) An exploratory language called $A$ designed to make it easy and convenient to express correctness assertions derived from the models. We briefly describe $A$ in Section 3.2. And (3) a runtime system that maps the monitored states of a service to $A$ program objects and runs $A$ validation programs. We describe this runtime system in Section 3.5.

We present our evaluation of model-based validation and discuss our experience in Chapter 4.

1.6 A Case for Machine Learning

Because model based validation is only as good as the models that designers create, we hope to assist designers in strengthening their understanding of their systems. To that end, we also introduce an approach to identify modes of incorrect behavior (i.e. mistakes) in a systematic way that combines both domain specific knowledge, as well as learned behaviors from observed behaviors. We attempt to use machine learning techniques in concert with domain specific knowledge to assist system designers with discovering attributes and values for use with model based validation. We also study the strengths and limitations of such an approach, as well as explore the state space of operator tasks and mistakes.

In order to facilitate the creation of models, we explore using machine learning techniques to assist system designers in sifting through the necessarily large state space. The monitored data encompasses almost 1000 attributes covering state representing operator commands, files, and processes.

While machine learning techniques usually rely on large amounts of training data, human operator experimental data is rare. Because of this, we propose a series of techniques to gain useful information from machine learning with minimal training data. We propose a new technique of combining system designer expert knowledge with the machine learner in a feedback loop. This method strengthens designer understanding,
as well as assists the machine learner with producing more qualitatively correct results. We also compare and contrast other methods of assisting the machine learner such as attribute filtering and synthetic generation of additional data.

1.7 Roadmap

This dissertation will describe the methods we propose to mitigate the effects of operator error on distributed systems. We detail each aspect of the tools and infrastructure of this system that we call Model Based Validation.

We begin by describing the research on which this work is based. Chapter 2 briefly introduces trace based and replica based validation as methods of mitigating operator mistakes, and goes on to survey other works in systems monitoring and management, formal modeling, and machine learning.

We then delve into the bootstrapping problem that occurs when validating systems that don’t have available replicas and traces with which to compare, and our solution of model based validation. Chapter 3 explains a domain specific language called $A$ that can be used by system engineers to describe system correctness. It provides support for operator task centric modeling and monitoring. We also describe the associated system infrastructure implemented.

Chapter 4 utilizes the theory and implementation described above in two distinct case studies. The first is a three tiered system on which we have computer operators performing various maintenance tasks. The second is a study of commercial search engine’s development group, where operator mistakes had the potential to have real life costs. In both cases, we examine the efficacy of the model based validation approach.

The models used in the case studies required a large amount of domain specific knowledge. Writing correctness models complete enough for our purposes was a non trivial task. Chapter 5 explores methods of using machine learning to assist in the creation of these models. We present multiple ways of learning important attributes from collected data.

We present our concluding remarks in the final chapter, Chapter 6.
1.8 Contributions

This dissertation makes the following contributions:

- We introduce a new Domain Specific Language, A, for describing correctness of a distributed system. The language design acknowledges the role of the human operator, and provides direct support for identifying operator errors through the constructs of task-aware assertions.

- We design and implement a runtime environment to monitor system state and evaluate assertions in several distinctly different types of distributed systems. These include a 3 tiered web service, enterprise network infrastructure, and a commercial search engine.

- We extend the concept of validation as it pertains to a series of operator tasks on selected distributed systems when no existing model of good behavior exists for comparison. To bootstrap the validation, we create comprehensive, but necessarily incomplete, models to describe correct behavior.

- We present the results of operator experiments where human operators of varying degrees of expertise interact with systems to perform predefined tasks. We evaluate the efficacy of our models created above in identifying cases of operator mistakes that potentially result undesired system behavior.

- We also present a case study of human operator error as witnessed in a commercial search engine, their perceived effects, as well as the models/infrastructure implemented to catch these mistakes.

- We examine model based validation as a component of mistake-aware systems management, whereupon the correctness models are used to allow or deny operator actions to prevent further harm to systems.

- We propose a method of using machine learning techniques to assist in the generations of correctness models. We feed the traces of known good and bad traces
to the machine learner in an effort to identify the attributes that are important to defining correctness.

- We explore the data space of all attributes involved in the performance of human operator tasks and propose different methods to increase the quality of the machine learning results.
Chapter 2

Related Work

2.1 Validation

Validation is a technique of checking system correctness. Specifically, we seek to validate operator actions, with the ultimate goal of detecting and preventing the effects of operator mistakes from reaching users. The mechanism by which this is achieved is described below.

2.1.1 Background

The core idea of validation is to verify operator actions under realistic workloads in a realistic but isolated validation environment [10]. Mistakes can then be caught before becoming visible to the users. To achieve realism, the validation environment is hosted by the online system itself (Figure 2.1). In particular, a service with validation is dynamically divided into two slices, an online slice that hosts online components and a validation slice where components can be operated on and validated before being re-integrated into the online slice. The validation slice contains a testing harness that can be used to load the components to be validated and to check their correctness. To achieve isolation, the components placed in the validation slice are masked (isolated) from the online slice using layer 2 and 3 virtual networking. Server nodes can be moved between slices without changing configuration parameters of the nodes or the software components that they host.

Hosting the validation environment in the online system makes it possible to: (1) avoid latent errors that escape detection during validation but become activated in the online system because of differences between the validation and online environments;
(2) load components under validation with as realistic a workload as possible; and
(3) enable operators to bring validated components online without having to change
any of the components’ configurations, thereby minimizing the chance of new operator
mistakes. On the other hand, the components under validation, which we shall call
*masked* components for simplicity, must be *isolated* from the online system so that
incorrect behaviors cannot cause system failures.

To host the validation environment on the online system itself, we divide the com-
ponents into two logical slices: an online slice that hosts the online components and
a validation slice where components can be validated before being integrated into the
online slice. Figure 2.1 shows this validation architecture when a component of the
Web server tier is under validation. To protect the integrity of the online service with-
out completely separating the two slices (which would reduce the validation slice to an
offline testing system), we erect an isolation barrier between the slices but introduce
a set of connecting *shunts*. The shunts duplicate requests and replies (i.e., inputs and
outputs) passing through the interfaces of the components in the live service. Shunts
either log these requests and replies or forward them to the validation slice.

We then build a validation harness consisting of *proxy components* that can be
used to form a virtual service around the masked components; Figure 2.1 shows an
application proxy being used to drive a masked DBMS. Together, the virtual service
and the duplication of requests and replies via the shunts allow operators to validate
masked components under realistic workloads. In particular, the virtual service either
replays previously recorded logs or accepts forwarded duplicates of live requests and
responses from the shunts, feeds appropriate requests to the masked components, and
verifies that the outputs of the masked components meet certain validation criteria.
Proxies can be implemented by modifying open source components or wrapping code
around proprietary software with well-defined interfaces.

Then validation works as follows. Suppose an operator needs to operate on a service
component (e.g., to upgrade its software). Before starting, the operator uses a script to
move the server hosting the component from the online slice to the validation slice. This
takes the node offline and completely masks it from all online components. All requests
The auction service is a three-tiered distributed system comprised of a web server tier, an application server tier, and a database tier - each with replicated components. In this particular case, a single component, a web server, is being validated inside the validation slice. The validation harness uses one or more client proxies to load the Web server and one or more application server proxies to field requests for dynamic content from the Web server. The proxies can either use previously logged data or a duplicate of the current workload of a functionally equivalent component in the online slice to load the Web server being validated.

that would be sent to the component are redirected to components that provide the same functionality but that are unaffected by the operator action. The operator can now work on the component without affecting the online system. After the operator action has been performed, the affected component is brought back online but is placed in the validation slice and connected to a validation harness. The validation harness consists of a library of real and proxy components that can be used to form a virtual service around the component under validation. The harness requires only a few machines and, thus, has negligible resource requirements for real services. Together, the isolation between the online and validation slices and the validation harness prevent the component, called masked component, from affecting the processing of client requests while providing an environment that looks exactly like the live environment and give the illusion that the masked component is in a complete system.
The system then uses the validation harness to compare the behavior of the component affected by the operator action against that of a similar but unaffected component. If this comparison fails, the system alerts the operator before the masked component is placed in active service. The comparison can either be against another live component, or against a previously collected trace. After the component passes the validation process, it is migrated from the sand-box into the live operating environment without any changes to its configurations.

2.1.2 Validation Strategies

In [10], Nagaraja et al. proposed two validation approaches: trace-based and replica-based validation. In trace-based validation, for each masked component to be validated, requests and replies passing through the shunts of an equivalent live component are logged and later replayed. During the replay, the logged replies can be compared to the replies produced by the masked component. In replica-based validation, the current offered load on the live service is used, where requests passing through the shunts of an equivalent live component are duplicated and forwarded in real-time to the validation harness to drive the masked component. The shunts also capture the replies generated by the live component and forward them to the harness, which compares them against the replies coming from the masked component.

In [6], Oliveira et al. described trace-based and replica-based validation for database servers, whereas Tan et al. [14] for replica-based validation for file servers. Oliveira et al. also proposed a primitive version of model-based validation in [6]. The idea was to have the administrator describe her future actions on the masked database at a high level, and compare the schema resulting from the actions with the schema that would be expected if all the actions were correctly performed. The expected schema then represents the model against which the actions are validated. Here, we extend our original proposal significantly by applying model-based validation to entire Internet services.

Unfortunately, trace-based and replica-based validation are only applicable when the output of a masked component can be compared against that of a known correct
instance. Many operator actions can correctly lead to a masked component behaving differently than all current/known instances, posing a bootstrapping problem. An example in the context of databases is a change to the database schema (a task that is cited as one of the most common DBA tasks in our survey). After the DBA changes the schema (e.g., by deleting a column) in the validation environment, the masked database no longer mirrors the online database and so may correctly produce different answers to the same query. The same applies to a previously collected trace. Thus, in Chapter 3, we will introduce Model Based Validation, which seeks to resolve these deficiencies.

2.1.3 Why Validation?

Validation is designed to address a serious issue in traditional testing (which we call offline testing). Specifically, testing environments tend to drift from the online environments over time. For example, 84% (i.e. 42 out of 51) of database administrators responding to a survey reported that they typically test their actions in environments that are different from their production systems [6]. Also, it is often difficult to apply realistic workloads in an offline testing environment. Thus, even with careful testing, operators can make mistakes when changing or deploying their changes to the online system. Validation closes this gap between offline testing and the online system, although the two approaches can be complementary: validation could be applied as the last step in a testing/validation process before exposing an operator action to the online system.

2.2 Formal Modeling and Monitoring

Related works fall into two main categories: formal modeling and runtime monitoring.

**Formal modeling.** Within the scope of formal modeling (e.g., enumerable states and transitions), there are many strategies to verify source code, one of which is model-checking [15]. A key distinction of our work is that it validates components using both static configuration and dynamic state information, rather than static source code.

Although both our approach and formal languages (such as Z [16]) model systems
as a set of valid states separated by transition functions, we do not try to model all possible valid states. Because the state of an A program represents a running system, not a theoretical model, it is faced with observability issues that do not exist in formal languages. In general, our approach is more domain-specific and favors practicality and programmability over the provability properties favored by formalisms.

There are general-purpose languages that many have adapted to formalize various real-time and distributed systems. Bowman and Derrick [17] applied the formal language Z to address consistency checking for Open Distributed Processing standard.

Lola [18] embodies a specification language and algorithms for online and offline monitoring of circuits and embedded systems. The language can describe assertions about system correctness along with statistical measures obtained from input data streams.

**Runtime Monitoring.** Given that formal modeling for verification of large systems is unfeasible, alternatives based on runtime monitoring have been proposed. Kim et al. [19] proposed a framework to specify the formal requirements of a program using Linear Temporal Logic, extract information from the program’s execution, and check runtime observations against the specified properties. Sammapun et al. [20] later extended this work by including the capability of verifying timeliness to check real-time systems. Also in the realm of real-time systems, Mok and Liu [21] proposed an approach to specify timing constraints, monitor them, and catch violations.

The prior work on systems management in Internet services can be divided into five main classes: automation, recommendation, validation, recovery/undo, and monitoring/auditing. In the first class, systems typically reduce operator intervention by automating repetitive tasks, e.g., [22–24]. Unfortunately, many tasks cannot be automated, creating the possibility of operator mistakes. Recommendation systems, e.g., [25], attempt to prevent mistakes by guiding the operators’ actions. As an extra safety net, previous validation systems ([6, 10, 14]) hide and detect certain types of mistakes by confining the operators’ actions to a sandboxed environment. When mistakes are made, recovery/undo systems ([23, 26–28]) can be used to bring the service (or the sandbox) back to a proper state. The undoable email store described in [26]
provides operators, who are on the front lines of ensuring system dependability, a tool to roll back undesirable or unintended system changes while maintaining data integrity. Their runtime infrastructure collects operator transactional data, which is then used to facilitate the rewinding of system state should the need arise. Operator Undo is a tool for use after an operator has detected a problem, and thus after the results of his/her mistakes have been made visible to end users.

Finally, monitoring/auditing systems, e.g., [29–36], attempt to detect (and sometimes diagnose) performance and behavioral problems regardless of their root causes.

Our work is orthogonal and complementary to recommendation and recovery/undo systems. It is also related to monitoring/auditing systems like PSpec [33], Pip [34], and D³S [36] that use assertion checking to help debug software, with the latter two focusing on distributed systems. However, because they are concerned with detecting bugs, and not operator mistakes, these systems did not consider some important static and structural issues, such as improper system configurations and latent security problems.

PSpec [33] and Pip [34] use assertion checking for performance debugging.

In PSpec, programmers specified performance expectations of given system. A runtime system consists of instrumentation that emits an event log file on which a checker can run the assertions. Similarly, Pip introduced an infrastructure for debugging distributed systems. Pip presents a behavior model that uses paths derived from recorded events of annotated software components. These events are then compared against the programmer-declared expectations to determine when and where unexpected behavior occurs.

The authors list elapsed time, throughput, utilization as some of the performance metrics for which assertions can be written. The focus of the work included parallel programming languages on multiprocessor systems, where performance bugs could be hard to catch using conventional techniques.

Additionally, PSpec provides programmers with a way of solving for unknown performance expectations. PSpec was successful in finding performance bugs in the runtime system on which it was tested.

A suite of analysis tools accompany Pip and help to find the root causes of the
unexpected behavior. Like Pspec, Pip was able to find multiple instances of unexpected behavior in the distributed applications to which it was applied.

Model-based validation addresses these issues, as well as the performance problems targeted by PSpec and Pip. We argue, however, that these systems are not sufficient to be applied to the realm of “bugs” caused by operator mistakes. There is a static nature to performance bugs caused by software components that simply does not exist during execution of an operator tasks. Our work addresses the dynamic environment of these scenarios. Both expectation works also ignore a huge portion of application behavior and concentrate largely on performance metrics of running systems. The A and associated runtime fully support analysis of static configuration information used by components of distributed systems. By allowing programmers the flexibility of choosing which dynamic metrics to monitor, as well as which configuration parameters are most important, A programs can better model the actual systems and tasks that operators perform. Operator tasks are first-class concepts in model-based validation in general and in our assertion language in particular, allowing us to detect these problems and relate them to specific tasks. Besides its focus on operator mistakes, model-based validation differs from other assertion-checking efforts [37] in that our assertions are external to the component being validated.

In contrast, the focus of [29] was performance-debugging systems with as little application-specific knowledge as possible by viewing the system as a black box and examining bottlenecks. Pinpoint [32] and Magpie [30] attempt to infer correct system behaviors from actual executions, without application-specific knowledge. They automatically construct normal behavior through statistical and probabilistic approaches. Magpie describes an architecture for modelling component interactions using system traces. Pinpoint uses a probabilistic context free grammar learned during a training phase to detect deviant behavior, while Magpie constructs stochastic workload models of request-response paths which may then be used for performance prediction, tuning and diagnosis. Our work differs from these efforts in that we ask service engineers to explicitly declare correct behavior, which is critical when no previous samples exist—a common problem in the face of operator actions. The operator task is a first-class
concept in model-based validation and our assertion language, allowing us to relate mistakes to specific tasks.

Along the lines of software testing, a recent related work is ConfErr [38], a tool designed to improve software resiliency to human-induced configuration errors. Our work is different in that we seek to detect a much broader set of mistakes, before they are exposed to the rest of the service and end users. Most other works on software testing are orthogonal to our work.

Finally, another related effort is Araujo and Vieira’s work on models of best practices for DBMS configuration relating to security [39]. This work is complementary to ours in that the tests derived from these models could be used in a model-based validation system to check for security-related configuration mistakes.

There has been other work examining operators and their role in distributed systems. Mirage [40] proposes a possible solution to errors the specific operator task of upgrading deployments. AutoBash [23] provides a set of tools to assist operators in finding and fixing misconfigurations and finding possible solutions. As such, it is complementary to our work here.

Service Level Agreements (SLAs) are another method of specifying required performance, e.g. [41]. For example, [42] includes a runtime monitor, detects SLA violations, and takes steps to enforce the SLAs. However, these works are not meant to pinpoint anomalous behavior or detect operator mistakes.

Finally, another related idea is the analysis of network security from the specification of both vulnerabilities and security policies. MulVAL [43] is a framework that uses Datalog (a subset of Prolog) to specify the security model of the components to be analyzed. As a result of the analysis, MulVAL highlights all violations. Whereas MulVAL is specialized in security analysis, our approach is more generic: the A assertions are applicable to all aspects of the system configuration (including security), as well as to dynamic system properties.
2.3 Machine Learning

Machine learning has been used in a variety of ways to increase the reliability of computer systems. Some of these works can be categorized as system and performance debugging, and others as security systems. With the scale of many Internet systems being as large as they are, and producing as much data as they do, the opportunities to use machine learning tools are great.

Xu et al. [44] recognize that with the wealth of data that many of these systems produce, the ability to find things that go wrong in systems becomes an increasingly difficult chore. They mine the free-text logs of data center services to identify operational problems, and propose ways of organizing the information for operators to understand them. The authors utilize source code analysis to derive structure for log messages and thus create a quality feature set to feed to machine learners. Similarly, Yamanishi and Maruyama [45] present a method of combining mixtures of Hidden Markov Models applied to log files and that are dynamically learned. The logs used were those to determine network failures.

To help system designers with their design choices, systems like Yale [46] allow for rich visualizations and rapid prototyping. In some ways ADIVA can be used to augment knowledge discovery processes. With each iteration of the refinement loop, more useful and specific attributes are found that can help system designers create rules defining correct behavior.

Host based intrusion detection systems monitor system state and can raise alerts when certain system characteristics meet conditions based on preconfigured notions of bad behavior (signature based) or meet some criteria for deviating from the norm (anomaly based). Lee and Stolfo [47] examine system call patterns in the context of Intrusion Detection Systems (IDS). They use RIPPER to classify system calls made by normal executions of known programs versus those made by abnormal executions (those with intrusions). Recognizing that even the best IDSs may miss intrusions, [48], like ADIVA, automatically generates traces to find attack sequences. The domain of their attacks and data sets, namely TCP packets, allows for a complete coverage in the
attacks generated. In other words, they are able to generate all possible attacks derived from a single attack sequence. Command traces, such as the ones proposed in ADIVA lack this rigid structure, and coverage suffers accordingly. We, however, attempt to mitigate this by use of our refinement loop (Section 5.6). We find a similar feedback mechanism used to probe IDS robustness in [49]. This work tests IDSs as [48] does, but generates the variations of attack sequences on a higher level than the network protocols. However, the results of these tests are used to identify weaknesses in the IDS, rather than used to automatically strengthen the IDS’s detection rules.

