Model-Based Validation for Internet Services

Andrew Tjang,* Fábio A. Oliveira, Ricardo Bianchini, Richard. P. Martin, Thu D. Nguyen*
Technical Report DCS-TR-601 May, 2006
Updated October 31, 2008
Department of Computer Science, Rutgers University, Piscataway, NJ 08854
{atjang,fabiool,ricardob,rmartin,tdnguyen}@cs.rutgers.edu

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 an approach for detecting mistakes while hiding them from the service and its users. Unfortunately, previous validation strategies have limitations, including the need for known instances of correct behavior for comparison. In this paper, we propose a novel *model-based validation* strategy that addresses these limitations and complements the other strategies. Model-based validation calls for service engineers to define models of Internet services that can be used to differentiate between correct and incorrect configurations and behaviors. These models are then used to guide the specification of assertions that can check the correctness of operator actions before they are exposed. We have implemented a prototype model-based validation system for two services, the Web crawler of a commercial search engine (Ask.com) and an academic yet realistic online auction service. Experimentation with model-based validation in these two systems 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 are becoming dependent on Internet services such as search engines, e-mail, and music jukeboxes for their work and leisure. These services typically comprise complex conglomerates of distributed hardware, software, and databases. Ensuring high availability for these services is a challenging task [9, 19, 20].

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, 18, 19, 20]. For example, a study of three commercial services showed that mistakes were responsible for 19-36% of failures, and, for two of the services, were the dominant source of failures and the largest contributor to time to repair [19]. More recently, we found that mistakes are responsible for a large fraction of the problems in database administration [18]. 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, 18]. 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 service component, the component is moved to this extension. After the operator activity is completed, the component is exercised with a previously collected trace or with a workload dynamically duplicated from that of a functionally equivalent online component. The correctness of the operator's actions is validated by comparing the replies of the component undergoing validation with those from the trace or from the online component. If validation succeeds, the system moves the server back online. If it fails, it alerts the operator.

While the above validation strategies can detect and hide a large class of mistakes, they have three 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 under the validation workload.

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 calls for service engineers (not oper-

*Part of this work was done while Tjang and Nguyen were with Ask.com.

ators)[1] to choose an abstract model (or a small set of models) that can be used to describe the system and differentiate between correct and 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. A failed assertion indicates incorrect behavior and helps the operator pinpoint her mistake(s). The purpose of the model is to ensure a systematic and proactive approach to generating validation assertions, rather than an ad-hoc and reactive approach that may leave many potential mistakes undetected.

To demonstrate and evaluate our approach, we have built a 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 [23]. This second service is smaller in scale than the Ask.com system but allows us to evaluate model-based validation more extensively using mistake injection. This prototype 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 previous validation approaches. Model-based validation would also have detected all 28 previously observed mistake instances that the other validation approaches were able to detect [16].

The above implementations have three main parts as follows. (1) A small set of models that we decided were simple yet allowed clear description of correct vs. incorrect behaviors. We describe the most important model, the flow model, in Section 4. (2) An exploratory language called *A* that is designed to make it easy and convenient to express correctness assertions derived from the models. We briefly describe *A* in Section 5. And, (3) a runtime system that maps monitored states of a service to *A* program objects and runs *A* validation programs. We describe this runtime system in Section 6.

[1] "Service engineers" are the people who design and implement a service, whereas "operators" are those responsible for its day-to-day management.

We present our evaluation of model-based validation in Section 7 and discuss our experience in Section 8.

Our main contributions are:

- Proposing and prototyping model-based validation;
- Applying model-based validation to a commercial service and an academic service, and demonstrating that it catches most operator mistakes in both of them;
- Describing the effort involved in applying model-based validation to the Ask.com crawler. This experience demonstrates that our approach is scalable in practice along all critical dimensions, including service knowledge, service modeling, writing validation programs, and state collection and runtime execution of assertions.

2 Validation Background

The core idea of validation is to verify operator actions under realistic workloads in a realistic but isolated validation environment [16]. Mistakes can then be caught before becoming visible to the users. To achieve realism, the validation environment is hosted by the online system itself. In particular, we divide the server nodes into two 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. 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.

