Model-Based Validation for Internet Services

F. Oliveira, A. Tjang, K. Nagaraja*, R. Bianchini, R. P. Martin, T. D. Nguyen
{*fabiool, atjang, knagaraj, ricardob, rmartin, tdnguyen*}*@cs.rutgers.edu*
Rutgers Department of Computer Science
Technical Report DCS-TR-601 May, 2006
Department of Computer Science, Rutgers University, Piscataway, NJ 08854

Abstract. *Operator mistakes have been identified as a significant source of unavailability in Internet services. In our previous work, we proposed operator action* validation *as a framework for detecting mistakes while hiding them from the service and its users. Unfortunately, previous validation strategies have limitations: they require known instances of correct behavior for comparison and they fail to detect latent mistakes, i.e. those that do not lead to unexpected behaviors during the validation process. In this paper, we propose a novel validation strategy, called* model-based validation*, that addresses these limitations and complements the other strategies. Model-based validation introduces a new language for service engineers to write assertions about expected behaviors, proper configurations, and proper structural characteristics, and an associated runtime system, which executes the assertions and monitors the service's execution. Our evaluation demonstrates that model-based validation is highly effective at detecting and hiding both activated and latent mistakes.*

1 Introduction

An ever increasing number of users now depend on Internet services, such as search engines, e-mail, workgroup calendars, and music jukeboxes for their work and leisure. Increasingly, these services are comprised of complex conglomerates of distributed hardware and software components, including load balancers, authenticators, loggers, databases, and multiple types of servers. Ensuring high availability for these services is a challenging task [9, 18, 19].

Our work seeks to alleviate one source of service failures: *operator mistakes*. Several studies have shown that mistakes are a significant source of unavailability [8, 9, 15, 17, 18, 19]. For example, a study of three commercial services showed that mistakes were responsible for 19-36% of the failures, and, for two of the services, were the dominant source of failures and the largest contributor to time to repair [18]. More recently, we found that mistakes are responsible for a large fraction of the problems in database administration [17]. An older study of Tandem systems also found that mistakes were a dominant reason for outages [8].

In our previous work, we proposed operator action *validation* as a framework for detecting mistakes while hiding them from the service and its users [16, 17]. The framework creates an isolated extension of the online service, in which operator actions are performed and later validated. Before the operator acts on a server, the server is moved to this extension. After the operator activity is completed, the server is exercised with a previously collected trace or with a replica of the current workload of a functionally equivalent online server. The correctness of the operator's actions is validated by comparing the replies of the server undergoing validation with those from the trace or from the online server. If validation succeeds, the system moves the server back online. If it fails, it alerts the operator.

Despite the ability of the above validation strategies to detect and hide a large class of mistakes, they have a few important limitations: (1) they require known instances of correct behavior for comparison, i.e., a valid trace or working replica; (2) they provide little guidance in pinpointing mistakes, since they simply detect the existence of one or more mistakes; and (3) they fail to detect latent mistakes, i.e., those that do not produce unexpected behaviors during the validation process.

In this paper, we propose a novel validation strategy, called *model-based validation*, that addresses these limitations and complements the other strategies. Rather than relying on instances of correct behavior, model-based validation relies on human-created models of correct behavior, and proper configuration and structure. These models are defined by service engineers using a new assertion-based language, called *A* , that is specifically tailored to dealing with operator tasks. The behaviors, configurations, and structure are validated by the language's runtime system, which executes the assertions and monitors the service.

The goal of *A* is to express the correct states and behaviors of real systems. However, the state space of real

*K. Nagaraja is now at NEC Laboratories America, Inc., Princeton, NJ 08540. His email address at NEC is kiran@nec-labs.com.

systems is too complex to specify or model completely. Thus, rather than specifying all correct states and behaviors, the *A* programmer needs only to specify assertions that can check the correctness of operator actions before they are exposed. For example, to verify the correctness of changes to a service's front-end load balancer, a simple *A* program could assert that the resource utilization at the back-end nodes should always be similar (within a small percentage of each other). After the operator actions on the load balancer are performed, the runtime system can check the assertions before any mistakes are exposed.