[50] uses program models to detect attack sequences within the execution of said programs. Much like ADIVA, this work uses state information in an OS abstraction model coupled with a model of program execution to make this determination. However, like the studies above and unlike ADIVA, it focuses on system call and attacks on singular running programs. ADIVA takes a more holistic approach and focuses on the entirety of a task to determine if the system is behaving correctly.
Chapter 3
The Model Based Validation Framework

3.1 The A Assertion Language

3.2 The A Assertion Language

This chapter describes the A language. We first give an overview. We then describe an example system, a common configuration mistake and an example A program that catches mistakes for that system. Finally, we give a brief overview of the major A language constructs.

3.2.1 Overview

In designing A as a domain specific language, we envisioned creating a tool to help distributed system designers reason about correct behavior of their systems, and codify these beliefs. We wished to make easier the task of providing them with access to describe correct conditions using their systems’ monitored state.

To that end, the A gives system designers expressive power through the use of elements, assertions, and tasks. These three abstractions allow the programmer to (1) describe correctness, (2) access system state, and (3) codify their ideas about correct state before, after, and during operator actions.

Elements in A were modeled after aggregate types, such as the C struct. They represent the core data type in the language. It is the interface through which programmers are able to reference actual components in their systems. An element describes the current system state of components such as load balancers and databases. Once an A program is compiled and run, the elements are bound to their physical counterparts by a runtime infrastructure. This runtime also instantiates the elements with currently
monitored values that are defined as properties. For example, the number of jobs in the run-queue of the OS on a particular machine might be a field of an element. A special `stat` variable type captures the notion of statistically sampled values, such as load average, for our class of system devices. In addition, because we are concerned with operator mistakes, elements can also describe the configuration state of devices; e.g., values in a configuration file.

System designers can then use elements in the creation of assertions. The `assert` construct was directly inspired by its use in procedural languages such as C and Java. Anecdotally, maintaining a large set of assertions about correct conditions has been critical toward writing correct programs. Indeed, [51], devotes an entire chapter to the subject of how craft assertions for C programs. Some of the key ideas from that work that cross-over into the domain of correctness for the systems we describe are that assertions should (1) be side-effect free, (2) exhibit fail-fast behavior, and (3) should be explicitly labeled to the high-level conditions they are testing.

Groups of assertions can then be organized together to represent sequential execution of an operator’s actions. The language-level construct allows for the programmer to specify which assertions should be checked, and provides interval points to wait for operators to complete actions. Tasks also allow the `A` programmer to save state in variables to compare past state with present state. The task abstraction thus directly addresses how to describe how system state should change with time, which is often left unresolved in many modeling languages.

Execution in `A` follows a discrete event model, similar in spirit to triggered databases or user-interface libraries (e.g., the Tk widget library [52]). The run-time takes monitored and measured values from real devices as well as input from human operators and forwards them to a central location. Assertions checks and wait statements in tasks map to a scheduled event stream executed at the central location. Operators are then notified when assertions fail.
3.2.2 Example System, Mistake, and Program

In this section, we briefly introduce an example system, the Linux Virtual Server (LVS), a common mistake made configuring LVS, and an A program designed to catch such mistakes.

One of the main applications of LVS is an advanced load balancing solution. A commonly seen LVS operator misconfiguration [53] occurs when LVS is set up in the direct routing mode with Web servers on the back-end. A popular method to achieve direct routing involves assigning the same IP address that the LVS uses to receive requests from the clients to a virtual interface of a loopback device on each Web server. This causes requests to be handled by the LVS machine, with responses handled directly by the individual Web servers. The caveat is that Web servers must ignore ARP requests for the loopback devices. Clearly, the effect of a misconfiguration here will lead to all Web servers and the load balancer answering ARP requests for the shared IP address, leading to a race condition. The manifestation of this misconfiguration is that some client requests might be sent directly to the Web servers, while others go through the load balancer, resulting in an unbalanced load on the Web servers.

Figure C shows a small A program designed to catch mistakes when manipulating an LVS server. At a high-level, lines 1-19 define elements and their properties. Lines 22-48 define the configuration files and properties that are later bound to the instantiated LVS and Web server (Apache) elements at Lines 51-52. These variables at runtime, will be bound to running servers, and lines 55-76 are assertions to describe correct behavior. We give a bottom-up description of these language constructs in the remainder of this section.

3.2.3 Elements

Elements represent states of running service components as reported by runtime monitors. Each element must be declared to be of some element type and consists of a number of fields. Each field can hold a value of a primitive data type, a statistical object, or another element. Rather than being programmer specified, these values within
elements are defined by the monitored state of the component. Elements are bound to specific components. In the listing, the element \textit{lbBalancer} represents a single load balancer bound to a load balancer that has the address \textit{domain.tld}. Similarly, \textit{ws.all} represents all Web servers, regardless of IP address. Mapping the state of the real system to values in the \textit{A} program is the job of the runtime system. Being able to map and refer to groups of machines is one of many features that makes \textit{A} ideal for distributed systems.

```
1 element CPU {
  2    stat utilization;
  3    stat idleness;
  4 }
  5
  6 element CPUGroup {
  7    stat utilization;
  8    stat idleness;
  9 }
 10
  11 element LoadBalancer{
  12    (IP address);
  13 }
 14
  15 element WebServerGroup{
  16    (IP address);
  17    CPUGroup cpu;
  18 }
 19
  20
  21 config lvs{
  22    : lvsadm: "ipvsadm.pl"
  23      single wname = /root/workers/worker/name,"";
  24      single port = /root/server/port,"";
  25  26  27 }
 28
  29 config WS_Apache{
  30    : httpdconf : "httpdconf.pl"
  31      single docroot = /root/DocumentRoot,"";
  32      single port = /root/Port,"";
  33      single jkfile = /root/JkWorkersFile,"";
  34      single jklog = /root/JkLogFile,"";
  35      set jkmount = /root/JkMount,"";
  36    : workerprops: "workersprop.pl"
  37      set wlist = /root/worker-list,"";
  38      set type = /root/workers/worker/[type!="lb"]/type ,"";
  39      set port = /root/workers/worker/port,"";
  40      single lbfactor = /root/workers/worker/lbfactor,"";
  41      set balanced_workers = /root/workers/worker/ balanced_workers,,"";
  42      set host = /root/workers/worker/host,"";
```
Primitive types. A supports a standard set of primitive types including int, double, and string with the typical operators.

Stat type. A stat object represents the tail of a stream of temporally sampled values. The CPU utilization of the Web servers over time is a good example of a property that exhibits this temporal behavior. There are many such monitored quantities where the exact value at an arbitrary time (instantaneously sampled) is insufficient to capture a system’s behavior.

A stat object can be aggregated statistically into a single value via operators such as average, median, standard deviation, min, and max. Stat objects can also be compared using the standard relational operators. A statement like “All web server CPU
utilization should be less than $X$" is thus simple to write in $A$. Statistical equality is calculated by using the Box and Jenkins [54] outlier model.

**Configuration and log types.** Systems like internet services have configuration parameters that determine their runtime behaviors. Mis-configurations are also part of a large class of operator mistakes. Each element may have an attachment of one or more objects of the configuration or log types. Each attached configuration object refers to a set of static information about the service component bound to the element. The definition of a configuration type involves the specification of a set of files or program outputs (Lines 22-50) that can be parsed to obtain the desired static characteristics, a set of drivers that can be used to translate the configuration files into the XML format, and a set of XPath queries to extract the desired characteristics.

In the same vein, definitions of log types, a log object is a stream of information being written to any file.

**Configuration Aggregates.** While it is helpful to express individual configuration parameters, distributed components often share settings. It is, therefore, useful to name an aggregate set of configuration parameters. This construct allows the programmer to easily write assertions regarding configuration parameter of entire tiers of service. For our example, the weight of each of the workers seen by LVS is referred to as a set (Line 41). A programmer can later refer to all weights using this one parameter to make comparisons.

### 3.2.4 Assertions

Much like the `assert` directives in C and Java, an assertion is a boolean expression. In $A$, assertions make statements about the values in elements. An assertion evaluating to true models correctness, while one evaluating to false models incorrect behavior.

Each assertion is comprised of four parts: a name, a conditional expression, a control block, and a set of action statements to be executed if the assertion fails. An assertion is thus like a stylized object in that it contains assertion specific parameters (e.g. frequency to be evaluated), as well as implements a single method that returns a boolean value.

**Name.** The name of the assertion acts as its identifier. We defer to common
programming conventions for the format of these identifiers and advise that appropriate names be used. Programmers can name assertions for what types of checks they are performing and/or what components are involved. This can provide readability and maintainability of A programs.

**Expression.** An expression returns a boolean value. If the statement evaluates to true, the system is behaving correctly with respect to its respective assertion. If the expression evaluates to false, the code in the action block (the else clause) is evaluated. In the listing, Line 71 shows an assertion that models the balanced nature of all Web servers by specifying that the load on each web server be equivalent.

**Control Block.** The programmer can optionally specify values to control the execution of assertions: frequency, delay (when to start an assertion), status (on or off), and type (global or task related) (Lines 56-57).

**Action Block.** The action block (Line 58-51) refers to what the runtime system should do in the event that the particular assertion has failed. We leave actuation, i.e. adjusting the system in response to failed assertions, as future work.

**Hierarchical Assertions.** The A language provides support for abstracting assertions, which can be used to build assertion hierarchies. This allows programmers to think about correct behavior in terms of high-level concepts that eventually resolve to low-level, specific parameter/value checking. For example, the idea that two components are “connected” is a high-level idea that requires more specific checks. This is achieved through assertion composition: the ability for one assertion to call upon one or more other assertions.

**Aggregates.** Relations in the expression part of an assertion can accept aggregate element types as arguments. As mentioned above in Configuration Aggregates, it is helpful to refer to an entire tier of service as a single entity. Very often all nodes of one tier (replicas) should behave similarly. The A language allows for specification of all nodes of a tier to be referred to as one identifier - as shown on Line 51 with ws.all. The A language also provides support for functions to reason about these aggregate elements. For example, the EQUAL operator can take an aggregate element (e.g., a set of replicas). The EQUAL operation applied to a stat field of an aggregate element returns
true if all stat objects in the field are equal, using the definition of statistical types. The \textit{balanced} assertion shows this usage.

\begin{verbatim}
library connected{
  param tier_2 with config TomcatCfg with log TomcatLog;
  param tier_3 with config MySQLCfg;

  assert tier2_to_tier3( numconn \&\& packetnum \&\& ...){
    freq = 10;
    delay = 30;
    on;
    taskonly;
  } else { Vivo.dialog("Tier2 disconnectd from Tier3!"); };

  assert numconn(SUM(tier_2..config.env-entry-value) <=
  (int)tier_3..config.max_connections){
    on;
    taskonly;
  } else{ //Do something }; 

  assert packetnum( tier_3..net.packets_in >= 0.20 * tier_2..net.
  packets_out ){
    on;
    taskonly;
  } else{ //Do something else }
}
\end{verbatim}

\textbf{Listing 3.2: Library Definitions}

\subsection{Libraries}

Libraries support abstraction over different element instances as well as hide detailed sets of assertions. One can think of libraries as “templates”. Libraries can be organized by components, subcomponents, or by goal. Libraries have the added benefit of allowing code reuse via parameterized assertions.

Our LVS assertions could be logically classified in many ways. The \textit{balanced} assertion could be part of a library of assertions on dynamic properties or could be part of a library filled with other properties that need to be “balanced” (eg. memory usage, network usage, etc.). Similarly the \textit{dest} assertion could be part of a configuration library or could be part of a library indicating connectivity of components. The assertions may also be found in libraries describing their component parts (e.g. an LVS library and a
Web server library). The A programmer is free to decide how to organize the models in the most suitable and easily manageable way.

```c
#include "connected.lib";

class add_app_server { name="Add an App Server"; }
{
    // Define an element to represent the DB server
    // running on dbserver.domain.tld in validation slice
    validation db::DBServer(IP="dbserver.domain.tld")
        with config} mySQLCfg;
        with log} mySQLLog;

    // Define an element group to represent all
    // app servers running in the online slice
    online as_all::ApplicationServerGroup(IP=".*")
        with config} TomcatCfg("/path/to/web.xml")
        with log} TomcatLog;

    // Wait for operator to start the task
    wait ("Begin task") { timeout = 30000; }
        else { break; }...

    // Wait for the operator to complete the task
    wait ("Begin Validation") { timeout = 30000; }
        else { break; }

    // Use the library 'connected' to validate
    // that all app servers are connected to the
    // DB server
    use connected with [as_all, db] as add_AS_connected;

    // Check that the App servers are load balanced
    assert balanced ( EQUAL}(as_all.cpu.util))
        ( EQUAL}( COLLECT}(as_all.cpu.util));
            on;
            else {// print "App servers load unbalanced"};
            ...

    assert notOverload (as_all.cpu.util < 0.80){
        on;
        else {// print "An app server is overloaded"};
    wait("End Validation") { timeout } = 30000; }
        else { break; }
}
```

Listing 3.3: Sample task-based A Program
3.2.6 Tasks

Tasks represent human-system interactions. One can think of tasks as sets of assertions grouped together with intervening \texttt{wait} statements. Tasks, unlike assertions, are stateful and can compare system properties throughout the course of task execution. This is particularly useful when checking “before” and “after” values of assertions with dynamic properties.

Our LVS assertions would be included in a task - perhaps one involving setting up a new Web server. For the sake of brevity, the task itself, has been omitted from the listing. Tasks have a few constructs specific to them:

- **Wait statements.** These tell the runtime system to wait on particular operator events, explicitly indicated by the operator.

- **Conditional waits.** The runtime waits on a specific condition within the system, e.g. an element field to reach a certain value.

- **External waits.** The runtime system waits for operator input. This is especially useful when some assertions depend on information only known to the operator. An example of this is the hostname of component currently being modified.

- **Call statements.** These explicitly call for other non-task specific assertions to be evaluated immediately.

- **Variable assignment.** A variable stores the current element field for comparison to future values.

Wait statements allow system engineers a way to segregate operator actions into subspaces - each with its own set of assertions. Wait statements have a timeout (specified in milliseconds), and will cause the task to run the else block if the action is not completed by the time allocated. A conditional wait keeps the task from reaching the next block of assertions until an expression evaluating element values becomes true (or it times out).
3.3 Model

3.4 Example Models

In this section, we present three example models that we later use as descriptions of correct behavior in our experiments: a data flow model, an access control model, and a hierarchical component model. The models provide high-level abstractions that can guide the A programmer in designing the proper assertions. Furthermore, they allow us to reason about the correctness of operator actions, i.e. how the actions affect the key behaviors of a service.

To reinforce some of the points made in the previous section, note that, in contrast with formal models that are used to develop systems, our modeling can be done before, during, and after system development: we create models that describe correct behavior. Moreover, our modeling does not require complete models of all possible system states and behaviors; it simply requires enough completeness to validate operator actions.

Next, we overview each of the models. Although we describe only three models, our approach is flexible enough that other kinds of models can also be abstracted and realized using A.

3.4.1 Access Control Model

Our model of access control uses an classical access control matrix. Each cell of the matrix specifies the access rights a user or a component should have to a particular resource. Almost all components of any distributed system have some level of access control, whether it is as simple as an end-user logging in to a service, or a service that uses functionality from another service.

We found this simple, well-known model quite useful in detecting mistakes that induced security problems. First, the notion of “user” and “resource” are easy to identify in an Internet service. Second, the matrix is easy to realize in an A program. Finally, many security-related mistakes fit within the scope of this model.
Figure 3.1: A partial data flow graph for a cluster of Web servers with 1 front-end load balancer and 1 back-end Web server; an entire graph would extend this basic architecture across the set of back-end servers.

3.4.2 Hierarchical Component Model

This model captures the service component to sub-component relationships. In the model, the components of a service are represented as a directed acyclic graph (DAG). Components (software processes or hardware) are nodes, and a directed edge from one component to another means that there is a dependency between the nodes, where the failure in the first implies failure in the second.

A hierarchical component model can also help to organize assertions and further assist system operators with pinpointing failures.

3.4.3 Data Flow Model

Data flow models characterize a service as a graph in which nodes are computations and edges are message flows. Figure 3.1 shows a data flow graph for a service comprised of a front-end load balancer and a group of back-end servers. The ovals represent request queues for each component, including memory caches and disks. The edges represent the flow of messages (requests and replies) between queues.

Figure 3.1 illustrates the following example assertion: if the input rate to the front-end device has mean $\lambda$, then the flow into each of the $N$ back-end nodes should follow a distribution with mean $\frac{\lambda}{N}$, and the flow out from each back end should have a mean of $\frac{\lambda}{N}$. Of course, in a real service, this assertion would need to be approximated by sampling over an appropriate interval. During run time, the $A$ runtime system can
detect violations of the assertion.

Obviously, describing an Internet service as a flow of messages is not a new concept. For example, see [32, 34, 55]. However, the application of flow models for validating operator actions is novel. In fact, the inspiration for our flow model came from our previous study of operator behavior [10], which suggests that this model is a natural model for detecting a wide range of mistakes.

We found this model to be the easiest to realize in practice because the connectivity between components is well defined for our example multi-tier Internet service.

Most Internet services are designed to handle streams of relatively small requests. Thus, a flow model characterizing services as graphs, where nodes are computation centers and edges are request flows from one node to another, naturally lends itself to defining correctness assertions for these services. In particular, we found that a flow model served very well for considering correctness conditions for both the Ask.com crawler and the auction service.

Figure 3.2 shows a high-level flow graph for the Ask.com crawler. Each tier is a replicated set of software components. Each work unit comprises a set of URLs to be downloaded. Work units traverse statically configured flow paths shown by the arrows. Tier 1 provides coarse-grained work scheduling; Tiers 2 and 3 provide host resolution and fine-grained work scheduling; and Tiers 4 and 5 are used to download and process Web pages. Tier 1 components are both sources and sinks, with work units flowing in the direction of the arrows, and acks flowing in the reverse direction.

Describing an Internet service as a flow of requests is not a new concept, e.g., [32, 34]. However, a flow model immediately brings to mind a number of correctness characteristics that should be validated. First, it dictates that the service engineer must
reason about connectivity, i.e., how components are connected to one another and how requests should flow through the system. We believe that checking system connectivity should be quite useful, since many mistakes observed in our previous study [10] led to incorrect connectivity.

A second characteristic is flow preservation. Requests can only enter the flow graph at known sources and then leave the system at known sinks; they cannot be generated spuriously inside the graph, nor can they disappear without leaving the system through known sinks. Of course, split points (points where a request may transform into several requests) and merge points must properly be defined as sources and sinks. For example, in Figure 3.2, a split point precedes the flow from Tier 2 to Tier 3, and the flow from Tier 3 back to Tier 2 ends at a merge point. Preservation also applies to each node and edge in the system. Over time, flows into a node/edge must equal flows out of it. Otherwise, there is stagnation at the node/edge, indicating either infinite queue build up or lost requests inside that node/edge.