Validation proceeds as follows. Suppose an operator needs to operate on a service component (e.g., to upgrade it to a new software version). Before operating on the component, the operator uses a script to move the server node hosting the component into the validation slice. This effectively takes the node and all software components that it hosts offline. After completing her task, the operator uses another script to place a workload on the masked component. The workload can be a previously collected trace (trace-based validation) or a replica of the current workload of a functionally equivalent online component (replica-based validation). Validation involves comparing the replies of the masked component with those in the trace or those of the online component.

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 moves the hosting server node back online. If validation fails, the system alerts the operator.

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% of database administrators responding to a survey reported that they typically test their actions in environments that are different from their online environments [18]. 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. For example, validation could be applied as the last step in a testing/validation process before exposing an operator action to the online system.

In [18], we revisited trace-based and replica-based validation for database servers, whereas Tan *et al.* [25] revisited replica-based validation for file servers. We also proposed a primitive version of model-based validation in [18]. 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.

3 Related Work

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., [2, 24, 32]. Unfortunately, many tasks cannot be automated, creating the possibility of operator mistakes. Recommendation systems, e.g., [5], attempt to prevent mistakes by guiding the operators' actions. As an extra safety net, previous validation systems [16, 18, 25] hide and detect certain types of mistakes by confining the operators' actions to a sandboxed environment. When mistakes are made, recovery/undo systems [6, 29, 24, 30] can be used to bring the service (or the sandbox) back to a proper state. Finally, monitoring/auditing systems, e.g., [1, 3, 28, 7, 21, 22, 27, 13], 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. As detailed above, model-based validation extends previous validation systems. Model-based validation is also closely related to monitoring/auditing systems in that our assertions can be constantly checked to ensure proper service performance and behavior. For example, PSpec [21], Pip [22], and D^{3}S [13] use assertion checking to help debug software, with the latter two focusing on distributed systems. However, because they are concerned with detecting software bugs as opposed to operator mistakes, these systems did not consider some important static and structural issues, such as improper system configurations and latent security problems.

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 [7] and Magpie [3] attempt to infer correct system behaviors from actual executions, without application-specific knowledge. Our work differs from these efforts in that we ask service engineers to explicitly reason about and declare correct behavior, which is critical when no previous samples exist—a common problem in the face of operator actions. Operator tasks are first-class concepts in model-based validation in general and in our assertion language in particular, allowing us to detect mistakes and relate them to specific tasks.

Along the lines of software testing, a recent related work is ConfErr [11], 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 [17]. 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.

4 Example Models

Flow model. 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

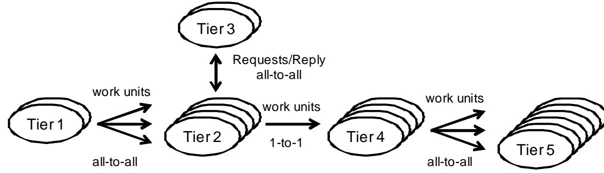

Figure 1: *Flow model for the crawler. Each tier is a replicated set of software components. Each work unit comprises a set of URLs to be downloaded. Work units traverse statically configured flow paths shown by the arrows. Tier 1 is responsible for coarse-grained work scheduling, Tiers 2 and 3 are responsible for host resolution and fine-grained work scheduling, and Tiers 4 and 5 are responsible for downloading and processing 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.*

and the auction service. Figure 1 shows a high-level flow graph for the Ask.com crawler.

Describing an Internet service as a flow of requests is not a new concept, e.g., [7, 22]. However, a flow model immediately brings to mind a number of correctness characteristics that should be validated. First, constructing a flow model dictates that the service engineer must reason about *connectivity*; that is, which components are connected to which other components 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 [16] led to incorrect connectivity.

A second characteristic is *flow preservation*, which specifies that requests can only enter the flow graph at known sources and then leave the system at known sinks. Requests cannot be generated spuriously inside the graph, nor can they disappear without leaving the system through known sinks. Of course, split points, i.e., points where a request may transform into several requests, and merge points must properly be defined as sources and sinks. For example, a split point precedes the flow from Tier 2 to Tier 3 in the Ask.com crawler 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. That is, 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 by a detectable amount.

A fourth characteristic is that flows out of a node to 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, which should then be checked against the context of the entire system for correctness. 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). Examples of stagnation include mistakes that lead to starvation of some requests or deadlock.