While modeling performance captures dynamic behaviors, we have found that many mistakes result in improper static configurations. Many mistakes resulting in security and latent performance faults fit this pattern. In fact, some security mistakes can remain latent for long periods. *A* and its runtime system have specific constructs that represent configuration state (e.g., a start-up file) as well as audit state (e.g., a log). Using these constructs, model-based validation can, for example, immediately detect that the operator assigned an incorrect password to the database subsystem of an e-commerce service. We found that validating both static and dynamic properties was critical for completeness; that is, both are needed to cover a wide range of mistakes.

To demonstrate and evaluate our approach, we have prototyped a model-based validation framework for an online auction service. Our prototype includes *A* , its runtime system, and a set of *A* programs. We also developed a set of representative mistakes for these tasks. The mistakes are realistic, as we derived them from previous works' observations of real operators. We were also careful to choose mistakes that are plausible yet not easily anticipated, and have significant impact on the system.

We found that the auction service is characterizable as a relatively small set of high-level models, which were straightforward to describe in *A* programs. Our models thus do not rely on obscure, detail-oriented constants. Rather, they use simple equalities and inequalities to describe the necessary changes caused by operations tasks. We also found that our operator tasks were easy to encode in the language as well.

Most importantly, we found that our approach caught 10 out of the 11 mistakes we injected into the operator tasks, and all of these would have been missed using previous validation approaches. Model-based validation would also have detected all 28 (out of 42) previously observed mistake instances that the other validation approaches were able to detect [16]. This is a very encouraging result, because it shows that a high-level understanding of the system, as encoded in our language, will catch a large fraction of unanticipated mistakes.

In summary, our contributions include:

- The proposal and prototyping of model-based validation, including *A* and its runtime system;
- The evaluation of model-based validation through an extensive set of experiments, showing that it catches a large fraction of mistakes; and
- Showing that simple, high-level models of a service, as opposed to low-level detailed ones, are sufficient to catch mistakes.

The rest of the paper is organized as follows. First, Section 2 overviews the background and related work. Next, Section 3 describes a few example models. Section 4 describes *A* , whereas Section 5 describes its runtime system. Section 6 gives an operational overview of how to apply model-based validation. Section 7 describes our prototype and evaluates model-based validation using extensive experimentation. Section 8 discusses the issues that remain open. Finally, Section 9 concludes the paper.

2 Background and Related Work

In this section we first present background on our validation approach. We then place our work in context by describing the related work.

2.1 Validation Background

We initially proposed validation of operator actions in [16]. The core idea of validation is to check the correctness of operator actions under realistic workloads in an isolated validation environment. Mistakes can then be caught before becoming visible to the rest of the system and users. To achieve realism and isolation at the same time, the validation environment is hosted in an extension of the online system itself. We divide the hardware components (server nodes) into two logical slices: an online slice that hosts the online components and a validation slice where components can be operated on and validated before being re-integrated into the online slice. Components in the validation slice are called *masked*. Components are logically moved between slices using layer 2 and 3 virtual networking.

The validation process proceeds as follows. Before operating on a component, the operator uses a script to move the server node into the validation slice. After completing his/her task, the operator uses another script to place a workload on the masked server. The workload can be a previously collected trace (trace-based validation) or a replica of the current workload of a functionally equivalent online server (replica-based validation). Validation involves comparing the replies of the masked server with those in the trace or those of the online server. If the

replies match (according to content-similarity and performance criteria) during a period of time, the framework considers the operator actions to be validated and logically moves the masked server back online without any changes to its configuration. If validation fails, the system alerts the operator.

In [17], we revisited trace-based and replica-based validation for database servers, whereas Tan *et al.* [23] revisited replica-based validation for file servers. We also proposed a primitive version of model-based validation in [17]. The idea was to have the administrator describe his/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 introducing a language and runtime system to enable the application of model-based validation beyond database servers.

2.2 Related Work

The related works fall into two categories: formal modeling and performance verification. In general, our approach is more domain-specific and favors practicality and programmability over the provability properties favored by formalisms. In addition, model-based validation complements many verification strategies as it relies on the designer to provide the verification logic instead of deriving it from running systems.

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 [26]. 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 [24]) model systems as a set of valid states separated by transition functions, *A* does not seek completeness; we do not try to model all possible valid states. Because the program state in *A* represents a running system, not a theoretical model, it is faced with observability issues that do not exist in formal languages.