A third characteristic is the expected composition of the work flow. The overall work flow typically comprises a number of sub-flows of different types of request and completion status. Mistakes can often change the normal flow composition.

A fourth characteristic is that flows out of a node into a set of similar (replicated) downstream nodes often should be balanced. This represents the standard load balancing that designers of most Internet services strive for to maintain stable, high system throughput.

A fifth characteristic is that each node and edge typically has a capacity limit which cannot or should not be exceeded. Such limits are sometimes set by configuration parameters. For example, in our multi-tier auction service, the number of threads available in the database server for handling client requests is a configurable parameter. When an operator adds application servers to the second tier, she needs to consider whether the number of threads in the database server needs to be adjusted.

Finally, one should check for stagnation, which corresponds to when a sub-stream of requests is passing through some subset of service components too slowly (or not at all). Stagnation can be caused by mistakes that lead to starvation of some requests or
deadlock.

3.5 Runtime

In order to support the operator centric features of the A language and the model based validation strategy described in Chapter 2.1, we needed monitoring infrastructure. While specific details to our case studies appear in Section 4, we will describe some of the general concepts that apply to any monitoring system used to feed model based validation.

Component Membership - The binding function for the A allows designers to easily make assertions about component types. Therefore, any supporting infrastructure should have facilities to add and remove members from any component type. This could be as simple as a static mapping of network addresses to component type, or could involve a dynamic membership protocol that automatically assigns membership.

State Monitoring - Besides membership, the basis of the assertions described above depend on accurate and real time monitoring of the various attributes of the various components in the distributed systems. These can be constantly changing attributes as general CPU utilization, memory usage, network traffic, and component connectivity as well as domain specific attributes. The number, frequency and type of attributes obviously vary depending on the system and fidelity required. Frequency of monitoring is also dependent on how tolerant a system is to failures, as it governs how quickly deviations from correct behavior are detected.

Configuration Monitoring - Providing dynamic state monitoring is merely a start. To fully support A’s log and configuration assertions, the monitoring infrastructure needs to be able to collect and parse configuration files.

Task Delineation - In the operator centric environment of model based validation, the concepts of task initializations and completion need to be implemented or accounted for in the runtime system. A simple user interface that allows operators to signal when tasks begin and end may be sufficient.
**Assertion Scheduling** - Lastly, any runtime infrastructure that is to support model based validation as described above needs to have a harness to schedule, run, and evaluate assertions. As in state monitoring, the frequency, type and number of assertions that are checked are system dependent, as well as dependent on fault tolerance. The more frequently assertions are checked, the faster operator errors can be detected.

### 3.6 Summary

Model Based Validation, unlike the validation strategies described in Section 2.1 does not require an existing instance of correctly behaving components in order to detect operator error. This is particularly useful in situations when there are non replicated components or when maintenance tasks change the behavior of a system.

We have presented a set of tools and infrastructure to apply the concept of model based validation to distributed systems. Specifically, we have implemented a domain specific language $A$ to assist system engineers to express their beliefs of correctly behaving systems. We also described different types of models that can help guide system engineers in thinking about their systems. The infrastructure implemented monitors and binds data to these codified models, and ultimately helps to determine if operators successfully performed their tasks.

We described the runtime system in two distinct types Internet services representing very different design points: a three tiered auction service and a multi-stage Web crawler at a commercial search engine. In the next Chapter we will evaluate the efficacy of model based validation in each of these scenarios.
Chapter 4
Case Studies and Evaluation

In this section, we describe our validation prototypes for the crawler and the auction service, and evaluate their effectiveness in detecting system misbehavior.

4.1 Ask.com Crawler

4.1.1 Crawler and Validation System

The Ask.com crawler comprises 5 different tiers, where each tier has hundreds to thousands of replicated components. All communication is inter-tier and follows either a one-to-one or all-to-all pattern (see Figure 3.1). Inter-tier messages are carried by distributed persistent communication channels similar to the Unix named pipe, except that each channel supports an arbitrary number of distributed senders and receivers. There are 6 element types that correspond to the components of the 5 tiers and a communication channel. Each crawler component has an average of 20 time series attributes and each channel has 1 time series attribute, giving the Integrator the task of regularly sampling over 20,000 time series.

The validation environment is somewhat different than the ideal environment described in Section 2.1. In particular, we do not have the capability of dynamically creating a validation slice. Rather, the system can be configured to crawl an internal testbed that emulates the Web for validation. Changing from the validation configuration to one that crawls the real Internet requires copying the tier 4 software to a different set of machines and, typically, changing a small number of configuration parameters for tier 1 to increase the crawling rate. The first change is entirely script-based so that direct human action is quite limited.
The testbed that emulates the Web comprises a cluster running a large number of Web servers. The URLs and content served by these servers are realistic since they derive from traces of actual crawls. On the other hand, the emulation has two limitations. First, the parallelism achievable against the testbed is much smaller than that against the real Internet. Second, the Web servers in the testbed could not emulate many of the error conditions that can and do occur on the Internet.

We defined a flow model for the crawler. Each replica in each tier contributed a node to the model. Each one-to-one channel contributed a directed edge to the model. Each all-to-all channel contributed \( n \times m \) directed edges when there are \( n \) source and \( m \) destination nodes. As explained in Figure 3.1, work units comprise sets of URLs to be downloaded and traverse the edges along their directions. Each tier 1 node is both a source and sink. The split and merge points in tier 2 were also identified as sources and sinks. Split points represent the receipt of batches of URL requests, while merge points represent the consolidation of responses submitted to the next tier. Using this model, we wrote one A program to validate 3 major tasks: installing a new instance of the crawler, adjusting the number of hosts and/or replicas, and upgrading the crawler software.

Our validation program contains 27 assertions: 15 of the assertions check for flow preservation across the nodes and edges of the flow model, 5 check for load balancing across the nodes in each tier, and 7 check for the proper composition of the work stream as it flows through the pipeline - e.g., the percentage of successful page downloads vs. the number of error replies received from the testbed/Internet. All assertions involve one or more aggregate elements, emphasizing the importance of dynamic group binding to scaling model-based validation to large systems.

Given the above infrastructure, validation takes place as follows. The operator configures the crawler to run against the internal testbed and starts a validation run with a significant and realistic workload (tens to hundreds of millions URLs to be downloaded). Validation may last between several hours to several days, depending on the extent of the changes being validated and include all components of the crawler. If validation succeeds, the operator changes the crawling rate if necessary and runs a
script to reconfigure the crawler to run against the Internet. The results obtained using the validation run are then discarded or archived for further analysis.

4.1.2 Ask.com Runtime

The A runtime system is hosted by a component called the Integrator as shown in Figure 4.2. The Integrator maintains a database of active service components, called the Component Database, against which A elements can be dynamically bound. In the Ask.com system, all service components communicate with a distributed directory proxy that maintains dynamic membership information and are linked to a remote control and monitoring package called CAPP. The Component Database is maintained by periodically obtaining the set of active service components from the directory proxy. The components’ attributes are obtained either from the directory proxy or by contacting the components themselves through CAPP.

The Integrator also periodically contacts each active service component through the CAPP interface to monitor time series attributes, such as the number of requests serviced and CPU utilization. This monitoring data is maintained in the Stat Database.

We implemented a simple hierarchical element type system with inheritance to support the mapping of component state information to A typed elements. Our type tree is rooted at the type GenericServiceComponent, which is designed to represent a generic service component. It contains four stat type fields (CPU utilization, memory utilization, number of messages sent, and number of messages received) and a config element to hold configuration parameters.

We then implemented 6 element types to represent the 6 types of components in the
Ask.com crawler, 5 of which extend the `GenericServiceComponent` type with fields specific to the software components of the 5 tiers, e.g., the `WebServer` type includes a field for the total number of requests that an instance has received to date. The 6th type represents the communication channels and has just 1 stat field: the number of messages present.

When an A element is instantiated, the runtime system creates an element of the appropriate element type and binds it to the active component from the Component Database whose attributes match the binding expression. Non-stat typed fields are set to the matching attributes, whereas stat typed fields are mapped to the corresponding time series attributes in the Stat Database.

For an aggregate binding, the runtime system creates as many elements as there are service components that match the binding expression, and binds each element to one of the matching components.

Each host in the Ask.com system also runs a host-level daemon that supports remote access to configuration information and log files for service components running on that host. At bind time, the Integrator queries these daemons to gather configuration parameters and setup log forwarding for A elements whose types are defined to contain configuration parameters and log outputs.

Finally, the Integrator runs an A program by instantiating and/or executing bindings and assertions in program listing order. Once started, each assertion is periodically scheduled according to its specified frequency. Scheduling is implemented via a sorted ready queue, with assertions sorted according to the next time they should be executed.

### 4.1.3 Effectiveness

Our work at Ask.com was performed as part of an effort to introduce a new set of technologies into their crawler. This was a great time for evaluating validation as many changes were made to the crawler and the new software system was deployed in a number of stages; we estimate that there were about 15 validation runs during our study. This is an estimate because our validation system was not ready for real use at the time. We view the very last test run before entering production mode as a
validation run. During these runs, there were a handful of mistakes and/or software problems that were easily identifiable by the normal monitoring system. There were also 6 instances (4 due to operator mistakes and 2 due to software/design bugs) that were not detected until after considerable post-run analysis.\(^1\) (Much of this analysis was motivated by our on-going work on model-based validation. In essence, we were applying model-based validation by hand.) We use logs from these 6 cases as a first step in evaluating model-based validation. In particular, we executed the A program described above on a live Integrator and replayed the logs from the 6 cases to feed the Integrator with monitoring data. We consider the two cases that were not operator mistakes because similar misbehavior could have been caused by mistakes (and because it is interesting to see if model-based validation can detect such misbehavior).

**Mistake 1.** Early on in our development, inter-tier communication through the persistent channels did not have timeouts. If the crawler is shutdown for any reason, the operator was responsible for clearing the channels before restarting the crawler. On one occasion, the crawler was shutdown for a software upgrade, and, when it was restarted, the operator forgot to clear the channels. To exacerbate the problem, the shutdown was done in such a way as to reset the channels to include a large amount of already processed requests. This caused the crawler to process a large amount of work from the previous run. Further, part of this work stream was not properly rate controlled because it entered the pipeline at an unexpected entry point. This led to a “politeness violation,” where the crawler downloaded from a set of Web servers at rates that exceeded the normal self-imposed rate designed to not overload Web servers with crawling traffic. Fortunately, this mistake was detected during the validation run (although it was not detected until a post-analysis of the validation run was performed). This problem led to the addition of application-level timeouts to messages passed via channels. Model-based validation quickly detected this mistake as it triggered the assertions checking flow preservation for flows into and out of the channels.

\(^1\)One mistake was detected quickly but finding the root cause took longer.
Mistake 2. Also early in our development, the parameter for the capacity limit of a channel was lost if the server managing it crashed. When the server recovered, it would set the channel’s limit to a small number under the assumption that the operator would quickly notice the bottleneck and correct it. Once, during a software upgrade of some tiers, the channels were left running. During the upgrade, the channel server crashed (the reason for the crash was unrelated to the upgrade) and automatically recovered, causing all channel capacity limits to reset. When the crawler was restarted, the operator forgot to check the channels’ capacities. This caused the system throughput to be very low. Model-based validation quickly detected this mistake via a simple threshold check assertion. More interestingly, when the problem was detected, the development team thought it was a software problem related to the upgrade. The real problem was not found until a careful investigation of the software failed to identify a bug. Our current validation program would not have accelerated the investigation, because it only checks dynamic flow properties. However, had we had a chance to use configuration-checking assertions as we did for the auction service (which we were in the process of developing), they would have helped to quickly pinpoint the root cause.

Mistake 3. In one validation run, an operator mistakenly started a replica in tier 1 from an old installation, when restarting the crawler after installing a new version of it to a different location. This led to an extra component running in tier 1 that initiated work from an old workload. Model-based validation quickly detected this mistake as it triggered assertions checking that work only originate from designated sources.

Mistake 4. In one validation run, about 30% of a newly created input workload (URLs to be downloaded) were mistakenly provided in an old format. This caused the crawler to generate and attempt to crawl incorrect/non-existent URLs. (This may seem like a rather benign mistake but it is not; a crawler attempting to download a large number of non-existent URLs from a site can annoy the site administrator and lead to complaints against or even blocking of the crawler.) This led to a failure rate of roughly 30%, which is much higher than normal. Model-based validation quickly detected this mistake as it triggered one of the assertions checking for the composition of the work flow. Interestingly, a similar mistake was made on another validation run, but only
affected a very small percentage of the input workload. Model-based validation did not detect this because the change in percentage of failed downloads was within the expected variance.

**Software bug.** In one validation run, a software bug (i.e., a programmer rather than an operator mistake) caused a failed HTTP decoding to result in the continuous downloading of the same URL for a portion of the workload. This problem is undesirable because both unnecessary work was done and a politeness violation ensued. Model-based validation detected this mistake as it triggered assertions checking preservation of flow into and out of the buggy component.

**Design change.** Web servers on the Internet can provide instructions to Web crawlers via *robots.txt* files. Thus, a crawler must always download (but may cache) this file to check for appropriate directives before crawling a Web site. In one version of the crawler, the caching policy was modified. In particular, if an HTTP request for a *robots.txt* failed because of a network or protocol error, the result was not cached. This is conservative, since such errors were expected to be transient. However, it turns out that Web servers on the Internet can enter and remain in such error states for much longer than expected. During one validation run (against the Internet), the crawler tried to download a small set of *robots.txt* files over and over again, because each try resulted in a network or protocol error and so was not cached. This was a valid design choice. However, the development team decided to change the choice to minimize the number of *robots.txt* requests to sites that are in a persistent error state. Model-based validation did not catch the fact that there were more requests for *robots.txt* than expected, because the error stream from these troublesome sites was quite small and thus within the expected variance threshold.

### 4.1.4 Performance Overhead

The Integrator runs on two machines, each with an Intel Xeon 2.4 GHz processor and 4 GB of RAM. The data collection part of the Integrator and the Stat Database are split across the two machines. The *A* execution environment runs on just one of the machines and accesses the Stat Database on the second machine using NFS. Currently,
this setup is able to sample each time series attribute once every 5 minutes. (We expect that fairly simple optimizations, such as parallelizing the periodic contacting of active components, would allow us to sample once every minute.) The granularity at which sampling occurs dictates how quickly problems can be detected, as the overhead imposed by assertion checking is negligible.

Our monitoring places negligible overheads on the crawler components, which already maintain extensive logging for offline analysis. Thus, each component only needed to maintain counts of ~20 types of logged events and to answer 1 RPC request every 5 minutes.

4.1.5 Summary.

Model-based validation quickly detected 5 out of the 6 problems described above. We find that the flow model and the corresponding A program are quite effective at detecting mistakes/problems that lead to statistically significant violations of flow principles, such as flow preservation and expected composition. These are the most important mistakes since they can have serious business consequences. On the other hand, because we are currently checking work flows at a coarse-grained level, mistakes that lead to changes in the flow that are within the expected variance thresholds cannot be detected. Fortunately, these “small” mistakes typically do not have significant impact.

Further, some of these mistakes can be caught by assertions on configuration parameters and log outputs, which we did not have time to write for the Ask.com system.

The effectiveness of model-based validation brings an unforeseen advantage: it allows Ask.com to execute validation runs against the Internet in the future with the confidence that any significant mistake/problem would likely be quickly detected. This provides Ask.com with the ultimately realistic validation environment. As already mentioned, model-based validation will also be continuously monitoring the production crawler to help Ask.com be a good Internet citizen.
<table>
<thead>
<tr>
<th>Library</th>
<th>Number of Assertions</th>
<th>Components required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Balancer</td>
<td>4</td>
<td>Load Balancer, Webservers</td>
</tr>
<tr>
<td>Webserver</td>
<td>8</td>
<td>Webserver</td>
</tr>
<tr>
<td>Webserver to Application Server</td>
<td>5</td>
<td>Webservers, Application Server</td>
</tr>
<tr>
<td>Application Server</td>
<td>4</td>
<td>Application Server</td>
</tr>
<tr>
<td>Application Server to Database</td>
<td>5</td>
<td>Application Servers, Database proxy, databases</td>
</tr>
<tr>
<td>Database</td>
<td>7</td>
<td>Database proxy, databases</td>
</tr>
<tr>
<td>Dynamic balanced utilization</td>
<td>2</td>
<td>any component</td>
</tr>
<tr>
<td>Dynamic capacity</td>
<td>2</td>
<td>any component</td>
</tr>
<tr>
<td>LVS policy (balanced)</td>
<td>2</td>
<td>Load Balancer, Webservers</td>
</tr>
<tr>
<td>Apache policy</td>
<td>6</td>
<td>Webservers</td>
</tr>
<tr>
<td>Tomcat policy</td>
<td>1</td>
<td>Application Servers</td>
</tr>
<tr>
<td>mySQL policy</td>
<td>3</td>
<td>Databases</td>
</tr>
</tbody>
</table>

**Flow Assertion Type**

- Content Assertions: 3
- Flow::Performance::Capacity: 8
- Flow::Performance::Connectivity: 30
- Flow::Security::Capacity: 0
- Flow::Security::Connectivity: 4

<table>
<thead>
<tr>
<th>Library</th>
<th>Number of Assertions</th>
<th>Components required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Balancer</td>
<td>4</td>
<td>Load Balancer, Webservers</td>
</tr>
<tr>
<td>Webserver</td>
<td>8</td>
<td>Webserver</td>
</tr>
<tr>
<td>Webserver to Application Server</td>
<td>5</td>
<td>Webservers, Application Server</td>
</tr>
<tr>
<td>Application Server</td>
<td>4</td>
<td>Application Server</td>
</tr>
<tr>
<td>Application Server to Database</td>
<td>5</td>
<td>Application Servers, Database proxy, databases</td>
</tr>
<tr>
<td>Database</td>
<td>7</td>
<td>Database proxy, databases</td>
</tr>
<tr>
<td>Dynamic balanced utilization</td>
<td>2</td>
<td>any component</td>
</tr>
<tr>
<td>Dynamic capacity</td>
<td>2</td>
<td>any component</td>
</tr>
<tr>
<td>LVS policy (balanced)</td>
<td>2</td>
<td>Load Balancer, Webservers</td>
</tr>
<tr>
<td>Apache policy</td>
<td>6</td>
<td>Webservers</td>
</tr>
<tr>
<td>Tomcat policy</td>
<td>1</td>
<td>Application Servers</td>
</tr>
<tr>
<td>mySQL policy</td>
<td>3</td>
<td>Databases</td>
</tr>
</tbody>
</table>

**Flow Assertion Type**

- Content Assertions: 3
- Flow::Performance::Capacity: 8
- Flow::Performance::Connectivity: 30
- Flow::Security::Capacity: 0
- Flow::Security::Connectivity: 4

Table 4.1: A program broken down into types

### 4.2 Auction Service and Validation System

We now turn to evaluating our validation approach using a three tiered system. We first consider the effectiveness of model-based validation using an extensive set of mistake-injection benchmarks. We then assess the performance impact of our validation framework on the live service as well as the scalability of the Integrator.