Other models. We also used two complementary models: a hierarchical component model and an access control model. The access control model is based on a simple access control matrix but is nevertheless effective because it allows the explicit checking of access configurations. The hierarchical component model specifies the component to sub-component relationships for complex components such as a server. This model aids the writing of assertions to ensure that parts of complex components do not fail silently.

5 The *A* Assertion Language

Our model-based validation programs are written in a new language called *A*. We designed *A* for two reasons. First, we wanted to explore whether a language tailored to validation, operator tasks, and Internet service management would improve expressiveness and readability—we strongly believe that it does. Second, in the future, we plan to explore the use of compiler analysis to analyze and improve validation programs. Within the scope of this study, however, a library written in a more familiar language such as C++ or Java would have served equally well. Thus, in this section, we give only a brief overview of *A*, focusing on general features that we found to be useful for reasoning about service configurations and behaviors and not on specific details of *A* itself. Figure 2 gives a small example *A* program to help make our description more concrete. We refer the interested reader to [26] for more details on *A*.

The fundamental idea behind *A* is to capture and expose the state of a running service as a set of language objects. Programs can then be written to examine this state and validate service configuration and behavior against

```
00:  #include "connected.lib";
01:  task add_app_server {name="Add an Application
Server";}
02:  {
     // Define an element to represent the database
server
     // running on dbserver.domain.tld in validation
slice
03:     validation db::DBServer(IP="dbserver.domain.tld")
         with config mySQLCfg;
     // Define an element group to represent all
     // application servers running in the online
slice
04:     online as_all::ApplicationServerGroup(IP=".*")
         with config TomcatCfg("/path/to/web.xml")
         with log TomcatLog;
     // Wait for operator to start the task
05:     wait("Begin task"){ timeout = 30000;}
     // Wait for the operator to complete the task
06:     wait("Begin Validation"){timeout = 30000; }
     // Use the library ``connected'' to validate
that all
     // app servers are connected to the database
server
07:     use connected with[as_all, db] as
add_AS_connected;
     // Check that the App servers are load balanced
08:     assert balanced (EQUAL(as_all..cpu.util)){
09:             on;
10:        } else{//print "App servers load unbalanced"}
     ...
14:     wait("End Validation"){ timeout = 30000; }
15:  }
```

Figure 2: *A sample A program written to validate the task of adding an application server to the auction service.*

some correctness model. Toward this purpose, *A* supports a number of features such as statistical operators to ease the task of reasoning about service correctness.

More specifically, each *A* program comprises a set of *assertions* written about *elements*. Elements are typed objects representing the monitored state of running service components, such as Web or application servers. Critically, elements contain information about components' configuration parameters and log output in addition to the components' running state (Figure 2, lines 03 and 04). This allows validation programs to detect misconfigurations, which comprise a major class of mistakes. State information is presented as fields within elements, where each field can hold a value of a primitive data type, a stream of temporally sampled values (called a *stat* object), or another element representing the state of a sub-component. Stat objects are used to hold time series data, such as the CPU utilization over time at a server component. *A* provides a number of statistical operators to manipulate such time series.

Each element is dynamically *bound* to a specific component, e.g., the database server running on the host dbserver.domain.tld (line 03). Mapping the state of each component to the corresponding bounded element in an *A* program is the job of the runtime system (Section 6).

An element can also be bound to *a set* of components of the same type; e.g., the set of application servers in the auction service (line 04). Such an element is called an *aggregate* element and is motivated by the fact that components in Internet services are often replicated for availability and performance. Aggregate bindings are powerful because they allow assertions about all components within a replicated set to be independent of the actual number of active components at runtime. For example, an assertion can specify that all application servers in a multi-tier service should be relatively load balanced, regardless of the number of servers currently running (line 08).

Assertions are Boolean expressions on elements that represent beliefs about how a service should be configured and how it should behave (line 08). Assertions are executed periodically with configurable frequencies.

Assertions can be organized into *tasks* (line 01), which are constructs designed to model human-machine interactions. Tasks allow validation programs to wait for expected operator actions (lines 05 and 06) and to define how the service state should change upon the completion of these actions. For example, a program written to validate the addition of an application server to a multi-tier service might ensure that there is indeed one more application server running and that all Web servers are connected to it after the operator has completed her action.

Finally, to provide a method of abstraction, assertions can also be organized into *libraries* for reuse (lines 00 and 07).

6 The *A* Runtime System

In this section, we briefly describe the *A* runtime system as it is implemented at Ask.com. The implementation for the auction service is similar, although some details differ because of differences between the two monitoring infrastructures.