Finally, assertion languages have been extensively explored for software debugging. For example, [7] imposes pre-conditions and post-conditions on software components. *A* supports a similar pre/post-conditions paradigm, but targets operator mistakes rather than software bugs.

Performance verification. Like model-based validation, other systems have combined assertion languages with dynamic assertion checking. Both PSpec [20] and Pip [21] used assertion checking for performance debugging. As Pip focuses on distributed systems, it also seeks to verify their communication structures. However, because they are concerned with dynamic, run time problems as opposed to detecting operator mistakes, these systems did not consider some important static and structural issues, such as improper system configurations and latent security problems. Operator tasks are first-class concepts in model-based validation in general and in our assertion language in particular, allowing us to detect these types of problems and relate them to specific tasks. Besides its focus on operator mistakes, model-based validation differs from other assertion-checking efforts [6] in that our assertions are external to the component being validated.

In contrast, the focus of [1] was performance-debugging systems with as little application-specific knowledge as possible by viewing the system as a black box and examining bottlenecks. With a similar philosophy, Pinpoint [5] and Magpie [3] attempt to infer correct system behaviors from actual executions, without application-specific knowledge. Our work differs from these efforts in that model-based validation asks programmers to explicitly declare correct behavior, which is critical when no previous samples exist—a common problem in the face of operator actions.

Finally, Service Level Agreements (SLAs) are another method of specifying required performance, e.g. [2]. For example, [11] 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.

3 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 these 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 works in the reverse direction: we create models that describe correct behavior in an existing system. 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

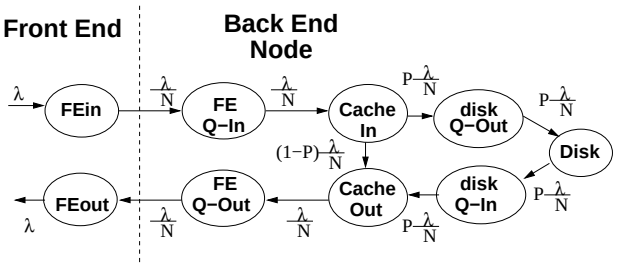

Figure 1: *A partial data flow graph for a cluster of Web servers with 1 front-end load balancer and 1 back-end Web server; an entire graph would extend this basic architecture across the set of back-end servers.*

and realized using *A* .

3.1 Data Flow Model

Data flow models characterize a service as a graph in which nodes are computations and edges are message flows. Figure 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 1 illustrates the following example assertion: if the input rate to the front-end device has mean λ, 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, as they are central concepts in previous distributed systems [5, 21, 12]. 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 [16], which suggests that this model is a very 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.

3.2 Access Control Model

Our model of access control uses an *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.

3.3 Hierarchical Component Model

This model captures the service component to subcomponent 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 on the source component "working" given the target is "working". The exact meaning of dependency and working are left as specifics for the *A* programmer.

Although this model is conceptually simple and its DAG structure maps nicely to fault-tree analysis, we found it to be the most difficult to realize in practice. The problem is that "working" can often only be defined in an ad hoc manner. Also, inter-component dependencies are often much more obscure than inter-component connectivity.

4 The *A* Assertion Language

In this section we describe the *A* language. Because model-based validation encompasses more than just the language, space constraints allow us only a brief introduction. We first give an overall picture of the language, and then describe its basic constructs using the example code in Figure 2.

The approach *A* takes centers around system engineers writing *assertions* about *elements*. An element is an object that represents a service component, such as a Web server. Each element comprises a number of fields representing the state of the component, e.g. the number of requests seen in the last minute. An element is thus analogous to a C `struct`. But unlike a normal programming language object, the values within elements come not from the programmer, but are defined by the monitored state of the component. Elements are *bound* to specific components, e.g. the Web server with IP address 192.169.10.2. Mapping the state of the real system to values in the *A* program is the job of the runtime system, which is described in Section 5.

In addition to element fields that describe performance, *A* has specific constructs to describe the static input configuration and information output of service components.

Returning to our Web server example, these two constructs could easily describe a configuration file and an output log. Thus, an assertion might check to see if the value of a password in a configuration file is not empty.

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

A critical difficulty with traditional modeling languages with regards to computer systems is the description of how system state should change with time. *A* has two novel mechanisms to describe system evolution over time. The first is the *task* construct, which models a human-machine interaction. Tasks are sets of assertions separated by *wait* statements, which allow an *A* program to wait for some expected operator action. This allows a programmer to capture how the set of valid assertions should change as a human operates on the system. The second temporal construct is the *stat* type, which captures temporally sampled values, e.g. the load average over time.