#### 4.2.1 Experimental Setup

Our evaluation is performed in the context of an online auction service modeled after EBay [13]. The service is organized into 4 tiers of servers: load balancer, Web, application, and database. We use 1 load balancing server running Linux LVS, 2 Web
servers running Apache, 2 application servers running Tomcat, and 1 database server running MySQL. Each node is a blade server with a 1.2 GHz Intel Celeron processor and 512 MB of RAM. All nodes run the Linux kernel 2.4.18-14 and are interconnected by a Gigabit Ethernet switch.

The service requests are received by the Web servers and may flow towards the second and third tiers. The replies flow through the same path in the reverse direction. Each Web server keeps track of the requests it sends to the application servers. Each application server maintains the soft state associated with the client sessions that it is currently serving. This state consists of the auctions of interest to the clients. All dynamic requests belonging to a session need to be processed by the same application server, thereby restricting load balancing. A heartbeat-based membership protocol is used to reconfigure the service when nodes become unavailable or are added to the cluster.

A client emulator is used to exercise the service. The workload consists of a “bidding mix” of requests issued by a number of concurrent clients that repeatedly open sessions with the service. Each client issues a request, receives and parses the reply, “thinks” for a while, and follows a link contained in the reply. A user-defined Markov model determines which link to follow. During our mistake-injection experiments, the overall load imposed on the system is 40 requests/second, which is roughly 50% of the maximum achievable throughput.

### 4.2.2 Auction Service Runtime

In this section we describe our prototype runtime system for executing $A$ programs. The prototype targets standard multi-tier Internet services and is implemented in Java. It consists of two parts: a set of monitors, which capture the state of running service components, and an Integrator, which collects system state in a central location and executes the tasks and assertions. Figure 4.2 shows the overall architecture of our prototype.

Each component of the service may host one or more monitors. Each monitor may observe multiple properties of the hosting component, e.g., CPU utilization and number
of messages sent. Monitors can stream observations to the Integrator and/or respond to specific requests for information.

The Integrator has three main functions: (1) receive and store observations from all monitors; (2) map accesses to each $A$ element to real observations; and (3) schedule and execute assertions from an $A$ program.

Next, we describe the monitors, the storage of monitoring data on the Integrator, the binding of $A$ elements to observations, and assertion scheduling.

### 4.2.3 Monitors

Monitors are software components designed and implemented outside the scope of the $A$ system. However, to communicate with the Integrator, each monitor must obey a naming convention and support appropriate interfaces. In particular, a monitor must have a unique name that captures relevant properties of the observed component.

All monitors are automatically started when their hosting components are started. At startup, each monitor registers itself with the Integrator along with a callback object if it can be queried or remotely controlled. When a registration request arrives at the Integrator, the Integrator inserts the monitor’s name and call-back object into a directory of active monitors. Each monitor’s name is also mapped to a monitor type, which determines the RPC interface supported by the callback object and the number and types of monitoring data streams that the Integrator should expect from that monitor. Using the type, the Integrator creates a monitoring stream object (MSO) to receive and store the appropriate tail of each monitoring data stream.

Currently, our prototype implements a generic server monitor that is used to monitor each dedicated server node (e.g., a Web server) in a multi-tier service. An instance of the monitor is run on each node and monitors the node’s CPU utilization, memory utilization, the number of packets sent and received, the state of the OS, and the configuration parameters and log files of the software server component that the node is hosting. The node’s characteristics and the log files are streamed to the Integrator as separate monitoring data streams. The OS state and configuration parameters are provided through a querying interface.
Figure 4.2: Architecture of the A runtime system. Shaded circles inside the service components (Web server, application servers, database server) represent monitors. Shaded circles in the directory of active monitors represent monitor callback objects. Hexagons inside the MSO Store represent monitoring stream objects (MSOs). The dynamic streaming of monitoring data to the receiving MSOs is only shown for one monitor for clarity.

There are two categories of MSOs, stat and log, which map directly to A stat and log objects.

4.2.4 Configuration Parameters

Our monitor implements a generic querying interface that the Integrator can use to query a range of configuration and status parameters. Recall that when a config element is instantiated in an A program, its properties are populated using a set of XPath queries against a set of configuration files. When binding elements, the Integrator contacts the monitor bound to that element and sends it a query containing a set of tuples. Each tuple contains the name of a configuration file, the name of the driver to be used to convert the configuration file into XML, and a set of field initialization tuples, each of which contains the name of a configuration parameter, its type (single or set), and the XPath needed to retrieve the parameter value from the generated XML file. Our implementation is fairly generic in the sense that a driver may be an arbitrary program that produces XML-formatted output.

Each monitor is responsible for running each driver against the specified configuration file to translate it into XML and then running the XPath queries against the resulting XML file. Results to all queries are then sent back to the Integrator. The
Integrator also refreshes all config elements after a task has been completed.

We designed and implemented the above querying interface instead of having the monitor forward the appropriate configuration files to the Integrator because some configuration parameters are stored as internal state of a service component, rather than in external configuration files. For instance, the MySQL DBMS stores access control information in tables of a special database; therefore, obtaining such information requires interacting with MySQL by means of SQL queries.

Our currently implemented drivers, all of which are Perl scripts, can convert into XML the following: Apache Web server configuration files, Tomcat application server configuration files, MySQL DBMS configuration files, MySQL access control database tables, output of the Linux Virtual Server (LVS) `ipvsadm` administration tool, output of the `iptables` command for packet filtering and NAT, output of the `ifconfig` command to get information about the status of the network interfaces in use, contents of the network-related files of the `/proc` file system, and i-node information of all directory entries of any given file system subtree.

### 4.2.5 Creating and Binding A Elements

Recall that the element types available to an A programmer are defined by the monitoring and runtime system, as opposed to the A programmer. Our prototype currently implements four element types: `LoadBalancer`, `WebServer`, `AppServer`, and `DBServer`. Each element type is designed to represent a particular service component type, i.e., an LVS load balancer, an Apache Web server, a Tomcat application server, and a MySQL DBMS. Each of these four types is built as a wrapper around a generic Server type used to represent our generic monitor. The Server type contains 4 stat MSOs to receive and store the data streams corresponding to CPU utilization, memory utilization, number of packets sent, and number of packets received, a log MSO to receive and store the server’s log messages, and a config object to hold the set of configuration parameters extracted from the server by the A program.

When an A element is instantiated via an A binding statement, the runtime system creates an object of the appropriate type, finds an active monitor whose name matches
the binding expression, and binds the object to the matching monitor, its callback object, and its MSOs.

For a group binding, the runtime system creates as many objects of the appropriate type as there are active monitors that match the binding expression. Each object is bound to a monitor as described above. A group object is also created to hold the set of created element objects.

4.2.6 Assertion Scheduling

Every assertion is assigned an evaluation time. These are then sorted on a ready queue. After evaluation, the assertion’s next evaluation time is calculated and it is placed on the queue.

The assertion scheduler distinguishes between two types of assertions, global assertions and task-specific assertions. Global assertions are inserted into the assertion queue when the $A$ program is loaded, and stay there as long as they are not turned off while the program is running. Task-only assertions, on the other hand, are added into the ready queue during the task to which they belong; they are removed when the validation of the task has been completed, regardless of the validation result.

4.2.7 Models and $A$ Programs

To evaluate the effectiveness of model-based validation, we acted as system engineers for the auction service and defined three models of the service as discussed in Section 3.4.3. Our flow model has three general requirements: (i) each component in a tier should be connected to all components in an adjacent tier; (ii) components should not connect to non-existing or offline components; and (iii) the capacity of each node should be equal to or greater than the sum of the incoming flows. Our hierarchical component model stipulates that (i) the configuration parameters for each component should match a given set of policies and should be internally consistent, (ii) each component type should have a number of observables to provide evidence that a component of that type is running, (iii) no component should be overloaded, and (iv) components within each replicated tier, i.e., Web and application, should exhibit similar resource utilization
over time. Finally, our security model specifies an access control matrix for controlling accesses to the components in the service.

We then used the above models together with results from previous works exploring the nature of operator mistakes [2, 6, 10, 56–59] to guide the writing of 12 A libraries (Table 4.1 shows the number of assertions in each of these libraries). Three libraries were written to express our flow model, one per pair of adjacent tiers. These libraries contain assertions about inter-tier connectivity, such as the destination port configured on a Web server for a TCP connection should be equal to the listening port configured on the application server. They also contain assertions about flow capacity, e.g., the sum of the allowed outstanding JDBC connections on the application servers should be no larger than the number of concurrent requests the database is configured to handle.

Eight libraries were written to express our hierarchical component model. Three of these libraries contain assertions about the correct running of components in the Web, application, and database tiers such as an httpd process must be running on a Web server. They also contain assertions about the consistency of related configuration parameters of each component type; e.g., the set of application servers that a Web server can connect to appears in two places in an Apache configuration file—these two lists should match. Three libraries contain assertions about per-component policies; for instance, the load balancing policy of the LVS balancer. We separated these two sets of libraries because the former should always hold true unless the component is changed fundamentally while the latter may change as policy settings change. Finally, two libraries assert that resource utilization should be balanced across the components of a replicated tier and that no component should be overloaded.

The last library was written to express our security model. For simplicity, we only included the access control matrix for the database server. We also wrote assertions to ensure that the database server has a non-null password for each user with access rights so that the access control matrix cannot be circumvented. All 12 libraries together amount to 749 lines of code.

We wrote four task-specific A programs: (1) adding an upgraded Web server, (2) adding an upgraded application server, (3) adding a load balancer, and (4) adding
a database to the DBMS. We used these programs along with our mistake-injection benchmarks (see Section 4.2.8) to evaluate our approach. Once we wrote the libraries, it was quite easy to write the task-specific programs. All programs together correspond to only 125 lines of code. In general, each task-specific program only consisted of two sets of statements: (1) binding statements identifying the components that are affected by the task and those not affected but needed for validation; and (2) invocation of the libraries to check the correctness properties that might have been impacted by operator mistakes when performing that task. This experience provides strong evidence that, given a good core set of assertions about the properties and behaviors of a correct system, writing task-specific validation programs requires relatively little effort. The simplicity of writing task-specific programs may be important since there are potentially many different operator tasks that should be validated.

4.2.8 Mistake-Injection Experiments

Table 4.2 summarizes our mistake injection experiments. We injected mistakes in the context of a number of maintenance tasks such as adding a service component, upgrading software, and changing the configuration of an existing component. Each experiment consisted of a single mistake injected into the scripted execution of one task. All tasks affecting a specific type of component, e.g., Web server, were validated using one task-specific program; since our tasks were performed on four different types of components, this led to the four task-specific programs mentioned above.
<table>
<thead>
<tr>
<th>Category</th>
<th>Mistake</th>
<th>Impact</th>
<th>Task</th>
<th>Caught?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connectivity</td>
<td>“LVS ARP problem”: Web server not configured to ignore ARP requests. <em>(synthetic but well-known [53])</em></td>
<td>Web server might respond to ARP requests originated by clients, bypassing the front-end load balancer.</td>
<td>Web server addition</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Web server not compiled with support for the membership protocol. <em>(synthetic [2])</em></td>
<td>Affected Web server will forward requests to application servers taken offline.</td>
<td>Web server addition</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>TTL of membership heartbeat messages is misconfigured in one application server. <em>(synthetic [2, 57])</em></td>
<td>If the TTL is too high, the Web servers will not stop sending requests to an application server taken offline.</td>
<td>App. server addition</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Misconfiguration of one pair of Web server and application server: wrong yet matching port numbers. <em>(synthetic [58])</em></td>
<td>All Web servers but the affected one will not be able to contact the misconfigured application server.</td>
<td>Web server addition</td>
<td>Y</td>
</tr>
<tr>
<td>Capacity</td>
<td>The number of connections handled by the database is exceeded. <em>(synthetic [56])</em></td>
<td>It causes the system not to work at the maximum capacity.</td>
<td>App. server addition</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Wrong front-end load balancer policy is activated. <em>(synthetic [59])</em></td>
<td>Undesired/inappropriate distribution of load across Web servers.</td>
<td>Load balancer addition</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Web server load balancer misconfigured. <em>(synthetic [59])</em></td>
<td>Undesired/inappropriate distribution of load across application servers.</td>
<td>Web server addition</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>DBMS performance parameters configured sub-optimally. <em>(survey [6])</em></td>
<td>Service overall performance might be jeopardized.</td>
<td>Database creation</td>
<td>Y</td>
</tr>
<tr>
<td>Security</td>
<td>Database administrator account not assigned a password. <em>(observed [10])</em></td>
<td>Serious security vulnerability.</td>
<td>Database creation</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Allowing any machine to access the database remotely. <em>(survey [6])</em></td>
<td>Security vulnerability and possibility of data corruption due to benign yet unauthorized data access.</td>
<td>Database creation</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Allowing an ordinary user to grant/revoke privileges to/from other users. <em>(survey [6])</em></td>
<td>Serious security vulnerability.</td>
<td>Database creation</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 4.2: Mistake-injection experiments.

We explored three main categories of mistakes: those affecting the connectivity of multiple service components (connectivity), those affecting the capacity of some subset of components (capacity), and those affecting the security of the system (security). One mistake was actually observed during operator experiments performed with volunteer human subjects in a previous study [10] (observed). Some of the mistakes were reported
in a survey of database administrators [6] (survey). All other mistakes were synthetically generated but motivated by previous work exploring the nature of operator mistakes [2, 6,10,56–59] (synthetic). Our injected mistakes share two key characteristics: (1) either they occur frequently or they can impact the system significantly, and (2) it is difficult or impossible to catch them using trace- and/or replica-based validation.

During the mistake-injection experiments, we performed model-based validation as described in Section 2.1 using the programs described in Section 4.2.7. The components in the validation slice were subjected to a load of 20 requests/second. Overall, we considered that model-based validation caught 10 out of the 11 injected mistakes shown in Table 4.2. In all cases, if a mistake was detected, the assertions that fired gave very strong clues for identifying the actual mistake.

In the next paragraphs we present more details for some of the mistakes, their potential impact on the service, and how they were detected (or not detected) by model-based validation.

“LVS ARP problem”. This is a well-known LVS misconfiguration [53] that may occur when LVS is set up in the direct routing mode. In this mode, the LVS machine is supposed to receive all client requests and forward them to the Web servers, but the Web servers are responsible for sending the responses directly to the clients. A popular method to achieve this direct routing involves assigning the same IP address that the LVS uses to receive requests from the clients to a virtual interface of a loopback device on each Web server. The caveat is that Web servers must ignore ARP requests for the loopback devices; otherwise, all Web servers and the load balancer will answer ARP requests for the shared IP address, resulting in a race condition. Consequently, some requests might be sent directly to the Web servers, while others go through the load balancer.

We injected the mistake of allowing a Web server to answer ARP requests for its loopback device, which is the default behavior. In the validation slice were the load balancer, the Web server operated upon, an additional Web server, two application servers and a database proxy. Interestingly, when validating the Web server, we noticed that only the assertion about the configuration of the loopback device failed. The load was
actually correctly distributed across the Web servers behind LVS. The reason was that the load generator had cached the ARP response given by the load balancer. Trace- and replica-based validation would have overlooked this mistake, since everything worked perfectly during validation; the problem was a latent error due to a misconfiguration. In the interest of completeness, we decided to perform another validation run after making sure that the ARP cache of the load generator was cold. In this run, all requests were sent directly to one Web server, bypassing the load balancer. This time not only did the configuration assertion fail, but our assertions that CPU and memory utilization should be uniform across the Web servers also failed.

**Membership protocol mistakes.** In our testbed service, the application servers periodically send heartbeat messages to the Web servers; accordingly, the JK module of the Apache Web servers (the module that extends Apache to communicate with Tomcat) keeps a list of available application servers based on the heartbeats. By doing so, the Web servers are able to stop sending requests to application servers taken off-line for maintenance/validation. In order for this membership protocol to work properly, two conditions must be satisfied: (1) Apache’s JK module must be compiled with support for the protocol; and (2) Tomcat must be configured to multicast heartbeat messages to the appropriate multicast group.

We injected two mistakes pertaining to the membership protocol described above. The first mistake was not compiling the JK module of a new or upgraded Web server with support for that protocol. The affected JK module would rely on a static list of application servers specified in a configuration file instead of building a dynamic membership table.

Replica and trace-based validation cannot deal with the above mistake because it impacts the machines residing in the online slice: the online Web servers would keep trying to reach the application server(s) isolated in the validation slice. During the experiment, the A program we used for model-based validation did not detect this mistake. In order for model-based validation to detect it, we would need to implement a driver to convert the output of the `ipcs` utility, which reports information on SystemV shared-memory segments, semaphores, and message queues, into XML, and write an
assertion to make sure that the number of threads attached to the membership shared-memory segment is greater than 0. However, for the purposes of our evaluation, we deem this mistake as not caught.

The second injected mistake related to the membership protocol was assigning an extremely high value to the TTL parameter carried by the heartbeat messages. This parameter is set in a Tomcat configuration file. Apache’s JK module receives this value via the heartbeat messages and uses it to determine when a certain application server can be regarded as off-line. Assigning a high value to this parameter could be the result of interpreting it as a quantity expressed in milliseconds as opposed to seconds. The impact of such misunderstanding is the same as the one described in the previous paragraph. During model-based validation, our assertion performing a sanity check on the value of such parameter caught the mistake.

Wrong yet matching pair of port numbers. When we proposed trace- and replica-based validation, we also introduced the notion of multi-component validation [10] to validate the interaction between a pair of components — e.g., Web and application servers. Imagine a situation where the Tomcat instance being operated on is incorrectly configured to listen on a port different from the one assumed by Apache’s JK module. Multi-component validation would catch this mistake. In the sequel, upon validation failure, suppose that the operator realizes the problem and, instead of correcting the port number on the Tomcat side, she changes the previously incorrect port number on the Apache side. Now, multi-component validation succeeds and both components are automatically moved from the validation to the online slice. Unfortunately, the other Web servers on the online slice will not be able to forward requests to the application server just validated; similarly, the just-validated Web server will not be able to contact the other application servers.

In contrast, our A program based on our connectivity model detects this mistake through the assertion enforcing a policy stating that all Web servers should be able to communicate with all application servers.

This injected mistake was motivated by a port misconfiguration mistake observed by Maglio et al. [58].
**Exceeded concurrent database connections.** Typically, the DBMSs are configured to handle a limited number of concurrent connections. In particular, one configuration parameter of MySQL specifies the maximum number of concurrent connections to be accepted. For performance reasons, each application server has a connection pool to overlap accesses to the database. If application servers are added, the cumulative demand for concurrent connections to the database may exceed this setting.

This mistake was inspired by a misconfiguration observed in [56].

We injected the mistake of exceeding the number of connections handled by the database. During the experiment, model-based validation caught this mistake by means of an assertion enforcing the policy that the connection pools of all application servers must be fully served by the database.

**Load distribution mistakes.** We injected a load distribution mistake into two load balancers: the front-end LVS and Apache’s JK module. The former distributes load across the Web servers, whereas the latter deals with balancing load to the application servers. It is worth noting that each Apache instance runs its own JK load balancer.

Depending on the desired load distribution policy, a different set of assertions is activated during validation. Assuming the policy of equally distributed load across homogeneous machines, we injected the mistake of configuring the load balancers to give more weight to a particular server in each of the affected tiers.