The *A* runtime system is responsible for obtaining the state of running service components and mapping this state to *A* elements. The runtime system is also responsible for scheduling and executing assertions from *A* programs.

The *A* runtime system is hosted by a component called the *Integrator* as shown in Figure 3. 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 link with an extensive set of middleware codes, which includes communication with a distributed directory service that maintains dynamic membership information and a remote control and monitoring package called CAPP. Maintaining the Component Database is then just a matter of periodically obtaining the set of active service components from the directory service. The components' attributes (e.g., name, IP address, etc.) are obtained either from the directory service or by contact-

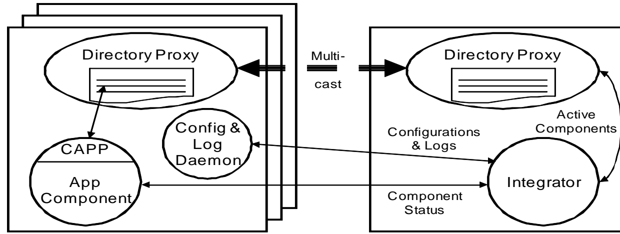

Figure 3: *The Ask.com A runtime system.*

ing the components themselves through the CAPP interface.

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 a second database called 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. `GenericServiceComponent` 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 the 6 types 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 via a binding statement, 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. Obviously, the element types were designed so that the fields match the attributes defined and maintained for the corresponding service component types.

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 parameters 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. We implemented this querying interface instead of having the monitor forward the appropriate configuration (or log) files to the Integrator because some configuration parameters may be stored as internal state of a service component, rather than in external configuration files. For instance, MySQL stores access control information in tables of a special database; therefore, obtaining such information requires querying the MySQL database.

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.

7 Evaluation

In this section, we describe our validation programs for the Ask.com crawler and the auction service, and evaluate their effectiveness in detecting system misbehaviors.

7.1 Ask.com

7.1.1 Crawler and Validation System

The Ask.com crawler comprises five different tiers, where each tier is a replicated set with hundreds to thousands of replicas. All communication is inter-tier and follows either a one-to-one or all-to-all pattern (see Figure 1). Inter-tier messages are carried by middleware distributed persistent communication channels, which are similar to the Unix named pipe except that each Ask.com channel supports an arbitrary number of distributed senders and receivers. The six element types mentioned in Section 6 correspond to components in the five tiers and the channels. 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 attributes.

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 sim-

ple optimizations, such as parallelizing the periodic contacting of active components, would allow us to sample once every minute.)

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.

The validation environment is somewhat different than that described in Section 2. 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 real Web. Changing from one configuration to another requires copying the software installation of one tier to a different set of machines and, typically, changing a small number of configuration parameters for a second tier 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, our Web servers could not emulate many of the error conditions that can and do occur on the Internet.

As already mentioned, we defined a flow model of 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 1, work units typically 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.

Using the above model, we then wrote one *A* program to validate three major tasks: installing a new instance of the crawler, adjusting the number of hosts and/or replicas, and upgrading the crawler software. We wrote one validation program to check overall system correctness, rather than task-specific programs, because the same validation program is also used to continuously monitor the health of the crawler. This is a significant advantage of model-based validation over other validation techniques (but a discussion of this monitoring application is beyond the scope of this paper). The disadvantage is that the validation program cannot use task-specific assertions.

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 real-life 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). A validation run may last between several hours to several days, depending on the extent of the changes being validated. If the run passes validation, the operator changes the crawling rate if necessary and runs the script to reconfigure the crawler to run against the Internet.

7.1.2 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 period. This is an estimate because our validation system was not ready for real use at the time. Thus, we typically 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.[2] (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 misbehaviors could have been caused by mistakes (and because it is generally interesting to see whether model-based validation can detect these misbehaviors).

Mistake 1. Early on in our development, inter-tier communication through the persistent channels described above did not have timeouts. If the crawler is shutdown for any reason, the operator was responsible for clearing

[2] One mistake was detected quickly but finding the root cause took a while.

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—the self-imposed rate is set in order to not overload Web servers with crawling traffic. Fortunately, this mistake was caught during the validation run (although it was not caught 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 detected this mistake very quickly as it caused a significant violation of flow preservation.