Finally, to provide a method of abstraction, a program can specify a *library*. Libraries are ways to abstract sets of assertions, and thus allow the separation of concerns between different programmers. In the rest of this section, we describe each of these constructs in turn.

In file: connected.lib

```
1:  config TomcatCfg{ ...
7:      :webxml:"xml2xml.pl"
8:          single env-entry-value =
              /web-app[een="jdbc"]/ev, "";
9:      :netint:  "netinterfaces.pl"
10:         single eth1hostname =
              /root/if[dev="eth1"]/hostname,"";
11:  } ...
12:  log mySQLLog{ "/pathto/mysql/my.err"; ...}
```

In file: add_app_server.a:

```
20:  #include "connected.lib";
26:  task add_app_server
      {name="Add an application server";}
27:  {
28:     online as_all::ApplicationServerGroup(IP=".*")
        with config TomcatCfg("/path/to/web.xml")
        with log TomcatLog;
29:     validation db::DBServer(IP="dbserver.domain.tld")
        with config mySQLCfg
        with log mySQLLog;
30:     wait("Begin task"){ timeout = 30000;}
        else{ break; }; ...
31:     wait("Begin Validation"){timeout = 30000; }
        else{ break; };
32:     use connected with[as_all, db] as
             add_AS_connected;
33:     assert balanced
             (EQUAL(COLLECT(as_all..cpu.util))){
34:            on; taskonly;
35:       } else{//print "App servers load unbalanced"}
36:     assert overload (as_all..cpu.util < 0.80){
37:            on; taskonly;
38:       }else{// print "An app server is overloaded"}
39:     wait("End Validation"){ timeout = 30000; }
        else{ break; };
40:  }
```

Figure 2: *An example A program comprised of a library (connected.lib) and a task specific program (add_app_server.a). Keywords are shown in bold.*

4.1 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. Unlike standard programming languages such as C or Java, *A* does not allow programmers to define element types; rather, an *A* programmer is restricted to a set of predefined types corresponding to the types of service components monitored and exported by the runtime system.

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, e.g. the CPU utilization sampled every second over the last 1 minute. The window size and sampling frequency of a stat object is dynamically configurable.

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. In a comparison, each stat object is aggregated into a single value using a specified aggregation operator. Equality can be checked for arbitrary numbers of stat objects. Three or fewer stat objects are considered equal if they are within a specified threshold of their median. Four or more stats are considered equal if there are no outliers according to the Box and Jenkins outlier model. Finally, inequality relations ($<, >$) are computed by first computing equality; the less or greater than relation is checked only if the stat objects are not equal according to our equality operator.

Element binding. Each element is instantiated and named by a variable via a binding statement, which creates and maps the element to the monitored state of the appropriate service component according to a binding expression. The binding expression typically specifies some property that uniquely identifies the service component such as the IP address of a node. Line 29 in Figure 2 gives an example of such an element binding, where the variable `db` names an element of type `DBServer` that is bound to a database server with IP address `dbserver.domain.tld`. If a binding operation

fails, the entire *A* program is halted.

Binding statements can also instantiate and name a set of identically typed elements. Such a binding leads to an aggregate element of an aggregate element type. An aggregate element type has the same fields as those of the member elements' type, but each field is a set of values derived from the set of member elements of the aggregate. Line 28 shows an example of such an aggregate binding, where `as_all` is bound to all application servers in the online slice; the aggregate type `ApplicationServerGroup` is the critical part of the binding expression in this case, specifying that the programmer only wants `as_all` to be bound to application servers. Aggregate bindings are motivated by the fact that components in Internet services are often replicated for availability and performance. The provision of aggregate elements allows *A* programmers to easily specify assertions about all components of a replicated group. Line 33, which asserts that the CPU utilization of all Web server nodes should be approximately equal, is an example of such an assertion. (Fields in an aggregate object are accessed using the `..` operator.)

When binding an aggregate element, a programmer must specify where the runtime system should search for matching components: in the online slice, in the validation slice, or in both slices. Each aggregate binding statement thus begins with a keyword to define the scope for the bindings to be affected: `online`, `validation`, or `all`.