This mistake-injection experiment on the LVS load balancer caused the assertions on uniform CPU and memory utilization of Web servers to fire. In addition, since we assumed that the Web servers were homogeneous, the assertion stating that their weights must be equal was activated during validation and also fired.

In contrast, injecting this mistake in the JK load balancer of one Web server did not cause the assertions on equal CPU and memory utilization of application servers to fire. The overhead inherent to Tomcat masked the uneven load distribution provided to the application servers because the load used to drive model-based validation (20 requests/s) was relatively low. In this experiment, the only firing assertion was the one stating the equality of the weights assigned to each application server. This assertion was running due to the homogeneity assumption.
**DBMS configured sub-optimally.** According to the results of a survey we conducted with DBAs [6], database performance problems (including the ones induced by DBA mistakes) are a serious adversity faced by them. They find themselves overwhelmed by a plethora of configuration parameters. Inappropriate values for those parameters may lead to poor performance in situations of high load or when certain types of queries are executed. This is therefore a latent problem likely to be discovered only after the performance degradation has been exposed to the users.

Checking the database behavior by actually exercising it requires replication and hard state handling; furthermore, it can be a time consuming task. Our approach of model-based validation can help detect latent performance problems by checking assertions about values of parameters following some guidelines suggested by the DBMS manual or based on DBAs' experience.

**Security mistakes.** In a previous study [10], we observed a severe security mistake during the live operator experiments we conducted with human subjects: while migrating the DBMS, one operator forgot to assign a password for a MySQL account having all DBA privileges. Also, DBAs we interviewed [6] mentioned database problems resulting from security vulnerabilities. The security mistakes shown in Table 4.2 were derived from our previous studies. During the experiments, our access control model caught all security mistakes we injected.

Similarly, although in a different realm, a study on configuration errors of firewalls of many organizations belonging to different market segments, research labs, academic institutions, and network security consulting firms [59] revealed that a significant fraction of the analyzed firewall rules violated well-established industry practices and security guidelines. We believe that model-based validation can help change this state-of-affairs by providing an infrastructure where a model derived from security guidelines can be implemented and checked by means of assertions.

**Final remarks.** Looking back at our mistake-injection experience, it is clear that inspecting which assertions fire provides strong clues in pinpointing the mistakes. In all cases, the assertions that fired were exactly those that related directly to the mistakes. For example, the assertion stating that all Web servers should connect to any application
server using the same port number fired when a pair of Web server and application server was given different, yet matching port numbers. Furthermore, we have observed that no assertions fire in our prototype during normal operation. Thus, by relating any firings to the actions they have performed, operators can easily correct their mistakes.

4.2.9 Performance Overheads

Our centralized approach raises the immediate concern of overload on the Integrator node. However, we found that, in practice, the resource consumption was minimal. We measured the network and CPU consumed by the Integrator and monitors. We stressed the Integrator by running an A program with all 49 assertions we wrote, regardless of task, and had the whole service in the validation slice (1 LVS, 2 Web servers, 2 application servers, and 1 database server). The incoming bandwidth was 8.38 KB/s, i.e. each component sent monitoring information to the Integrator at 1.40 KB/s, which is negligible for Gigabit networks. Moreover, the average CPU utilization of the Integrator was also negligible: only 2.73%. In addition, we investigated the overhead of running the monitors on live service components. The monitors added at most 6% to the average CPU utilization.

These results suggest that our single Integrator node could handle a substantially larger service. In fact, given the statistics listed above, we expect that the Integrator node could handle in excess of 200 server nodes. A more modern node could handle substantially more.

4.3 Application of Model Based Validation: Barricade

Model based validation and the associated A Framework can be used as stand-alone tools to help engineers think about correctness of their systems. Barricade [60] is joint work with Oliveira et al that utilizes Model Based Validation as the basis for a management framework that protects large computer systems against the effects of operator errors. What follows is a brief description of Barricade, a discussion of Model Based Validation’s role in the larger work, and the experimental results of using Model
Based Validation in Barricade. The details of how Barricade works are beyond the scope of this dissertation. A more in depth discussion of Barricade can be found in [60].

4.3.1 Overview

The Barricade Framework is a management tool that defends against operator caused mistakes via a monitoring and proactive management infrastructure. It is designed to aid in replicated distributed systems management. Typically, in these environments, an operator needs to perform the same task on a number of different machines. Barricade allows operators to act upon a subset of these machines, test their actions, then propagate the changes to the remaining machines. This approach of mistakes aware system management actively helps to prevent the spread of the harmful effects of operator error. This is accomplished by erecting barricades or barriers to confine operator actions within a small partition of the system. Changes made by the operator are only allowed to spread to the remainder of the system and become visible to clients after Barricade has successfully completed validation checks to ensure that there is a low probability of operator mistakes.

Barricade is composed of several important modules, whose descriptions appear below:

Monitoring - This module’s purpose is to constantly collect data about the state of a distributed system. These include real time statistics regarding CPU/memory utilization, network traffic, and configuration file parameters. System operators’ interactions are also recorded and analyzed. This data includes command line traces and machine names/types.

Task Prediction - The reactions of a self-protecting system like Barricade are task specific. The types and numbers of resources that might be affected are dependent on the kind of task an operator is performing. If barricades are to be erected, these too, are task specific. The task prediction algorithms in Barricade continually detect what type of task is being performed to ensure the correct resources are protected. System designer defined task signatures or machine learned signatures may be used in task
Figure 4.3: This figure shows erected barriers protecting a set of machines from operator actions. These barriers are in place until validation ensures the operator is progressing without making mistakes.

Prediction.

Cost - Many times, the decision to erect barricades have to be weighed against the ability for the operator to make forward progress in accomplishing his assigned task. An ill-tuned cost module may prevent all operator mistakes at the cost of no maintenance task ever being accomplished. When designing Barricade’s cost module, factors like probability of mistakes, cost to erect blocks, and potential cost of failures are taken into consideration.

Testing - Validating system correctness can be done in any number of ways including those described in Section 2.1. For the purposes of the operator experiments, Model Based Validation was used to provide correctness models to the testing module. Assertion based programs were written for each of the tasks. This module is responsible for when to invoke and revoke actuations in the blocking module.

Blocking - Based on the results of of the testing module and cost module, Barricade erects a set of barriers around vulnerable components of the system. Figure 4.3 shows a basic three tiered system and how Barricade partitions the system to prevent errors from propagating. Blocking actions can disallow an operator from performing specific
commands, prevent him from accessing specific machines, or guard against potentially
dangerous file access. Blocking actions can be imposed and lifted continuously through
the operator’s task as conditions change.

*Guidance* - If the blocking module is activated due to an operator error, the guidance
module offers the operator information to help him pinpoint the cause of his error,
and to ultimately fix it. Traditionally, an operator would look into system state and
configuration files. In Barricade, the guidance module represents a secondary usage for
the model based validation used in the testing module. Currently firing assertions are
presented to the operator.

Barricades modules work cooperatively during 4 main phases through the operator’s
task. The containment phase limits the effects of operator actions to the minimum set
of machines necessary to begin the task and are defined for operators performing a
task on a protected system. This ensures that operators must demonstrate a baseline
proficiency (e.g. correctly performing his task on the minimum set of machines) before
he is allowed to affect operation on additional machines. Once the testing module
has determined sufficient forward progress has been achieved (and achieved correctly),
Barricade will move into a dissemination phase, where certain blocks are lifted to allow
the operator to complete the task by migrating changes to other machines.

### 4.3.2 Experiments

We evaluate Barricade using two different systems: The three tiered (web server, appli-
cation server, and database) described in Section 4.2.1, and a new enterprise computer
infrastructure. We designed the latter to reflect an administrative domain at a small
organization. This system consisted of eight machines of various tiers and redundant
components. More specifically, there were 2 client machines, 1 DNS server, 2 mail
servers, and 3 authentication servers. While the design of the tiers is very different
from our three tiered auction system, the experimental setup was very similar.

The validation module used for Barricade in our experiments is that of Model Based
Validation. The assertions used to validate the 3 tiered auction service were borne
directly out of our earlier operator experiments. The enterprise infrastructure required
a new set of validation programs. We developed a set of assertions for each of the new
tasks described below.

As in the previous experiments, volunteer computer operators of various skill levels
were given tasks to accomplish. These 20 operators were observed during the course of
their interactions and their commands, mistakes and reactions were recorded. For the
three tiered auction system, the set of operator tasks was the same as the maintenance
tasks studied by Nagaraja et al in [10]. We also added three additional tasks involving
the enterprise infrastructure. We summarize these tasks below:

*Add Webserver:* This task required the administrator to download, unpack, and
install the an Apache webserver. The administrator then had to configure and start
the requisite services.

*Migrate Database:* This task simulated the migration of database services and data
from one machine to another. The administrator had to install and configure mySQL
on a new host, stop the mySQL service on the original database server, copy data to
the new machine, start the new database service, and configure all application servers
to utilize the newly installed database.

*Upgrade Application Server:* The administrator was responsible for upgrading the
Tomcat services on all application machines. His actions here included downloading and
extracting the new software version, then, installing and configuring the new application
server to behave as the old one did.

*Add Application Server:* Much like add webserver above, the administrator was
asked to install and configure the application server on a new machine.

*Add MX Server:* This task involved creating a fail-over mail server using postfix,
and required the administrator to install and configure the Postfix mail server, and
configure the DNS server to advertise its existence. We created a set of 9 assertions to
describe this correctly executed task.

*Add a LDAP Server:* The administrator was asked to install and configure an
LDAP server and convert existing client machines from NIS to LDAP and Kerberos for
authentication purposes. We created a set of 18 assertions to describe this correctly
executed task.
Migrate LDAP Server: This task was similar to the database migration in that the administrator needed to install new instances of the LDAP service on new machines and copy the existing data from the primary LDAP server. We added an additional 21 assertions to describe this correctly executed task.

4.3.3 Results

The former case study was chosen to reflect the experiment setup of [10], with the same tasks, and the same number of experiments with operators of the same experience levels. During the 43 experiments of the 3 tiered system, we observed 37 mistakes that affect the system in various ways. Of the 37 mistakes, Barricade was able to contain 34 of the errors and prevent propagation. The uncaught mistakes were due to either 1) implementation bugs in Barricade or 2) incomplete tests in the testing module.

The second case study of the enterprise infrastructure had 15 operator experiments (3 per task) with 39 observed mistakes. Again, Barricade was able to detect and contain the majority of mistakes (36). The entirety of the uncaught mistakes were due to incomplete tests.
Chapter 5

ADIVA: A tool for Automatic Discovery of Important Validation Attributes

Thus far, the onus to create valid and correct models for model-based validation is placed entirely on the system designer. We stipulate that such a person must have expert knowledge of his/her systems and are best qualified to create the models used in model based validation. However, we also recognize that the state space of all the attributes of a distributed system that can indicate incorrect component behaviors is vast. In this section we examine how we can leverage machine learning techniques to assist the designer with the challenging task of creating more robust models.

As mentioned above, previous work has been done in attempting to contain operator mistakes by using the idea of “validation”. In [60] model based validation was used as a tool to detect when or whether operators successfully completed a task. This required system designers to include a set of rules, a priori, of what properties represented a correctly completed task. One noticeable consequence of this type of system is that the nature of mistakes one is able to detect is largely dependent on how complete (or incomplete) the designer’s view is of his system.

Other works [61–63] have used machine learning techniques to detect errors/anomalies in the operation of systems to varying degrees of success. The common theme, however, in these cases was a large amount of data available for training that provided decent coverage for most classifications.

In a large environment where a system designer’s knowledge is necessarily incomplete, and there is a lack of large amounts of training data, we find need to find a better way to leverage what we do have.

In this work, we propose a method, for Automatic Discovery of Important Validation Attributes (ADIVA).
Attributes (ADIVA), to ease the creation of validation rules, while providing more structure to the way rules are created. We then further explore other techniques of filtering and synthetic trace generation and compare the quality of the results produced by the machine learner. Ultimately, we seek to build a tool to help a human write good rules, where the tool helps to identify potentially important attributes that should be considered for inclusion in the human-generated set of rules.

5.1 The ADIVA Tool

The overall schematic design of ADIVA appears in 5.1. We begin with a set of traces of operator actions during the completion of some system administrative task. Each of these traces has associated with it, a set of system state and a classification of whether the respective operator was successful in the course of his actions. These traces are then fed into a machine learning algorithm, which in turn, produces a set of rules that classify the traces as either “good” or “bad”. Figure 5.2 shows a reasoning engine that provides feedback to refine the rules generated by the machine learner to prevent data set over fitting. Here, the system designer can encode his view of the system through a series of rules and transitions. Along with the machine learner’s output, we can then generate a series of challenge traces used to refine the training data - to ultimately guide the machine learner towards utilizing the additional information provided by the system designer.

In order to create a machine learning environment that will reveal important attributes and their respective values, we must first collect training and testing data to
initialize the learner. We obtain this data from the series of traces and operator action observations we have gathered through past experiments - as described by Section 4.3.2. We describe the data collection efforts in more detail in the next section. By tagging each trace as either good or bad (in terms of operator action correctness), we create a baseline machine learner classifier. The accuracy of this initial classifier suffers mainly from the limited data contained within the training data set.

This tool attempts to give the machine learner a feedback loop with which it can refine its result set. The premise of the tool is creating a state space for the machine learner through the operator’s actions and associated system changes. This state space includes command line traces, and their effects on system state, as it does in the aforementioned baseline classifier. However, ADIVA incorporates a distributed system designer’s vast knowledge of the system that he is building. Through an iterative process, this knowledge is applied to the initial classifier to produce higher quality results. Our goal is to be able to identify the parameters that differentiate between good and bad system state, as well as identify their associated values.

5.2 Data Processing

The traces from the studies conducted in [60] and [10] provided the basis for our training and testing data. The traces were bash command line traces that contained all
commands typed by the operators and the machines on which they were typed. Missing, however, were the various starting conditions and transitions of the systems as the operations were performed.

We first completed the data set by including system state features. We hand simulated each trace in the original set and observed and recorded the changes in a few critical categories: process changes, port changes, file system changes, and critical resource changes. In some cases, there was not enough information from the traces or the authors’ descriptions to make a determination on how system state was affected (e.g. `vi file.ext` could either have no effect on state features, or could indicate `file.ext` was edited). Sometimes, context was helpful (`files.ext` was a log file) and in others, still ambiguous (`files.ext` was a configuration file). In the most ambiguous cases, we made educated assumptions about operator intent. Alternatively, the traces could have been replayed via a virtual machine or identical system, with the results of state changes recorded. However, the same caveats apply using this method, as assumptions on user intent via commands still must be made.

The actual traces in the data set were then processed by the UIMA [64] framework for text processing. The processed traces were comprised of approximately 900 tokens representing directories, files, commands, arguments, users, and machines found in the traces. We created document processors to report tokens as a binary feature (exists/does not exist) and as frequencies (how many times a token has appeared in a trace). The document processors also merged the system state features into the format (ARFF) understandable by the machine learning package.

In the migrate database task, there were 950 attributes in all: 30 process/port/file attributes (10 per tier), 15 file specific attributes, 900 tokens, and 5 miscellaneous support attributes. The process/port/file attributes consist of attributes indicating the number of processes started or stopped, the number of ports opened and closed, and the number of files created, deleted, and modified. The file specific attributes refer to specific critical resources, and whether they have been created, deleted, or modified. Of the nine hundred tokens, approximately 20 - 30 were found in the traces.

As a general example of the collection and production of the attribute/value pairs,
consider an operator task that requires the operator to change a configuration file and restart a service. The command trace listing appears below:

```bash
$ emacs httpd.conf
$ /etc/init.d/httpd stop
$ /etc/init.d/httpd start
```

In this very simple set of 3 commands, the state space and resulting attribute/value pairs collected include the following:

**Trace Tokens** - The tokens processed from the command line include all of the alpha-numeric characters. We have also chosen to include the period character in the tokens, as often, these will identify files. In the above example, these would be the set - \{emacs, httpd.conf, etc, init.d, httpd, stop, start\}.

**File Attributes** - Assuming the operator had changed the file `httpd.conf`, the corresponding `httpd.conf-modified` attribute will be incremented. Every file’s modified count can be incremented by at most once per instance. Change is identified from the file state before the start of the task. Only task critical files are enumerated as separate attributes. In other words, if the operator opened and edited a file, e.g. `out.tmp`, the modification would increment the `generic-file-modified` attribute.

**Process Attributes** - Process attributes are calculated by comparing the process table before and after the task. Even processes are restarted will be identified in this methodology, as process IDs will change. The result of the commands above will result in 1 new process and 1 dead process.

**Port Attributes** - New and dead ports are identified the same way as new and old processes are identified. In the case of restarted processes, if the process always occupies a standard port, as httpd does, no new or dead ports are identified.

**Configuration Parameters** - Depending on whether the file edited is critical, a driver may exist to allow us to collect parameters from within the configuration files. This allows us to determine what actual values the operator has changed.
Classifier Model (A)
JRIP rules:
==========

(web.xml-mod >= 4) and (machine15 = 0) and (cp = 1) => Mistake=0 (2.0/0.0)
(killall = 1) => Mistake=0 (1.0/0.0)
(startup = 0) and (emacs = 1) => Mistake=0 (1.0/0.0)
=> Mistake=1 (17.0/0.0)

Number of Rules : 4

Figure 5.3: This is the rule set that represents a classifier for the Migrate Database task. Each line is a separate rule and each parenthetical statement is a clause. 0 and 1 indicate absence and presence of a token, respectively. Mistake=0 indicates that the task was completed successfully, while Mistake=1 indicates that a task was not completed successfully. The numbers in parentheses represent the actual weights of the traces that are accurately represented by that rule, and those that are not, respectively.

5.3 Machine Learner

Using the Weka [65] machine learning package, we experimented with many of the algorithms available to find the best fit for our data set. The rules based algorithm JRip showed the best baseline accuracy for our training set.

JRip is an implementation of the propositional rule learner, Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [66]. Using a series of stages to build and prune rules, JRip outputs a set of classification rules for the minority class. The rules take on the form shown in Figure 5.3.

The clauses between parentheses on the same line being AND, and those on different lines being conditional ORs. Each of the clauses contains a feature (either a token from the trace, or an associated state feature) and its value. Values for tokens can be 0 for “not found in trace” or 1 for “found in trace”. Values for state features can be binary as in those for tokens, or can be numerical, representing the actual integer value for that state feature (i.e. number of new processes started). The rules are listed to classify the minority class first (in this case, the traces where there were not any mistakes). The
majority class is classified by an “else” statement. In other words, all traces that are not classifiable by the JRip rules will be classified in the majority class (in this case, those traces with a mistake). In parentheses after each rule, JRip also provides us with information on how many training instances are covered by each rule, and how accurate the entire set of rules is in classifying the training data (correctly/incorrectly).

In its default configuration, JRip attempts to prune and optimize the set of rules to use as few rules as possible in classification. This sometimes results in rules that do not accurately classify all training instances. In order to achieve a 100% accuracy on our training data, and at the risk of over fitting, we configured JRip to use as many rules as possible for classification and disabled pruning. This is useful in providing many more opportunities to refine the rules using system designer knowledge and provide the learner with the best chance to discover important attributes.