Mistake 2. Also early on in our development, the channels had capacity limits that were "soft" parameters; that is, the capacity limit of a channel was lost if the server managing it crashed. When the server recovered, it would set the channel's capacity 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 of the 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. 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, if we had had a chance to include 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 because it led to a violation of flow preservation.

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 detected this problem very quickly as one of the assertions checking for the composition of the work flow triggered on a much lower threshold for failed downloads. Interestingly, a similar mistake was made on another validation run, but only affected a very small percentage of the input workload. We were not able to detect this because the change in percentage of failed downloads was within the expected variance and so below the threshold of our checking assertion.

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 both because unnecessary work was done and because of the politeness violation. Model-based validation quickly caught this mistake, as it led to a significant violation of flow preservation.

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, the result of a failed HTTP request for a robots.txt because of a network or protocol error 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. Thus, during one validation run, 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.

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 to catch since they can have serious business consequences. On the other hand, because we are currently checking work flows at a coarse-grained level, mistakes/problems 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.

7.2 Auction Service

7.2.1 Validation System

We also defined models and wrote validation programs for an auction service modeled after EBay [23]. The service is organized into 4 tiers of servers: load balancer, Web, application, and database. Work units correspond to client requests. We defined a flow model similar to that for the Ask.com crawler. We also defined a hierarchical component model, specifying all the components that must be running for each of the servers to be completely "up," and an access control matrix specifying access rights.

We then used the above models together with results from previous works exploring the nature of operator mistakes [4, 10, 14, 16, 18, 19, 31] to guide the writing of 12 *A* libraries containing 49 assertions (749 lines of code). Three libraries containing 14 assertions were written to check the correctness characteristics of the flow model as discussed in Section 4. 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.

Three of the libraries deal with 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 to which a Web server can connect appears in two places in an Apache configuration file—these two lists should match. These libraries contain 18 assertions.

Three libraries contain 9 assertions about per-component policies; for instance, the load balancing policy of the Linux LVS balancer. Two libraries (4 assertions) assert that resource utilization should be balanced across the components of a replicated tier and that no component should be overloaded. Finally, the last library (4 assertions) asserts the access control matrix, which only included access control for the database server for simplicity.

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. 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 consists 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.

7.2.2 Mistake-Injection Experiments

To evaluate model-based validation for the auction service, we injected mistakes in the context of a variety of maintenance tasks. 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.

Our auction service comprised 1 load balancing server running Linux LVS, 2 Web servers running Apache, 2 application servers running Tomcat, and 1 database server running MySQL. Each server was a blade with a 1.2 GHz processor and 512 MB of RAM. Servers were connected by a Gbit Ethernet switch. The service was loaded using a client emulator. The overall load imposed on the system was 40 requests/second, which is roughly 50% of the maximum achievable throughput. The components in the validation slice were offered a load of 20 requests/second.

The Integrator was hosted on 1 additional blade. Our monitoring imposed overheads of at most 6% CPU utilization and a 1.4 KB/s data stream to the Integrator on each service component. The Integrator was only lightly loaded (e.g., average CPU utilization of 2.7% when the entire service was under validation).

Table 1 summarizes our experiments. We injected three categories of mistakes: those affecting the connec-

Category	Mistake	Impact	Task	Caught?
Connectivity	"LVS ARP problem": Web server not configured to ignore ARP requests. (**synthetic but well-known** [12])	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** [19])	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** [19, 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** [31])	Undesired/inappropriate distribution of load across Web servers.	Load balancer addition	Y
	Web server load balancer misconfigured. (**synthetic** [31])	Undesired/inappropriate distribution of load across application servers.	Web server addition	Y
	DBMS performance parameters configured suboptimally. (**survey** [18])	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** [18])	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** [18])	Serious security vulnerability.	Database creation	Y

Table 1: *Mistake-injection experiments: mistakes, motivating sources, impact, and whether they were detected by model-based validation.*

tivity of multiple components (labeled "connectivity"), those affecting the capacity of some subset of components ("capacity"), and those affecting the security of the system ("security"). One mistake was observed in a previous study [16] ("observed"). Some of the mistakes were reported in a survey of database administrators that we conducted [18] ("survey"). All others were synthetically generated but motivated by previous work on operator mistakes [4, 10, 14, 16, 18, 19, 31] ("synthetic"). The mistakes share two key characteristics: (1) they either occur frequently or can impact the system significantly, and (2) it is difficult or impossible to catch them using trace- and replica-based validation.