Configuration and log types. Each element may have an attachment of 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. *A* allows programmers to define new configuration types, but not new elements. This is because the definition of configuration types only involves parsing configuration files, rather than requiring runtime monitors to be implemented. The definition of a configuration type involves the specification of a set of configuration files (Line 7) or program outputs (Line 9) 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.

Each configuration type can specify multiple sources for information. Due to the vastly different formats that services and devices use, we assume that system engineers will create drivers in the runtime system to translate configuration information into the XML format. Configuration types are then defined using XPath queries to obtain the relevant parameter values from the XML output of these drivers.

Lines 1-11 show the definition of a configuration type called `TomcatCfg`. This definition contains specifications for two different configuration sources: the `web.xml` file and the output of the `netinterfaces.pl` driver. The actual path of the configuration files to be parsed may be left until the instantiation and binding of a configuration object of that type, as shown in line 28. However, these sources are named in the definition for ease of referral later in assertions. Within each file specification, there are lists of properties that are contained within the files. Line 8 shows the extraction of the `jdbc` property of a Tomcat installation from the `web.xml` file. The right hand side of the assignment in this line is the XPath location of the value of that particular property. The parameters in web.xml are parsed by the driver `xml2xml.pl`.

Definitions of log types include paths to locations of log files. Similar to stat objects, each log object is a tail of a stream of information being written to some log file. Currently there is only one operation allowed on log files: `contains`. This provides, at a very minimal level, "grep"-like functionality. Log files intrinsically have a temporal component, so while the simple `contains` function might be sufficient for most uses, allowing programmers to specify intervals on which to search for information in the log files could be useful.

Configuration Aggregates. It is often useful to name an aggregate set of configuration parameters. For example, the `workers.properties` file in an Apache configuration may contain multiple workers. If the programmer specifies an XPath that may refer to all such workers, he should precede the declaration with the keyword `set`. Standard set operators apply when making comparisons between aggregate configuration parameters.

4.2 Assertions

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 that implements a single method that returns a boolean value.

Expression. An expression returns a boolean value. If the statement evaluates to `true`, the system is correct. If the expression evaluates to `false`, the code in the action block (the `else` clause) is executed. An expression can name another assertion in it, which returns a boolean. When an expression is evaluated, all the named assertions are first evaluated.

Control Block. The programmer can optionally specify values for the following parameters to control the scheduling of an assertion:

frequency: The periodicity with which the assertion should be checked in seconds.
delay: The time, in milliseconds, that the run-

time waits before checking the assertion upon program startup.
status: Describes whether the assertion should be `on` or `off` upon program startup.
type: Indicates whether the assertion is a standalone assertion ("global"), or part of a task, ("taskonly").

Action Block. We allow arbitrary Java code to be inserted into the runtime via the action block. However, all of our current programs simply print to standard output. 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. For example:

```
assert connected(A_talks_to_B && B_talks_to_A){ ...}
assert A_talks_to_B(A_pings_B && A_http_get_B){ ... }
```

Here, A and B are connected if both sides of the connection are working, and communication is defined as the success of a ping and HTTP GET. If any of the sub-assertions fail, `connected` would fail. For the purposes of scheduling, sub-assertions will run as fast as its fastest parent.

Aggregates. Relations in the expression part of an assertion can accept aggregate element types as arguments. 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. A field of an aggregate object can also be compared to a value as shown in line 36, in which case the relation has to hold true for all values in that field for the relation to hold for the aggregate field; the expression in line 36 only returns TRUE when all utilizations are less than 80%. Reduction operators such as `SUM` and `MEAN` can also be used to reduce the set of values in an aggregate element's field to a single value.

4.3 Libraries

Libraries support abstraction over different element instances as well as hide detailed sets of assertions. Libraries can be organized by components, subcomponents, or by goal. A library definition is not shown in Figure 2, but line 32 shows the `use` of a library. A library must declare all elements and config and log objects referenced inside it as parameters. To use a library, a task or program first includes it (line 20), and then issues a `use` statement with the required arguments (line 32). The `use` statement binds the variable names in the library to the arguments and then activates all assertions in the library (unless their default `status` is off).