5.4 A Sample Run of ADIVA

Using the annotation process described in Section 5.1 and the JRIP rule learner, the classifier composed of the rules listed in Figure 5.3 were produced. We immediately see some problems with this original rule set. The attributes the learner has chosen to represent the rules is seemingly disjunct, and too general. The learner was only able to discover one attribute (web.xml-mod) that is specifically related to the database migration task. Due to the limited training set size, this attribute is enough to correctly classify the training set. This one attribute, however, is insufficient to completely describe a correctly behaving system in all situations. The goal of ADIVA is to develop a more complete set of attributes to describe correctly behaving systems.

ADIVA accepts three sets of inputs: (1) a representative set of operator actions (i.e., a set of traces) denoting actions that are erroneous (bad traces); (2) a representative set of operator actions denoting actions that are not erroneous (good traces); and (3) general rules of how to transition traces from good to bad (as well as good to good and bad to bad) states. These rules codify domain-specific knowledge about the properties of good and bad traces, but are not expected to be comprehensive. The output of ADIVA is a
Classifier Model (B)
JRIP rules:

```
(web.xml-mod >= 4) and (tier-db-X.modif-c <= 0) and (ssh = 0)
 => Mistake=0 (2.0/0.0)
(web.xml-mod >= 4) and (startup = 0) => Mistake=0 (1.0/0.0)
(machine15 = 0) and (machine14 = 1) and (kill = 0)
 => Mistake=0 (1.0/0.0)
 => Mistake=1 (28.0/0.0)
```

Number of Rules : 4

Figure 5.4: This figure shows the rule set after to the first feedback iteration.

set of rules that can be used to identify good traces from bad traces.

*ADIVA* uses a refinement loop to arrive at the set of rules that it outputs. *ADIVA* begins by presenting features extracted from the good traces and bad traces to a machine learner. The output of this step is a classifier (i.e., a set of rules) that differentiates good traces from bad traces. These rules capture the properties that caused the good and bad traces to be labeled as such. While these rules can themselves be used to achieve the goal of *ADIVA*, in practice, we have observed that they may need to be supplemented with domain-specific knowledge. *ADIVA* uses domain-specific rules in combination with its trace generator component to produce traces that contradict the classifier, e.g., traces that are erroneous according to the domain-specific rules, but are classified as good by the machine learner, or vice-versa. These traces are presented again to the machine learner, labeled as determined by the domain-specific rules, and the whole process iterates. *ADIVA* can be configured in a variety of ways to terminate this iterative process. For example, we can specify the maximum number of iterations, or supply thresholds for the precision and recall of the classifier output by the machine learner.

To illustrate the workflow through *ADIVA* we shall use the Database Migration task as described in Section 4.3.2. This task requires an operator to successfully migrate a database from one machine to another.
We use ADIVA’s refinement loop to generate additional traces to help the learner find additional relevant attributes. Each attribute is examined and the feedback mechanism attempts to produce a trace addition that violates the clause. Take the cp clause Figure 5.3: we can create traces that violate this part of the rule in the following way. Since the clause states that the token ‘cp’ should exist in order for a trace to be classified as a good trace, one way to violate the clause is to generate a trace that has the token but is classified as “bad”. In the case of ‘cp’, cp can be generated by copying and overwriting system files, rendering any originating trace (good or bad) bad. We then add these newly modified traces back into the training set and rerun the learning algorithm. Figure 5.4 shows the rules that resulted from this second iteration.

We see a change in the rule set after the addition of the generated traces. There is an increase in the number of clauses that are specific to the database migration task (from 1 to 2). We show that through multiple iterations of this process, we can effectively increase the quality of the rules generated by the learning algorithm, and in some cases, see an increase in the accuracy of the rules themselves. We will investigate these changes, both qualitatively and quantitatively in the following sections.

5.5 Rules Engine

In deciding how to design the structure to codify the system designer’s expert knowledge, we first examine the requirements for such a tool. In the experiences described in [60], we found that any useful tool would have to allow for the expression of the following attributes:

- assumptions about the system - what the preconditions are for the systems upon which operators work
- facts about the system - what machines belong to which tiers, who are privileged users, etc
- dependencies - are there tokens that affect the existence of other tokens
- transitions - how one might transition from “good” traces to “bad” or any other
such permutations

- known bad sequences - increase the knowledge of the system using observation over time

As with choosing any tool, there are many that exist that will do some fraction of the above with various levels of simplicity and/or effort. We chose to implement our rules engine in Prolog for our ability to organize facts/rules as well as its exhaustive search and backtracking. It should be noted, however, any number of tools could equally have sufficed.

Again, the facts describe the current configuration of the system. Some facts like file, editor, process and command remain largely stable across many installations at different organizations, while some (machine, user) are stable within an organization. Still others (trace, tracestate) are specific to individual tasks within an organization.

The tracestate predicate is generated through the process described in Section 5.2, and is simply a vector with all the state based features, including tier specific versions of process changes (started/stopped), port changes (added/removed), and files modified/deleted/created. This is a one-to-one mapping between traces in the training set. Similarly, the tracelist predicate is the Prolog representation of a full trace, derived from each trace in the training set.

Given the structural rules for trace generation and the facts regarding the system environment, the high level predicate present combines the other predicates to create new traces that include a specific attribute.

If we go back to our running example of the Migrate Database task, Table 5.1 shows example instantiations of facts and rules.

The refinement program consists of five sections or modules: Traces, TraceStates, Utilities, Facts, and Transitions. Traces are translated and filtered traces collected from operator experiments. The traces maintain the order of commands and arguments and allow for manipulations of the original trace data. Similarly, TraceStates are the internal representation of state attributes and are also formatted to allow for manipulations. There is a one to one relationship between Traces and TraceStates. Utilities are just a
<table>
<thead>
<tr>
<th>Type</th>
<th>Category</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fact</td>
<td>hosts</td>
<td>machine(db,machine, mysql). machine(trillian04, apache). machine(trillian06,tomcat)</td>
<td>Machine named db_mahcinge resides on the mysql tier</td>
</tr>
<tr>
<td>Fact</td>
<td>editors</td>
<td>editor(vi). editor(emacs). editor(nano).</td>
<td>vi, emacs, and nano are all editors</td>
</tr>
<tr>
<td>Fact</td>
<td>files</td>
<td>file(httpd.conf). file(my.cnf). file(somedir,10).</td>
<td>httpd.conf and my.cnf are files, some dir contains 10 files</td>
</tr>
<tr>
<td>Fact</td>
<td>commands</td>
<td>command(httpd). command(startup.sh). command(mysqld-safe).</td>
<td>httpd, startup.sh, mysqld-safe are commands</td>
</tr>
<tr>
<td>Fact</td>
<td>services</td>
<td>service(httpd). service(nfsd). service(sshd).</td>
<td>httpd, nfsd, and sshd are services</td>
</tr>
<tr>
<td>Fact</td>
<td>critical</td>
<td>critical(my.cnf, migratedb). critical(httpd.conf,addweb). critical(startup.sh, upgradeapp)</td>
<td>my.cnf is a critical resource in the migrate db task</td>
</tr>
<tr>
<td>Fact</td>
<td>trace</td>
<td>trace(tracefilename, [commands,in,trace ... ]). state(tracefilename,[state features])</td>
<td>The import of the traces into prolog</td>
</tr>
<tr>
<td>Rule</td>
<td>composition</td>
<td>editfile([E,F],State) :- editor(E), file(F), changestate(State,F)</td>
<td>an edit file command is comprised of an editor and a file. changestate changes the state appropriately for this type of operation</td>
</tr>
<tr>
<td>Rule</td>
<td>good transition (not bad)</td>
<td>delfile(<em>,N,[rm,X],0):- del([rm,X],N),not(critical(X,</em>))</td>
<td>Deletions of files not listed as critical are considered not bad</td>
</tr>
<tr>
<td>Rule</td>
<td>bad transition</td>
<td>newproc(Tier,N,Trace,1) :- invoke(Trace,N,_), service(X), memberchk(X,Trace), not(critical(X,Tier))</td>
<td>It is considered bad to start services not related/critical to the current task</td>
</tr>
</tbody>
</table>

Table 5.1: Example facts and rules in the refinement program.
collection of predicates that assist in filtering, composing, and otherwise manipulating traces and states. Facts and Transitions are the central components of the feedback loop and deserve a more in depth review.

Facts are how the system engineer codifies his view of the system. While Prolog allows any of a number of ways to represent these predicates, we propose a more structured method of system specification. The database of facts build from top level predicates representing Tasks. Tasks are associated with tiers of service - these are the categories of machines. For example, the db migration task requires operator action on two tiers: mysql and tomcat. Each tier then has machines, files, and processes associated with it. Certain resources (files/processes) are tagged as critical resources. The facts database also includes composition rules that specify how trace elements are created (e.g. what does the invocation of a server process look like?). Together, the facts database instantiates the challenge traces created by the transition rules. Facts are also used to describe how command sequences affect system state. This is important in order to generate sound state vectors based on the command sequence manipulations (e.g. we can encode that rm + file leads to an increase in the state counter for “removed files”, or the invocation of mysqld leads to the creation of an additional process and listening port).

Transition rules fall under a few categories: good to good, good to bad, and bad to bad. Good to good transitions derive good traces from other good traces. These types of transformations involve no state changes, but do introduce new tokens into traces. For example, introducing the commands ps, less, or other side effect free commands to traces already labeled as good will produce only good traces. (This may not always be desirable, especially in cases where information security might be important, but these behaviors can be specified in the rule set). Good to bad transformations are any specifications of all the known ways that a good trace can become a bad trace. Currently these types of transitions involve 1) editing unrelated (but critical) resources, 2) starting/killing unrelated (but critical) processes, or 3) known sequences of bad behavior (in which common operator errors can be encoded). Bad traces will, barring trace specific corrections, remain bad - regardless of additions or subtractions. The
transformations are therefore the conglomeration of both 1) and 2) applied to bad traces.

Transition rules are seeded with the Traces and TraceStates. The transition rules are further divided into token attribute affecting rules and state attribute affecting rules. The query process requires a challenge token, a desired outcome (good or bad), and either the token attribute to be added/removed or the state attribute to be changed. It will return with a new trace (based off of an original trace from the training data set) and a corresponding appropriately modified state vector if one can be generated.

Here is simplified high level predicate that produces new traces from old traces and the aforementioned transition rules:

\[
\text{present}(X, \text{FinalTrace}, 1, \text{FinalState}, \text{Task}) :- \text{task}(\text{Task}, \text{Tiers}), \\
\quad \text{member}(\text{Tier}, \text{Tiers}), \\
\quad \text{tracestate}(\text{Name}, \text{Task}, _, \text{TraceState}), \\
\quad \text{addbad}(\text{Trace}, [], \text{Tier}, \text{TraceState}, \text{FinalState}), \text{member}(X, \text{Trace}), \\
\quad \text{traces}(\text{Name}, \text{OrigTrace}), \\
\quad \text{mergeTraces}(\text{FinalTrace}, \text{Trace}, \text{OrigTrace}).
\]

The unbound variables FinalTrace and FinalState are what we wish to query the program for. We would like the final trace to contain the token X, and the resulting trace + state to be considered a “bad” data point. As mentioned above, bad data points can result from bad command sequenced appended to either good or bad traces. So the first task is to obtain an original trace (tracestate) meeting our requirements. We do an exhaustive search over all the tiers involved in the completion of the tasks and over all possible producible command sequences to find one that contains the token (addbad + member). We then adjust the state vector (addbad) and merge the newly produced command sequence with the original training trace (mergeTraces + traces).

The addbad predicate is a conglomeration of many known bad command sequences, both generic and specific. Generic addbad sequence include adding/deleting/modifying/replacing critical files that are not recognized as part of the task (i.e. deleting /etc/passwd during the course of the addweb task). Similar predicates exist for the
creation and deletion of critical processes (killing sshd during the course of the migrate database task). Specific addbad predicates can be formed from a system engineer’s previous experiences. Knowing that a specific sequence of commands has, in the past, caused problems allows the engine to learn what the engineer has learned.

5.6 Refinement Loop

As we have alluded to before, neither the machine learner, nor the rules engine can represent the system and the operator tasks completely. Our goal with ADIVA is to augment one with the other to drive the machine learner towards a better “understanding” of the system as a whole. We accomplish this by using a feedback mechanism to refine the rules that the machine learner produced.

| Collect training traces (associated with task); |
| Document (pre)Process the traces, extracting tokens; |
| Extract and store system state associated with each trace; |
| \textbf{while} stopping condition not met do |
| Run ML (JRip Classifier on training set; |
| \textbf{Output:} Classifier outputs set of rules $R = R_1...R_n$ |
| for $i \leftarrow 1$ to $n$ do |
| $R_i$ is made up of clauses $C_1$ to $C_m$ |
| for $j \leftarrow 1$ to $m$ do |
| \textbf{if} $R_i$ predicts class 0 \textbf{then} |
| Query prolog for trace that: |
| Takes class 0 trace, add additional lines to violate $C_j$ to produce class 0; |
| Takes class 0 trace, add additional lines to follow $C_j$ to produce class 1; |
| Takes class 1 trace, add additional lines to follow $C_j$ to produce class 1; |
| end |
| Add created traces to training set; |
| end |
| end |

\textbf{Algorithm 1:} Algorithm for refinement loop. JRip creates rules for the majority class, in this case the “good” traces (those in class 0). In the unlikely case that the majority case flips to class 1, e.g. $R_i$ predicts class 1, the refinement loop would be the inverse of what is presented above.
Algorithm 1 shows the pseudocode for the feedback loop. The algorithm does an exhaustively attacks each of the clauses in the rule set. It accomplishes this by querying the prolog rules engine for a trace that 1) violates the clause while having the same classification and/or 2) follows the clause while having a different classification.

The generated traces are then folded back into the training data, whereupon the machine learner is once again invoked, and the process repeats. There are a variety of stopping conditions that indicate either that the iterations were successful or that the algorithm is unable to converge to an optimal solution.

The first stopping condition is when all the traces from the reserved testing set have been classified correctly. As long as the testing set is diverse enough, this will stop the algorithm when the rules have been refined enough to detect anomalies in specific tasks. If the data is not diverse enough, then this will cause the algorithm to stop too early, expressing a very minimal set of differentiating rules.

Stopping conditions can also be based on either absolute or relative thresholds. These are the hard limits: e.g. the accuracy of the machine learner has exceeded some threshold percentage, or the number of iterations through the algorithm has exceeded some number. These types of stopping conditions can help in the cases where the rule set does not converge or where it might fluctuate wildly.

Another stopping condition is when the rule set before and after an iteration through all of the clauses within a rule set are equivalent. Here, the algorithm has reached an equilibrium, and will not be able refine the rule set any more. It should be noted that this point of equilibrium may never occur in the cases where the rules do not converge. For the purposes of this study, this is the stopping condition that we have chosen to use, due to the limitations of our data set (not diverse), determining threshold values is beyond the scope of this paper. In reality, the best stopping condition is likely a composite of all the ones mentioned above.

The listing in Figure 5.4 are the result of one refinement iteration for the Migrate Database Task. We see here that the machine learner has chosen somewhat of a new set of attributes.
5.7 Experimental Methodology

Over the course of previous experiments [60], command line traces from volunteer system administrators attempting a variety of tasks were collected. While full details of these experiments can be found in the respective papers, we overview each of these tasks below in the context of our current work. Each task involved various installation and configuration tasks, with the final goal being a system that was functional and correctly configured.

For each experiment, we observed operator mistakes and resulting effects of system state. We also collected command line traces containing all command names and arguments used. As described in Section 5.1 these traces form the basis for our feature set. The entire trace was tokenized, with each token forming a unique feature. To this set, we added features that reflected state transitions for each experiment. We then generated additional synthetic traces to create the training data to be used in the propositional rule learner JRIP. Synthetic traces were used to increase the number of data points in the training set by adding and/or deleting commands to the original set of traces. This process included adding innocuous commands to good and bad traces, and adding known bad commands to good traces. The rules generated were used in cross validating the training data as well as testing new traces to determine the accuracy of classifications.

5.7.1 Operator Experiments

In each of the experiments below, volunteer system administrators were asked to perform the tasks described. They were given generalized instructions as to how to accomplish the tasks, and had whatever informational resources required to complete the task e.g. online search engines and discussion boards. Administrators were of various degrees of experience, and thus each administrator was able to complete this task with various degrees of competency. We used the the operator experiment data as described in Section 4.3.2.
5.7.2 Feature Selection

Feature selection is often a difficult task better described as an art form. Using the tokens of the traces and state features, there are over 900 features available. We can understand feature selection as a spectrum. With full knowledge of the entire system, we can select only the exact features that uniquely classify our traces into good an bad. This would allow the machine learning algorithm to only focus on those features that allow it produce rules that have differentiating power. However, having the amount of knowledge necessary to accomplish this would diminish the need for a machine learning approach at all.

On the other side of the spectrum is when there is no domain knowledge whatsoever. Here, we would be unable to select any particular features, and must include all of them. With the inclusion of all features, it is likely that the machine learner may return rules that only coincidentally differentiate the classes. While it is the goal of the feedback loop to refine and remove these anomalies, we found that there were cases where the coincidental rules remained even after feedback.

The ideal attribute selection is thus somewhere in the middle. We should use some of our domain knowledge to filter out those attributes we know to be spurious, and only include the ones that are either unknown or are definitely differentiating. Table 5.2 is a summary of our attribute selection experiments, and shows how important proper attribute selection can be to the accuracy of the machine learner.

5.7.3 Synthetic Trace Generation

While each administrator was able to complete this task, they did so with various degrees of competency. This provided an general sampling of the type types of mistakes. However, the data set collected was still relatively small. In order to create a training data set that provided more coverage, we generated synthetic traces to augment the collected ones.
<table>
<thead>
<tr>
<th>Task</th>
<th># Rules</th>
<th># Attr.</th>
<th># Match</th>
<th>% Match</th>
<th>FP</th>
<th>FN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>Min set of attr.</td>
</tr>
<tr>
<td>8 add’l</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8 add’l</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>After iterations</td>
</tr>
<tr>
<td>16 add’l</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>16 add’l</td>
<td>3</td>
<td>9</td>
<td>4</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>After iterations</td>
</tr>
<tr>
<td>31 add’l</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>31 add’l</td>
<td>2</td>
<td>7</td>
<td>3</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>After iterations</td>
</tr>
<tr>
<td>57 add’l</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>57 add’l</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>After iterations</td>
</tr>
<tr>
<td>With machine</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>names</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With machine</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>After iterations</td>
</tr>
<tr>
<td>names</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All attributes</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>All attributes</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>15</td>
<td>0</td>
<td>2</td>
<td>After iterations</td>
</tr>
</tbody>
</table>

Table 5.2: Sampling of attribute selection, and related qualitative and quantitative metrics. FP = false positives, FN = false negatives
Table 5.3: The number of attributes used in developing manual rules. Manual rules were those created for the Model Based Validation implementation described in Sections 4.2.1 and 4.3.2, modified by the constraints of the types of data we were able to collect from the traces. These are later used as a baseline for evaluating the machine learner.