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

"LVS ARP problem". This is a well-known misconfiguration [12] that may occur when LVS is set up in 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 replies directly to the clients. A popular method to achieve this behavior is to assign the same IP address that LVS uses to receive requests 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. 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 relatively balanced 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 communicates

with Tomcat) keeps a list of available application servers based on the heartbeats. This allows the Web servers to stop sending requests to application servers taken off-line for maintenance/validation. Because this is a local extension, Apache's JK module must be recompiled to support this protocol.

One of the two injected mistakes pertaining to the membership protocol 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. We could have caught this mistake by adding a configuration attribute to the Web server element type that corresponds to the number of threads attached to the membership shared-memory segment and an assertion to ensure that this attribute is greater than 0. However, for the purposes of our evaluation, we deem this mistake as not caught.

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.

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.

When injected into LVS, this mistake triggered the assertions requiring the CPU and memory utilization of the Web servers to be relatively balanced. In addition, since we assumed that the Web servers were homogeneous, the assertion requiring their weights to be equal also triggered.

Summary. Overall, model-based validation caught 10 out of the 11 injected mistakes shown in Table 1. Many of the assertions that caught mistakes were ones that check configuration parameters, highlighting the importance of validating configurations, in addition to dynamic service behaviors. Further, in contrast to the Ask.com system, where the current coarse-grained assertions do not provide much help in identifying the source of a mistake, the assertions that triggered in these experiments gave very strong clues for finding the mistake. We believe this is because we worked with the auction service over a longer period of time, and so the models and corresponding validation programs were more refined (see discussion in the next section).

8 Discussion And Lessons

Our proposal of model-based validation often prompts the following question: *are there service engineers with sufficient knowledge of the entire system to write practical yet effective validation programs?* Our experience at Ask.com provides a strong *yes* answer. In particular, Tjang, the author who implemented the Ask.com model-based validation system, started with no knowledge of the crawler. Reviewing the design documentation and discussing the crawler with lead engineers led to the adoption of the flow model as the overall abstraction of the system.

Tjang then collaborated with component developers to define the attributes/properties that should be monitored. The initial discussion with each component developer required on average 2 hours. Understanding the components at a sufficient level of detail took approximately two weeks.

Many of the attributes/properties to be monitored were already implemented. Thus, the component developers required only a small amount of time to completely support what they and Tjang agreed upon. A number of improvements were made over time, but no major implementation effort was ever required.

Tjang then developed the validation program described above using the flow model as the primary guidance. Writing, testing, and improving the validation program took approximately 2 months.

Reflecting on our experience over both validation efforts, we observe that model-based validation's ability to span a range of modeling efforts is the fundamental characteristic that makes it practical to implement in real systems. In particular, to start, a model-based validation system can be quickly deployed using an initial set of high-level models and corresponding validation programs. Likely, this initial infrastructure will only catch gross mistakes, e.g., one that causes a significant violation of a flow principle. However, these kinds of mistakes (and bugs) are the critical, high priority ones and so it is worthwhile to deploy these initial validation programs as soon as possible. This situation is similar to where we are today with the Ask.com system.

Subsequently, service engineers can iteratively develop more detailed models and corresponding validation programs. These more detailed validation programs should be able to catch mistakes that have less obvious impact on the system (but should be caught and corrected nevertheless). In addition, the more detailed assertions provide much more help to the operator for identifying

the source of the mistake. While the more detailed models and validation programs do require increased understanding of the system, we strongly believe that it is still within the ability of one (or a small group) of service engineer(s) to learn within a reasonable amount of time. This situation is similar to where we have gotten to with the auction service.

9 Conclusions

In this paper, we proposed using model-based validation as a strategy for identifying operator mistakes in Internet services. We applied it to two Internet services: the real-life Ask.com crawler and a smaller scale auction service. We showed how the use of three relatively simple models can drive the systematic design and implementation of effective validation programs. Correctness conditions derived from the models were straightforward to express in (remarkably compact) *A* validation programs.

We evaluated our approach in terms of its effectiveness in detecting the mistakes/problems that arose during a number of validation runs of the crawler. Model-based validation would quickly have detected 5 out of 6 of these problems. We also performed mistake-injection experiments with the auction service. To ensure that our injected mistakes were of comparable complexity and subtlety to real mistakes, we used a combination of mistakes observed from human-factors studies, those reported 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] AGUILERA, M. K., ET AL. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. of SOSP* (Oct. 2003).