4.4 Tasks

Tasks represent human-system interactions. One can think of tasks as sets of assertions grouped together with intervening `wait` statements. Tasks can be stateful and compare system properties throughout the course of task execution. The structures found in tasks are the following:

Taskonly assertions. These have a lifetime that begins at the definition of the assertion and ends at some predefined time at the end of task completion (lines 33-38). Task assertions may be derived from library usage.

Wait statements. These tell the runtime system to wait on particular operator events, explicitly indicated by the operator (lines 30-31).

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. In the task in the code listing, notice the waits on operator actions. These waits are interpreted by the runtime and the strings are displayed as buttons to the operator to be activated when that particular subaction is completed. 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. If the statement within the else block is a `break`, the operator task is aborted. 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).

5 The *A* Runtime System

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

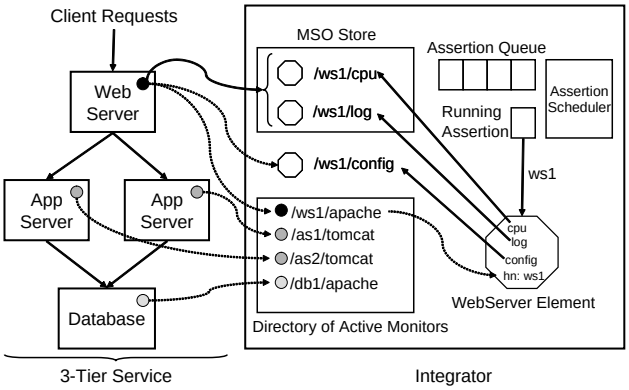

Figure 3: *Architecture of the current A runtime system. Shaded circles inside the service components (Web server, application servers, database server) represent monitors. Shaded circles in the directory of active monitors represent monitor callback objects. Hexagons inside the MSO Store represent monitoring stream objects (MSOs). The dynamic streaming of monitoring data to the receiving MSOs is only shown for one monitor for clarity.*

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 3 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.

In the following subsections, we will describe the monitors, the storage of monitoring data on the Integrator, the binding of *A* elements to observations, and assertion scheduling.

5.1 Monitors

Monitors are software components designed and implemented outside the scope of the *A* system. However, in order 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.

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

5.2 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` filesystem, and i-node information of all directory entries of any given filesystem subtree.

5.3 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.

5.4 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 calculed 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 from the queue when the validation of the task has been completed, regardless of the validation result.

6 Using Model-based Validation

In this section, we give a brief operational overview of model-based validation, walking the reader through the steps for deployment and execution.

Deploying the model-based validation infrastructure consists of two major steps. First, component classes and monitors need to be built. Following the appropriate programming interface, Java classes for the components that are of interest need to be created and included in the Integrator. Monitors for the relevant properties of components need to be implemented according to the interface defined by the Integrator. Our current prototype includes 4 classes of components — LoadBalancer, WebServer, AppServer, and DBServer —, as well as a generic monitor for CPU, memory, network, and configuration files.

The second step for deploying the infrastructure is to create models and programs. A system engineer develops models to capture the correct behavior of the system and expresses them as *A* libraries. Just as importantly, a person who understands the tasks to be performed by operators encodes them as *A* programs using the appropriate libraries. The *A* programs are then compiled, which results in a Java executable. When the Integrator is started up, a menu shows all defined operator tasks.

Once the infrastructure is deployed, model-based validation comes into action everytime the operator performs a task for which an *A* program has been written. Each task requires at least one input from the operator specifying the machine to be operated on. In order for model-based validation to work properly, a subset of components must be brought into the validation slice so that the interaction between adjacent tiers and load distribution can be checked. In the multi-tier system we modeled, during our experiments we needed at most 4 components in the validation slice. Proxies are used for each tier whose components are not needed in the validation slice. The components in the validation slice are exercised based on requests from a trace, and the operator is responsible for specifying how much load (requests/s) must be imposed and for how long validation should run.

As the operator completes sub-tasks, i.e., the intervals defined by `wait` statements, the operator watches for assertion firings. If an assertion fires, the operator must pause or abort the task and try it again. Only after a task is completed with no assertion firings are the components moved back into the online slice. Note that moving the components from one slice to the other is done automatically and does not require configuration changes.

7 Evaluation

We now turn to evaluating our validation approach. 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.

7.1 Experimental Setup

Our evaluation is performed in the context of an online auction service modeled after EBay [22]. 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.