The manually generated synthetic traces were derived in multiple ways. The first was merely a substitution operation, where we inserted (and labeled) a trace from another task within our experiments. This simulated an administrator attempting a task that was completely unrelated to the one expected. The second method simulated an installation of software using the default configuration. These traces were the same as the original collected traces with the tokens for editing relevant configuration files removed. The third method injected innocuous commands (e.g. non-state changing) commands to both good and bad traces, resulting in no change in classification.

5.7.4 Manual Rules

Figure 5.5 shows an example of a manual rule set used to evaluate the qualitative benefits of our methodology. It contains a mix of state-based and token-based attributes, but lacks dynamic data values as they were not collected from the original operator experiments. Without dynamic data values for the various states (e.g. throughput, etc), the manual set of rules is able to catch 2/3 of the original mistakes in traces, and would have caught all the mistakes contained by Barricade. We treat this set of conditions as a reference solution by which to measure the quality of the classifiers generated by JRIP. The assumption made here is that the better the classifier, the more conditionals it can uncover that contain important attributes as defined by the system designer. Table 5.3 shows a summary of the manual rules created for each task and the number of attributes associated with those rules.
(web.xml=4) and (mysqlpassword=1) and (db-files-mod=0)
and (db-numnewports=1) and (db-numnewproc=1)
and (my.cnf=1) and (my.cnf-mod=1) and (rubis=1) and (web-files-mod=0)
and (app-files-mod=0) and (web-newport=0) and (app-newport=0)
and (startup | restart = 1) and (mysqld=1) and (tomcat=1)
=> Mistake=0

=> Mistake=1

Figure 5.5: Manual Ruleset for our Migrate Database example task

Table 5.4 shows a summary of how each iteration of the feedback loop affect the classifier - both qualitatively and quantitatively. The iterative process continues until either no more changes to the rule set are possible, or some other threshold condition is met. The rules and attributes columns show how many rules the classifier contains and how many attributes are referenced in those rules. The greater the number of rules, the more the rules are fitted for the training data (and potentially less general the classifier). The Match column displays how many attributes the generated classifier shares with the above mentioned reference solution. Note that the matches here means that a correct attribute was chosen. Whether or not the associated value is a valid one is an entirely different metric and is one that we did not study, as the goal of ADIVA was to discover the attribute set itself. False positives are the number of “good” traces in the testing set that were identified as “bad”, and false negatives are the number of “bad” traces identified as good.

The first entry in the table is that of the classifier produced by a learning algorithm that produces 0 false positives and 0 false negatives. The number of attributes is confined to just state attributes and a few additional token based attributes as reported by Weka’s attribute ranker. In each successive subsection of the table, more attributes are added. The data shows that the learning algorithm struggles as the number of attributes increases. Within each subsection, we see that iterations of the feedback loop generally increase accuracy of the classification of the test sets, as well as increase the quality of rule set.

Additional quantitative evaluation metrics are also used to explore the effects of
<table>
<thead>
<tr>
<th>Task</th>
<th>#iter.</th>
<th>#rules</th>
<th>#Total Att</th>
<th>% Qual</th>
<th>% Match</th>
<th>FN</th>
<th>FP</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add App Server</td>
<td>1</td>
<td>4</td>
<td>11</td>
<td>2</td>
<td>25</td>
<td>0</td>
<td>1</td>
<td>85.71</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>3</td>
<td>14</td>
<td>5</td>
<td>62.5</td>
<td>1</td>
<td>2</td>
<td>57.14</td>
</tr>
<tr>
<td>Add LDAP Server</td>
<td>1*</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8.3</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Add Mail Server</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>8.3</td>
<td>2</td>
<td>0</td>
<td>71.43</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>8.3</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Add Web Server</td>
<td>1*</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Migrate LDAP Server</td>
<td>1*</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>15.4</td>
<td>1</td>
<td>1</td>
<td>71.43</td>
</tr>
<tr>
<td>Migrate Database</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>5.3</td>
<td>2</td>
<td>0</td>
<td>88.24</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>5.3</td>
<td>1</td>
<td>0</td>
<td>94.11</td>
</tr>
</tbody>
</table>

Table 5.4: This table shows the results of multiple iterations through the refinement process. FP = false positives, FN = false negatives, Acc = accuracy. * Entries where only 1 iteration appears indicate runs when the feedback loop was unable to change the initial ruleset.

ADIVA. The standard set of metrics reported by the machine learning algorithm serves as a starting point for evaluating the performance of ADIVA. Accuracy is determined by how many instances of the test set have been classified (predicted) correctly versus how many were incorrectly classified. More specifically, the output matrix of the machine learner provides information about the false positives and false negatives of each set of classifying rules.

We ran the algorithms presented in Section 5.1 on the traces of tasks described in Section 5.7.1. Each of the task were of varying levels of complexity, with different types of mistakes. We show the efficacy of ADIVA using a few metrics.

### 5.7.5 Results

The utility of ADIVA is dependent on a number of factors. Below we outline the results as it pertains to each of the tasks, and discuss how to interpret the results. Table 5.4 shows a summary of running the refinement algorithm on the set of tasks described in Section 4.3.2. The attribute set has been filtered to reflect only files, commands and system state.

Add a web server - This was an extremely simple task requiring installation of some files, editing of a couple of files, and finally running a few commands to start processes.
The JRip algorithm was extremely efficient in producing a set of rules covering all training instances and produced 2 rules involving 2 attributes. This provided 100% accuracy over all training and testing sets. Attempts to refine the quality of the rules failed to produce a different rule set. As in the other task for which this was true, this inability to progress is a result of the necessarily incompleteness of the transformation rules.

Add an application server - This task was a little more complex than add a web server - requiring slightly more complicated configuration changes. The first rule set produced by the machine learner consisted of 4 rules involving 11 attributes classified the testing set with 1 false positive and no false negatives. Of all the attributes, 2 were identified as having some qualitative value. At the end of all of the iterations of the refinement process, the testing set was classified with 2 false positives and 1 false negative. Qualitatively, the ending rule set contained 5 attributes matching the manual rule set.

Migrate database - This task was more difficult than the previous two, by virtue of having more components, more services, and more configuration files. We saw in Section 5.7.2 how important attribute selection was in this case. There were a total of 6 distinct attributes over the 3 rules given by the machine learner. After the completion of all possible refinements, overall accuracy became 94% with no false positives. Note here, that while the quantitative result has improved (accuracy), the qualitative result has remained the same. This is likely due to the small training set and that the accuracy threshold was met by the machine learner.

Add a mail server - This is the first of the tasks on the enterprise-like administrative systems. Here, 71% accuracy on the test set and the training set is achieved during the first iteration. Three attributes over two rules were derived, with 1 qualitatively interesting attribute. At the end of refinement, accuracy is increased to 100%. Note here, also, that while the quantitative result has improved (accuracy), the qualitative result has remained the same. This is likely due to the small training set and that the accuracy threshold was met by the machine learner - even more so than the above task, as the machine learner was able to achieve 100
Migrate LDAP server - After the first iteration, test data showed classifier accuracy at 71% with two rules, 5 distinct attributes, and 2 qualitatively significant attributes. The refinement loop was unable to add to the training set.

Add LDAP server - Subjectively, this was the most difficult task. Operators, on average, took longer to complete this task than any of the previous ones. Surprisingly, only one iteration was enough to achieve 100% accuracy on both testing and training data sets. After investigation, it appeared that all of the operators that completed the task incorrectly misconfigured the same configuration parameter. One attribute was enough to differentiate those who successfully completed this task from those who did not. No refinement iterations were possible.

5.8 Data Space Exploration

With such a limited data set, the results shown in the previous section can only give anecdotal evidence that the machine learning techniques can be used to help determine important attributes. In this section, we attempt to explore how changes to the original dataset might affect machine learner performance.

5.8.1 Attribute Filtering

In order to gain more insight into the effects of attribute filtering, beyond that seen is Section 5.7.2, Table 5.5 shows what occurs when filtering is done in a systematic manner. Here we observe how the machine learner copes with different types of attributes. We see that for three of the tasks, system state attributes were sufficient to achieve 100% accuracy in the testing set. This means that for the mistakes observed during those tasks, simple monitoring of file changes and configuration parameters with associated rules were sufficient for detection. In the remaining cases, adding attributes seemed to confuse the machine learner into choosing sub optimal attributes. Specifically, if we look at the migrate database task (mysql), we see that accuracy dropped from 100% to just 5% when given more attributes. With the refinement loop in place, the latter was improved to 53%.
<table>
<thead>
<tr>
<th>Type</th>
<th>#iterations</th>
<th>Accuracy</th>
<th>#attributes in rules</th>
<th>#attribute matching</th>
<th>% Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Application Server</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>57%</td>
<td>10</td>
<td>8</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>II</td>
<td>42%</td>
<td>5</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>III</td>
<td>42%</td>
<td>7</td>
<td>7</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>IV</td>
<td>85%</td>
<td>10</td>
<td>3</td>
<td>38</td>
</tr>
<tr>
<td>Add LDAP Server</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>100%</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Add Mail Server</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>100% *</td>
<td>3</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>Add Web Server</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>100%</td>
<td>2</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>Migrate Database</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>100%</td>
<td>4</td>
<td>4</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>II</td>
<td>35%</td>
<td>35%</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>III</td>
<td>94%</td>
<td>94%</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Migrate LDAP</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>85%</td>
<td>2</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>II</td>
<td>57%</td>
<td>7</td>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>III</td>
<td>57%</td>
<td>8</td>
<td>7</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>IV</td>
<td>71%</td>
<td>4</td>
<td>2</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.5: The effect of filtering feature set on accuracy and relevancy. Splits in the last two columns indicate results prior and following the refinement loop. * indicates errors predictions in the training sets. Type I: Files and state, II: Commands and state, III: Files, commands, and state, IV: Files, commands, state, + attributes from other tasks.
5.8.2 Attribute Consolidation

Our experiences with using the feedback loop showed that many times, the system stopped because the encoded rules were not complete/sophisticated enough to challenge any more attributes. Besides filtering, attribute consolidation has the potential to further assist the machine learner by reducing the number of attributes while not affecting the semantics of operator traces. Here we attempt to group similar commands together into classes of commands. For example, the EDITOR class contains the attributes vi, emacs, pico, nano, gedit and the MACHINE class contains the attributes of all machine names. Unfortunately consolidating these attributes made no difference in the accuracy or relevancy of the generated rule sets.

5.8.3 Attribute Removal Algorithm

If we use ADIVA as a guide to draw designers toward important attributes, a secondary method to reduce the attribute set is to continually remove attributes to reveal those that are truly useful. We wish to filter the attributes discovered by the machine learner and force it to discover additional attributes that may be important. The hope is that this can overcome the limitation of the conservative machine learner. Throughout this process the system engineer can collect the attributes that were output by the machine learner at each iteration. This can be the starting point from which he can create better rules to identify errant and correct behavior. Ideally, after many iterations, we can expose all of the qualitatively important attributes. The purpose of this exercise is to determine if the machine learner is capable of identifying all of the attributes that the ideal manual rule set contains. In each of the tasks, a maximum of 10 pinning iterations were required to reveal all such attributes. Furthermore, after these 10 iterations, the machine learner was unable to create any rules to differentiate the two classes (good vs bad), and defaulted to one “rule” that placed all instances into the majority class. Table 5.6 shows how this method was able to produce more matches to manual rules than the machine learner’s first attempt.
<table>
<thead>
<tr>
<th>Task</th>
<th>Type</th>
<th>Before</th>
<th>Iter</th>
<th>After</th>
<th>Relevant</th>
<th>Noise</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add App Server</td>
<td>Min</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Add LDAP Server</td>
<td>Mid</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>Add MX Server</td>
<td>Full</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>20</td>
<td>28</td>
</tr>
<tr>
<td>Add LDAP Server</td>
<td>Min</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Add LD Server</td>
<td>Med</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>4</td>
<td>13</td>
<td>27</td>
</tr>
<tr>
<td>Add MX Server</td>
<td>Full</td>
<td>1</td>
<td>7</td>
<td>10</td>
<td>5</td>
<td>11</td>
<td>26</td>
</tr>
<tr>
<td>Add MX Server</td>
<td>Med</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>Add MX Server</td>
<td>Full</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>Add Web Server</td>
<td>Min</td>
<td>1</td>
<td>4</td>
<td>13</td>
<td>2</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Add Web Server</td>
<td>Med</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td>2</td>
<td>15</td>
<td>28</td>
</tr>
<tr>
<td>Add Web Server</td>
<td>Full</td>
<td>2</td>
<td>7</td>
<td>14</td>
<td>2</td>
<td>15</td>
<td>31</td>
</tr>
<tr>
<td>Migrate DB</td>
<td>Min</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>Migrate DB</td>
<td>Med</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>17</td>
<td>28</td>
</tr>
<tr>
<td>Migrate DB</td>
<td>Full</td>
<td>1</td>
<td>11</td>
<td>19</td>
<td>5</td>
<td>19</td>
<td>43</td>
</tr>
<tr>
<td>Migrate LDAP</td>
<td>Min</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Migrate LDAP</td>
<td>Med</td>
<td>3</td>
<td>11</td>
<td>15</td>
<td>1</td>
<td>15</td>
<td>31</td>
</tr>
<tr>
<td>Migrate LDAP</td>
<td>Full</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.6: The effect of increasing number of attributes on the Attribute Removal Algorithm. Min sets indicate a very minimal set of relevant attributes for the listed task, Full sets indicate the entire attribute list, and Med sets have only relevant attributes from the entire corpus of attributes in every task studied. Before and After refer to the number of matches before and after the iterations of attribute removal. Matches are 1:1 matches to important attributes as identified in the manual rule sets. Relevant attributes are those attributes that are related to the task at hand, but were not part of the manual rule set. These attributes may not have been critical to correctly executing the tasks, but they are nonetheless relevant. Noise attributes are those that are entirely irrelevant to the task at hand (e.g. no-ops).

### 5.8.4 Artificial Data Expansion

Finally, to investigate how a larger data set might affect the machine learner, we performed an artificial data expansion. While keeping all attributes values necessary and sufficient and modifying all other attributes randomly, we generated 2000 samples (1000 positive and 1000 negative). The rule set produced by this random trace generation differed little from our original rule sets (without the random generation). While the random generation reduced the number of spurious attributes in the rule set, the primary attributes were sufficient to satisfy the machine learner’s threshold. Without the spurious attributes, the feedback loops are unable to assist in creating additional rules. However, the behavior of applying attribute pinning is the same as described above.
5.9 Discussion and Limitations

ADIVA in its current incarnation is useful insofar as it gives system designers a starting point from which to begin creating rules for classifying good vs bad operator behavior. However, ADIVA still requires human oversight. Ideally the system designers have ADIVA, among other tools to assist them in deriving a good set of manual rules to identify operator errors. The need for a proper framework like [60] is still necessary to prevent operator errors from spreading.

5.9.1 Open Issues

As we have seen in Section 5.7.5, the results of the initial rule set produced by the machine learner is insufficient due to the large attribute space that the traces cover. Attribute selection can be a long and arduous task that is largely dependent on the designer’s knowledge of their system. Any incorrect assumptions can lead to inaccurate classifiers. Optimally filtering attributes is an open problem. Too much filtering, besides potentially inaccurate, negates the need for ADIVA: i.e. if the designer knows enough to filter out all extraneous information, he does not need a tool to do the same thing. Little to no filtering can make it difficult for the machine learner to zero in on the important attributes. We propose a solution that “types” attributes to allow the designer to filter out classes of attributes to find the one that best suits his system. In our test cases, we found that filtering out machine names led to more quality results.

In situations without a large training dataset covering a large percentage of possible operator mistakes, it is possible that the machine learner produces rules that achieve optimal accuracy rates with very few attributes and rules. In these cases, the refinement loop may not run at all (as observed in the Migrate and Add LDAP Server tasks in Section 5.7.5). Without any additional intervention, the system designers will have little guidance from ADIVA. We have explored the concept of “pinning” attributes. By removing all attributes in the final iteration of the refinement loop, and rerunning the machine learner, we were able to discover additional attributes and values. Doing so obviously adversely affected the accuracy of the resultant classifying rules (as we had
removed the most differentiating attributes from the pool). However, this technique served its purpose in providing us with an extended set of attributes.

The last open issue is the formulation of rules after using ADIVA. ADIVA gives system designers insight into the attribute and values that can serve as a basis to write the rules they use to detect operator errors. Each iteration of the refinement loop can potentially reveal new and useful pairs, but it is unclear what the best methods of combining the attributes may be. Aggregation of all attribute value pairs can miss subtle dependencies. Using all of the rule sets of each iteration in succession can provide the basis of a consensus based machine learning strategy.

5.9.2 Future Work: Quality and quantity

Machine learning thrives on data, and it is foreseeable that large data centers can produce enough data to alleviate some of the issues described above. Besides the very real need for more data, there are a few improvements that can be made to the quality of the attributes. While we were limited in our ability to collect data from existing experimental data, we believe that other types of data could potentially increase the usability of the results of the machine learner. Many of the following proposed improvements increase the fidelity with which a human might be able to distinguish between correct and incorrect operator behavior. They should also give the machine learner more distinguishing features.

**Timestamps** - Temporal data of commands, process starts, and file changes can provide new perspectives on correctness, and allow for finer distinctions between correct and incorrect behavior. For example, an operator who edits a file and then restarts the corresponding process will have different results from the operator who restarts a process and then edits the corresponding configuration file. Currently, the machine learner would interpret both of those sets of actions as equivalent. Beyond just simple ordering, time stamps can also provide insight as to how long operators take to accomplish subtasks.

**Command structure** - Currently tokens are produced from the command traces and broken down into individual units of contiguous alphanumeric characters. This
method is common in machine learning as a “Bag of Words” technique, and can be useful for analyzing texts. However, what is lost in the conversion is structural information about the commands. Ideally, we could build semantic meaning into strings of commands where argument placement of commands is, itself, an attribute. This could allow the machine learner to differentiate between the commands `cp foo bar` and `cp bar foo`.

**File System Versioning** - Knowledge of which files have changed during the course of an operator task has been useful, but we posit that knowing *how* files have changed could prove even more useful. If our systems had, at the very least, version control of configuration files, the machine learner could leverage changes that occurred in any parameter within the configuration file, rather than just specifically chosen ones.
Chapter 6
Conclusions

In this dissertation we investigated using models combined with validation as a strategy for identifying and hiding operator mistakes in Internet services. We recognize that an important source of Internet service failures is that of operator mistakes, and designed tools to allow system designers to reason about correct behavior of those systems.

Model Based Validation addresses some significant limitations in previous work done in validation including detecting latent mistakes and bootstrapping validation with no known instances of good behavior. In support of Model Based Validation, we have created a domain specific language called \( A \) that allows system engineers to codify their beliefs about the correctness of their systems.

While outside the scope of this particular work, the domain specific language implementation opens a world of possibilities in terms of analysis of what system engineers believe to be correct models of their systems. In the future we can leverage static analysis techniques of \( A \) programs to analyze everything from logical inconsistencies within models, to identifying single point of failures within large systems. One could imagine that with complete enough \( A \) programs, one could even optimize how validation is completed. Furthermore, merging multiple systems introduces new challenges in terms of how \( A \) programs, themselves, could be merged and reconciled.