[2] ANDERSON, P., GOLDSACK, P., AND PATERSON, J. SmartFrog meets LCFG: Autonomous Reconfiguration with Central Policy Control. In *Proc. of LISA* (Oct. 2003).

[3] BARHAM, P., ET AL. Magpie: Real-Time Modelling and Performance-Aware Systems. In *Proc. of HOTOS IX* (May 2003).

[4] BARRETT, R., ET AL. Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices. In *Proc. of CSCW* (Nov. 2004).

[5] BODIK, R., ET AL. Advanced Tools for Operators at Amazon.com. In *Proc. of the HOTAC* (June 2006).

[6] BROWN, A. B., AND PATTERSON, D. A. Undo for Operators: Building an Undoable E-mail Store. In *Proc. of USENIX* (June 2003).

[7] CHEN, M. Y., ET AL. Path-Based Failure and Evolution Management. In *Proc. of NSDI* (Mar. 2004).

[8] GRAY, J. Why do Computers Stop and What Can Be Done About It? In *Proc. of SRDS* (Jan. 1986).

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

[10] HERBERT, G. W. Failure from the Field: Complexity Kills. In *Proc. of EASY* (Oct. 2002).

[11] KELLER, L., UPADHYAYA, P., AND CANDEA, G. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *Proc. of DSN* (June 2008).

[12] LINUX.ORG. Linux Virtual Server. http://www.linuxvirtualserver.org, 2006.

[13] LIU, X., ET AL. D^{3}S: Debugging Deployed Distributed Systems. In *Proc. of NSDI* (Apr. 2008).

[14] MAGLIO, P., AND KANDOGAN, E. Error Messages: What's the Problem? *ACM Queue* (Nov. 2004).

[15] MURPHY, B., AND LEVIDOW, B. Windows 2000 Dependability. Tech. Rep. MSR-TR-2000-56, Microsoft Research, June 2000.

[16] NAGARAJA, K., ET AL. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proc. of OSDI* (Dec. 2004).

[17] NETO, A. A., AND VIEIRA, M. Towards Assessing the Security of DMBS Configurations. In *Proceedings of DSN* (June 2008).

[18] OLIVEIRA, F., ET AL. Understanding and Validating Database System Administration. In *Proc. of USENIX* (June 2006).

[19] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. Why do Internet Services Fail, and What Can Be Done About It. In *Proc. of USITS* (Mar. 2003).

[20] PATTERSON, D. A., ET AL. ROC: Motivation, Definition, Techniques, and Case Studies. Tech. Rep. UCB//CSD-02-1175, UC, Berkeley, Mar. 2002.

[21] PERL, S. E., AND WEIHL, W. E. Performance Assertion Checking. In *Proc. of SOSP* (Dec. 1993).

[22] REYNOLDS, P., ET AL. Pip: Detecting the Unexpected in Distributed Systems. In *Proc. of NSDI* (May 2006).

[23] RICE UNIVERSITY. DynaServer Project. http://www.cs.rice.edu/CS/Systems/DynaServer, 2003.

[24] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proc. of SOSP* (Oct. 2007).

[25] TAN, Y.-L., ET AL. Comparison-based File Server Verification. In *Proc. of USENIX* (June 2005).

[26] TJANG, A., ET AL. A: An Assertion Language for Distributed Systems. In *Proc. of PLOS* (Oct. 2006).

[27] VERBOWSKI, C., ET AL. Flight Data Recorder: Monitoring Persistent-State Interactions to Improve Systems Management. In *Proc. of OSDI* (Nov. 2006).

[28] VERBOWSKI, C., ET AL. LiveOps: Systems Management as a Service. In *Proc. of LISA* (Dec. 2006).

[29] WANG, H., ET AL. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proc. of OSDI* (Dec. 2004).

[30] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proc. of OSDI* (Dec. 2004).

[31] WOOL, A. A Quantitative Study of Firewall Configuration Errors. *IEEE Computer* (2004).

[32] ZHENG, W., BIANCHINI, R., AND NGUYEN, T. D. Automatic Configuration of Internet Services. In *Proc. of EuroSys* (Mar. 2007).