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 approximately 50% of the maximum achievable throughput.

7.2 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. 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.

Library	Number of Assertions
load balancer to Web servers	4
Web servers to application servers	5
application servers to database	5
Web servers	8
application servers	4
database	6
LVS policy (balanced)	2
Apache policy	6
Tomcat policy	1
dynamic balanced utilization	2
dynamic capacity	2
security	4

Table 1: *Summary view of our A program.*

We then used the above models together with results from previous works exploring the nature of operator mistakes [4, 10, 14, 16, 17, 18, 25] to guide the writing of 12 *A* libraries (Table 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 such as the sum of the allowed outstanding JDBC connections on the application servers should be less than or equal to 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 al-

ways 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.

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 7.3) to evaluate our approach. Once we wrote the libraries, it was quite easy to write the task-specific programs. 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.

7.3 Mistake-injection Experiments

Table 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 *A* programs mentioned above.

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 [16] (observed). Some of the mistakes were reported in a survey of database administrators [17] (survey). All other mistakes were synthetically generated but motivated by previous work exploring the nature of operator mistakes [10, 18, 16, 14, 4, 25, 17] (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 6 using the programs described in Section 7.2. 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 2. (We also intend to run mistake-injection experiments against *A* programs written by a number of graduate student groups as part of a class project. Unfortunately, these student groups were unable to complete their work in time for this submission. We hope to have results from this effort if this paper is accepted.) In all cases, if a mistake was detected, the assertions that fired gave very strong clues for identifying the actual mistake. While we cannot currently quantify the effectiveness of model-based validation for pinpointing mistakes, our experience indicates that this is an important advantage of model-based validation that should be explored in the future.

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 [13] 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 dis-

Category	Mistake	Impact	Task	Caught?
Connectivity	"LVS ARP problem": Web server not configured to ignore ARP requests. (**synthetic but well-known** [13])	Web server might respond to ARP requests originated by clients, bypassing the front-end load balancer.	Web server addition	Y
	Web server not compiled with support for the membership protocol. (**synthetic** [18])	Affected Web server will forward requests to application servers taken off-line.	Web server addition	N
	TTL of membership heartbeat messages is misconfigured in one application server. (**synthetic** [18, 10])	If the TTL is too high, the Web servers will not stop sending requests to an application server taken off-line.	Application server addition	Y
	Misconfiguration of one pair of Web server and application server: wrong yet matching port numbers. (**synthetic** [14])	All Web servers but the affected one will not be able to contact the misconfigured application server.	Web server addition	Y
Capacity	The number of connections handled by the database is exceeded. (**synthetic** [4])	It causes the system not to work at the maximum capacity.	Application server addition	Y
	Wrong front-end load balancer policy is activated. (**synthetic** [25])	Undesired/inappropriate distribution of load across Web servers.	Load balancer addition	Y
	Web server load balancer misconfigured. (**synthetic** [25])	Undesired/inappropriate distribution of load across application servers.	Web server addition	Y
	DBMS performance parameters configured sub-optimally. (**survey** [17])	Service overall performance might be jeopardized.	Database creation	Y
Security	Database administrator account not assigned a password. (**observed** [16])	Serious security vulnerability.	Database creation	Y
	Allowing any machine to access the database remotely. (**survey** [17])	Security vulnerability and possibility of data corruption due to unmalicious yet unauthorized data access.	Database creation	Y
	Allowing an ordinary user to grant/revoke privileges to/from other users. (**survey** [17])	Serious security vulnerability.	Database creation	Y

Table 2: *Mistake-injection experiments.*

tributed 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.

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 to the 3rd tier, the cumulative demand for concurrent connections to the database may exceed this setting.

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. Unexpectedly, the 4 assertions implementing our database access control security model also fired. It turns out that the driver converting database access control information into XML was not able to connect to the database; therefore, the corresponding XML file was not generated, causing the assertions to fire.

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 the weights given to all Web servers 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.

Security mistakes. In a previous study [16], 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 [17] mentioned database problems resulting from security vulnerabilities. The security mistakes shown in Table 2 were derived from our previous studies.

During the experiments, our access control model caught all security mistakes we injected.

7.4 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.