For the purposes of Model Based Validation, we developed three high-level models to capture the flow, component, and security correctness properties in a service and realized these models in \( A \). We found that the models were straightforward to express in \( A \), and the features of the domain specific language were useful in organizing and maintaining the validation programs.

The \( A \) language would be incomplete without its corresponding runtime system.
In this work, we implemented a runtime monitoring system for 3 distinct distributed systems. These included a 3 tiered online auction service, an enterprise business administrative domain, and a web crawler for a commercial search engine. The runtime system includes the necessary infrastructure for monitoring various dynamic and static properties of the distributed system. It is also responsible for the actual process of validation e.g. scheduling the assertion checks, and verifying system correctness based on the $A$ programs.

With the $A$ designed, and the runtimes implemented, we looked to evaluate the efficacy of Model Based Validation on detecting and containing operator mistakes. In all the systems we studied, we discovered that our models and corresponding $A$ programs were sufficient in catching most of the operator mistakes we encountered during operator experiments and fault injection. We also found that simple models can drive the systematic design and implementation of effective validation programs.

In the online auction service, to ensure our mistakes were of comparable complexity and subtlety to real mistakes, we used a combination of mistakes observed from human factors studies in our previous work, those reported by observations in the literature, and those reported by database administrators in the course of a survey. Our efforts resulted in a suite of 11 sample mistakes. We found our model-based validation approach highly effective, catching 10 of the 11 mistakes, none of which could be found using trace and replica-based validation.

In the web crawler for the search engine, we observed operator mistakes during the course of a few maintenance tasks carried out by the development team. The flow model and corresponding $A$ program would have caught 5 of 6 of these mistakes. Moreover, the mistakes were of significance, as they could have serious business consequences.

The last case study of the enterprise network infrastructure was part of a study using Model Based Validation in a larger management tool called Barricade. Here we were able to observe 39 mistakes committed by operators on the enterprise system. Here, the models and associated validation program was able to detect and contain 36 of those mistakes.

While all of the above findings are encouraging results for Model Based Validation,
it is important to notice where and how Model Based Validation has failed. In all of the cases, we see that missed mistakes were due to incomplete models, and thus, incomplete A programs. We stipulated from the beginning that the models we create will necessarily be incomplete, and the goal of Model Based Validation is to as comprehensive as possible. Ideally, further study can be made to see how Model Based Validation can evolve over time, as system engineer gain more knowledge about their system, and new operator mistakes are observed and protected against. It is our hope that providing a domain specific language and high level modeling guidelines that this process can be improved with a higher degree of maintainability than current processes.

To address that above mentioned challenge of incompleteness, we finish this dissertation with a study of leveraging machine learning in assisting the system designer with writing better A programs for Model Based Validation. We first show how a rule learner like JRip fares with the limited operator experiment data that exists. We then propose a method of enhancing the output of the rule learner by augmenting it with expert knowledge provided by the system engineer in a system we call ADIVA. We further explore how attribute manipulation affects the generated rule set. We conclude that through a combination of these techniques, the system engineer can discover important attributes with which to write his A programs - hopefully with the result in less incomplete programs.

We believe that future work in the direction of machine learning would be greatly assisted by larger data sets. The collection of data associated with operator tasks is a difficult and time consuming one. There is very limited publicly available data that is associated with the operation of distributed system maintenance tasks. The techniques described in this work seem to be promising, but more data would serve to strengthen the study.

This body of work as a whole shows that Model Based Validation is a effective technique to assist system engineers in mitigating one of important contributors to failures in Internet services - that of operator error.
Appendices
Appendix A

A Language Grammar

\[
\text{start} = \text{stmts} ;
\]

\[
\text{stmts} = \text{stmt},';' | \text{stmts} \text{stmt} ;
\]

\[
\text{stmt} = \text{element} | \text{binding} | \text{assert} | \text{task} | \text{config} | \text{log} | \text{library} | \text{usestatement} ;
\]

\[
\text{usestatement} = \text{USE ID WITH '}[ idlist '] AS ID ;
\]

\[
\text{library} = \text{libegin } '{' \text{paramlist libassertions }'} ;
\]

\[
\text{libegin} = \text{LIB ID} ;
\]

\[
\text{libassertions} = \text{assert libassertions | assert | config | log} ;
\]

\[
\text{paramlist} = \text{paramlist paramlisthelper | paramlisthelper} ;
\]

\[
\text{paramlisthelper} = \text{VAR type ID '};' | \text{EXT ID '};' | \text{PARAM ID wcfg wlog'}' ;
\]

\[
\text{element} = \text{ELEMENT ID '}{' \text{fblock }'} \} ;
\]
config = CONFIG ID '{' configstmts '}' ;

log = LOG ID '{' logstmts '}' ;

logstmts = logstmts CCONST ';'/ CCONST ';','

configstmts : configstmts cstmt | cstmt ;

cstmt = ':' ID ':' drivername cproplist ;

drivername = CCONST ;

cproplist : cproplist cplstmt | cplstmt ;

cplstmt : ptype ID ASG PATH ',', CCONST ';','

ptype : SINGLE | SET ;

btype : ONLINE | VALIDATION | ALL ;

binding : btype ID BINDS ID '(': CCONST ')' wcfg wlog ;

constlist : constlist ',', CCONST | CCONST ;

idlist : idlist ',', idlisthelper ;

idlisthelper : ID | FCONST | CCONST ;
wcfg : WITH CONFIG ID 'constlist' | /**/ ;

wlog : WITH LOG ID | /**/;

assert : ASSERT ID 'cond' '{ ablock precon }' ELSE '{ sblock }' ;

task : TASK ID '{ NAME ASG CCONST ; ' '}' '{ tblock }'

tblock : tstmt ';' | tblock tstmt ';

tstmt : waitblock | callstmt | assert | varstmt | usestatement | binding ;

varstmt : VAR type ID ASG op ;

callstmt : CALL ID ;

waitblock : WAIT '(' waitfor ')' '{ TIMEOUT ASG ICONST ;' FREQ ASG FCONST ' ; ' } ELSE '{ sblock }' ;

waitfor : cond | CCONST | EXT ID ASG CCONST ;

fblock : construct ';' | type ID ';' | type ID ';' fblock |
| construct ';' fblock ;
construct : '(:type ID:)' ;

cond : NOT cond | '(' cond ')' | cond OR bool_expr
    | cond AND bool_expr | bool_expr ;

bool_expr : op threshcomp op | grpeq
    | property '(: CCONST :)' | collection '(: CCONST :)' 
    | uselistfun | ID| ID PERIOD logcall | ID RANGE logcall ;

uselistfun: property2 PERIOD listfun'(: property2 :)' ;

listfun : SUBSET | SUPERSET | DIFFERENCE ;

grpeq : EQUAL '(: collection :)'
    | EQUAL '(: collection ,,' FCONST')'
    | EQUAL '(: collection ,,' property ')' ;

op : property2 | uselistfun
    | FCONST
    | ICONST
    | CCONST
    | op MATH op
    | '(: op :)' 

ablock : freq delay stat kind ;

freq : FREQ ASG FCONST ;' | /**/
delay : DELAY ASG ICONST ';'; | /**/

kind : GLOBAL ';'; | /**/

stat : ON ';';
    | OFF ';';

sblock : sblock astmt
        | astmt
        |

astmt : property ASG property ';';
        | property ASG constant ';';
        | property ';';
        | voidmethod ';';
        | BREAK ';';

precon : PRECON ASG cond ';';
        | /**/
        | /**/

collection : ID RANGE property
             | ID RANGE configcall

agg : SUM
    | MEAN
    | STDEV
    | MEDIAN

comp : EQ
| NEQ
| GT
| LT
| LEQ
| GEQ

threshcomp : comp '{ FCONST }'
| comp '{ property }' | comp

property2 : '(' type ')' property2
| property
| ID PERIOD configcall | collection
| agg '(' COLLECT '{ collection ')''
| agg '(' property ')

property : ID
| ID PERIOD property

voidmethod : property '(' ')
| collection '('')' | PRINT '(' CCONST ')

configcall : CONFIG '[' CCONST ']' PERIOD ID

logcall : LOG '['ICONST']' PERIOD logfun '('CCONST ')

logfun : WITHOUT
| CONTAINS

constant : ICONST
| FCONST
| CCONST
| TRUE_TOK
| FALSE_TOK

; type : STAT
  | IP
  | GROUPSTAT
  | CON
  | GROUPCON
  | stype

  ;

stype : INT
  | DOUBLE
  | STRING
## Appendix B

### Selected Attributes

<table>
<thead>
<tr>
<th>Command</th>
<th>Machine</th>
<th>Action</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>adduser</td>
<td>machine07</td>
<td>shutdown</td>
<td>wget</td>
</tr>
<tr>
<td>apache</td>
<td>machine07s</td>
<td>slapdodotconf</td>
<td>xml</td>
</tr>
<tr>
<td>apachectl</td>
<td>machine08</td>
<td>slapd</td>
<td>zone</td>
</tr>
<tr>
<td>authconfig</td>
<td>machine08s</td>
<td>slapcat</td>
<td>tier-web-numdeadproc</td>
</tr>
<tr>
<td>chdir</td>
<td>machine10s</td>
<td>slapadd</td>
<td>tier-web-NumNewPort</td>
</tr>
<tr>
<td>chkconfig</td>
<td>machine14</td>
<td>slurpd</td>
<td>tier-web-NumNewDeadPort</td>
</tr>
<tr>
<td>cp</td>
<td>machine14s</td>
<td>spamassassin</td>
<td>tier-web-F.created-nocom</td>
</tr>
<tr>
<td>crontab</td>
<td>machine15</td>
<td>spamd</td>
<td>tier-web-F.modif-noc</td>
</tr>
<tr>
<td>DB_CONFIG</td>
<td>machine15s</td>
<td>spamuser</td>
<td>tier-web-F.del-noc</td>
</tr>
<tr>
<td>db_machine</td>
<td>make</td>
<td>service</td>
<td>tier-web-X.created-c</td>
</tr>
<tr>
<td>depend</td>
<td>Makefile</td>
<td>sshd</td>
<td>tier-web-X.modif-c</td>
</tr>
<tr>
<td>EDITOR</td>
<td>main</td>
<td>start</td>
<td>tier-web-X.del-c</td>
</tr>
<tr>
<td>emacs</td>
<td>master</td>
<td>start</td>
<td>tier-app-numnewproc</td>
</tr>
<tr>
<td>export</td>
<td>mcast_heartbeat</td>
<td>stop</td>
<td>tier-app-numdeadproc</td>
</tr>
<tr>
<td>ftp</td>
<td>mountd</td>
<td>startup</td>
<td>tier-app-NumNewPort</td>
</tr>
<tr>
<td>httpd</td>
<td>mv</td>
<td>tar</td>
<td>tier-app-NumNewDeadPort</td>
</tr>
<tr>
<td>httpdodotconf</td>
<td>myopdotcnf</td>
<td>tomcat</td>
<td>tier-app-F.created-nocom</td>
</tr>
<tr>
<td>inetd</td>
<td>mysql</td>
<td>transport</td>
<td>tier-app-F.modif-noc</td>
</tr>
<tr>
<td>install</td>
<td>mysqld</td>
<td>trillian01</td>
<td>tier-app-F.del-noc</td>
</tr>
<tr>
<td>kill</td>
<td>nano</td>
<td>trillian02</td>
<td>tier-app-X.created-c</td>
</tr>
<tr>
<td>killall</td>
<td>named</td>
<td>trillian03</td>
<td>tier-app-X.del-c</td>
</tr>
<tr>
<td>kinit</td>
<td>netstat</td>
<td>trillian04</td>
<td>tier-db-numdeadproc</td>
</tr>
<tr>
<td>ldapodotconf</td>
<td>nssd</td>
<td>trillian05</td>
<td>tier-db-NumNewPort</td>
</tr>
<tr>
<td>ldapadd</td>
<td>nsswitch</td>
<td>trillian06</td>
<td>tier-db-F.del-noc</td>
</tr>
<tr>
<td>ldapsearch</td>
<td>nscd</td>
<td>trillian07</td>
<td>tier-db-X.created-c</td>
</tr>
<tr>
<td>ldif</td>
<td>perl</td>
<td>trillian08</td>
<td>tier-db-X.del-c</td>
</tr>
<tr>
<td>MACHINE</td>
<td>pico</td>
<td>trillian09</td>
<td>etcpasswd-mod-nocmd</td>
</tr>
<tr>
<td>machine00</td>
<td>postfix</td>
<td>trillian10</td>
<td>etcpasswd-modified-cmd</td>
</tr>
<tr>
<td>machine00s</td>
<td>pkill</td>
<td>trillian11</td>
<td>httpd.conf-created</td>
</tr>
<tr>
<td>machine01</td>
<td>rm</td>
<td>trillian12</td>
<td>httpd.conf-deleted</td>
</tr>
<tr>
<td>machine01s</td>
<td>rpm</td>
<td>trillian13</td>
<td>mcast-created</td>
</tr>
<tr>
<td>machine04</td>
<td>scp</td>
<td>unzip</td>
<td>mcast-modified</td>
</tr>
<tr>
<td>machine04s</td>
<td>server</td>
<td>useradd</td>
<td>mcast-deleted</td>
</tr>
<tr>
<td>machine05</td>
<td>sendmail</td>
<td>vi</td>
<td>mysqlpassword</td>
</tr>
<tr>
<td>machine05s</td>
<td>serveropdotxml</td>
<td>virus</td>
<td></td>
</tr>
<tr>
<td>machine06</td>
<td>sftp</td>
<td>web</td>
<td></td>
</tr>
<tr>
<td>machine06s</td>
<td>sftpsserver</td>
<td>webodotxml</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C

Example A Program

1  element CPU {
2      stat utilization;
3      stat idleness;
4  }
5
6  element Net {
7      stat inPkts;
8      stat outPkts;
9      stat inBytes;
10     stat outBytes;
11  }
12
13  element Memory {
14      stat used;
15      stat free;
16  }
17
18  element CPUGroup {
19      stat utilization;
20      stat idleness;
21  }
22
23  element NetGroup {
24      stat inPkts;
25      stat outPkts;
26      stat inBytes;
27      stat outBytes;
28  }
29
30  element MemoryGroup {
31      stat used;
32      stat free;
33  }
34
35  element LoadBalancer{
36      (IP address);
37      CPU cpu;
38      Memory memory;
39      Network net;
40  }
41
42  element WebServerGroup{
43      (IP address);
44      CPUGroup cpu;
45      MemoryGroup memory;
NetGroup net;
}
element AppServerGroup{
  (IP address);
  CPUGroup cpu;
  MemoryGroup memory;
  NetGroup net;
}

config lvs{
: lvsadm: "ipvsadm.pl"
  single wname = /root/workers/worker/name, "",
  single port = /root/server/port, "",
  single sched = /root/server/sched, "",
  single wport = /root/workers/worker/port, "",
  single forward = /root/workers/worker/forward, "",
  single weight = /root/workers/worker/weight, "",
}

config WS_Apache{
: httpdconf: "httpdconf.pl"
  single docroot = /root/DocumentRoot, "",
  single port = /root/Port, "",
  single jkfile = /root/JkWorkersFile, "",
  single jklog = /root/JkLogFile, "",
  set jkmount = /root/JkMount, "",
: workerprops: "workersprop.pl"
  set wlist = /root/worker-list, "",
  set type = /root/workers/worker/[type!="lb"]/type
    , "",
  set port = /root/workers/worker/port, "",
  single lbfactor = /root/workers/worker/lbfactor, "",
  set balanced_workers = /root/workers/worker
    balanced_workers, "",
  set host = /root/workers/worker/host, "",
: mcasthb: "txt2xml.pl"
  single interface = /root/interface, "",
  single address = /root/address, "",
  single port = /root/port, "",
}

config AS_Tomcat{
: serverxml: "xml2xml.pl"
  single jvmroute = /Server/Service/Engine/@jvmRoute
    , "",
  single port = /Server/Service/Connector[@className="
    org.apache.coyote.tomcat4.CoyoteConnector"]/@port,
    "",
  single address = /Server/Service/Engine/Host/Listener
    [@className="org.apache.catalina.McastHeartbeat"]/
    @address, "",
  single mcastport = /Server/Service/Engine/Host/
    Listener[@className="org.apache.catalina.
    McastHeartbeat"]/@port, "";
single hbinterval = /Server/Service/Engine/Host/
    Listener[@className="org.apache.catalina.
    McastHeartbeat"]/@heartbeatInterval, "";
single hbttl = /Server/Service/Engine/Host/Listener[
    @className="org.apache.catalina.McastHeartbeat"]/
    @heartbeatTTL, "";

ws_all::WebServerGroup("skull.*") with config WS_Apache("/scratch/
    httpd/conf/httpd.conf","/scratch/httpd/conf/workers.properties",
    "/scratch/httpd/conf/mcast_heartbeat_db.conf");
as_all::AppServerGroup("skull.*") with config AS_Tomcat("/scratch/
    tomcat/conf/server.xml");
lbalancer::LoadBalancer("skull04s.rutgers.edu") with config lvs("");
assert wktype(ws_all..config["workerprops"].type == "ajp13"){
  on;
  global;
} else{

}

assert wkport(ws_all..config["workerprops"].port == "8009" &&
  as_all..config["serverxml"].port == "8009"){
  on;
  global;
} else{

}

assert workers(ws_all..config["workerprops"].host == ws_all.
  config["workerprops"].balanced_workers){
  on;
  global;
} else{

}
Appendix D
Handwritten Rules

D.1 Add a Webserver

httpd=1 AND (httpd.conf-mod=1 OR httpd.conf-created=1)
AND tier-web-numnewport=2 AND (tier-web-numnewproc >= 6
AND tier-web-numnewproc < 8) tier-web-F.created-nocom=460+/-1
AND etcpasswd-mod-nocmd=1 AND httpd.conf-created=1
AND mcast-created=1 AND [restofstate=0]

D.2 Migrate Database Server

mysql=1 AND mysqld=1 AND configure=1 AND make=1 AND apachectl=1
AND (my.cnf-mod=1 OR my.cnf-created=1) AND tier-db-numnewports=1
AND tier-db-numnewproc=1 AND web.xml-mod=1 AND tier-app-numnewproc=1
AND tier-app-numdeadproc=1 AND tier-app-numnewport=0
AND tier-app-numdeadport=0 AND mysqlpasswd=1 AND etcpasswd-mod
AND [restofstate=0]

D.3 Add an Application Server

tomcat=1 AND (tier-web-numnewproc>=6 AND tier-web-numnewproc <8)
AND mcast=1 AND mcast-modified=1 AND tier-app-numnewport=1
AND tier-app-numnewproc=2 AND httpd.conf-mod=2

D.4 Add a Mail Server

tier-mail-numnewproc=2 AND tier-mail-numdeadproc=1
AND tier-mail-X.modif-c AND tier-dns-numnewproc=1
AND tier-dns-numdeadproc=1 AND tier-dns-X.modif-c
AND etcpasswd-mod-nocmd=1
AND smtpd_recipient_restrictions=permit_mynetworks
AND relay_domains=rutgers.hal AND mynetworks=192.168.200.0/24
AND main.cf-created=1 AND master.cf-created=1

D.5 Add a LDAP Server

D.6 Migrate LDAP Server
References