8 Open Issues

There are many open issues with our approach. The first stems from the amount of knowledge needed to develop working models and *A* programs. Our results show that general models work quite well, although to translate these into the details needed to make working *A* programs requires knowing some details about specific subsystems. We also found that we did not need too many difficult-to-know, hard constants. Instead, we tried to use simple invariants like "all equal" or inequalities (e.g., the absolute load must be lighter on a node after adding a replica).

We could have used a running *A* program for extensive monitoring and diagnosis of the online system by using global assertions. We instead focused on validating operator tasks, which tends to focus on a few components. In addition, having a large number of assertions can provide enough information to pinpoint problems, but it may result in information overload if many assertions fire.

Finally, we left open the issues of actuation–what to do when an assertion fails. Our current prototype allows

the execution of arbitrary Java code in the `else` block of an assertion, but currently we just print error messages. Our future work will consider taking actions to correct operator mistakes when an assertion fails.

9 Conclusions

In this paper we investigated using models combined with validation as a strategy for identifying and hiding operator mistakes in Internet services. We developed three high-level models to capture the flow, component, and security correctness properties in a service. We next realized these models in a novel language, *A* , and its runtime system, for an online auction service. We found that the models were straightforward to express in our language; only 49 assertion statements encapsulated in 12 libraries were needed to capture the important abstractions of all the models.

Using the language constructs that describe operator tasks, we encoded 4 representative tasks: (1) adding a web server, (2) adding an application server, (3) adding a load balancer, and (4) creating a database. Using the libraries, we found that representing the correctness properties of the auction service during the operator tasks was simple.

To evaluate our approach, we performed mistake injection experiments to observe how well mistakes were caught and hidden from users. 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.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. of SOSP'03*, pages 74–89, New York, NY, USA, 2003. ACM Press.

[2] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Oceano: SLA-based Management of a Computing Utility. In *Proc. of International Symposium on Integrated Network Management*, May 2001.

[3] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie:Real-Time Modelling and Performance-Aware Systems. In *Proceedings of HotOS IX*, May 2003.

[4] R. Barrett, E. Kandogan, P. Maglio, E. Haber, L. Takayama, and M. Prabaker. Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices. In *Proceedings of CSCW'04*, 2004.

[5] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *Proceedings of NSDI'04*, Mar. 2004.

[6] Y. Cheon and G. T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In *Proceedings of SERP '02*, June 2002.

[7] Building bug-free o-o software: An introduction to design by contract. http://archive.eiffel.com/doc/manuals/technology/contract/.

[8] J. Gray. Why do Computers Stop and What Can Be Done About It? In *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems*, Jan. 1986.

[9] J. Gray. Dependability in the Internet Era. Keynote presentation at the 2nd HDCC Workshop, May 2001.

[10] G. W. Herbert. Failure from the Field: Complexity Kills. In *Proceedings of EASY '02*, Oct. 2002.

[11] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1), 2003.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[13] Linux.org. Linux Virtual Server. http://www.linuxvirtualserver.org, 2006.

[14] P. Maglio and E. Kandogan. Error Messages: What's the Problem? *ACM Queue*, 2(8), 2004.

[15] B. Murphy and B. Levidow. Windows 2000 Dependability. Technical Report MSR-TR-2000-56, Microsoft Research, June 2000.

[16] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of OSDI '04*, Dec. 2004.

[17] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Validating Database System Administration. In *Proceedings of the 2006 USENIX Annual Technical Conference*, June 2006.

[18] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet Services Fail, and What Can Be Done About It. In *Proceedings of USITS'03*, Mar. 2003.

[19] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. ROC: Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, UC, Berkeley, Mar. 2002.

[20] S. E. Perl and W. E. Weihl. Performance Assertion Checking. In *Proceedings of SOSP'93*, Dec. 1993.

[21] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proc. NSDI 2006 (to appear)*, May 2006.

[22] Rice University. DynaServer Project. http://www.cs.rice.edu/CS/Systems/DynaServer, 2003.

[23] Y.-L. Tan, T. Wong, J. D. Strunk, and G. R. Ganger. Comparison-based File Server Verification. In *Proceedings of the USENIX Annual Technical Conference*, June 2005.

[24] The z notation. http://vl.fmnet.info/z/.

[25] A. Wool. A Quantitative Study of Firewall Configuration Errors. *IEEE Computer*, 37(6), 2004.

[26] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of OSDI '04*, Dec. 2004.