Model-Based Validation for Internet Services

Andrew Tjang, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, Thu D. Nguyen
Department of Computer Science
Rutgers University
Piscataway, NJ 08854, USA
Email: {atjang, fabiool, ricardob, rmartin, tdnguyen}@cs.rutgers.edu

***Abstract*—Operator mistakes are 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. Previous validation strategies have limitations, however, including the need for instances of correct behavior for comparison. In this paper, we propose a novel *model-based validation* strategy that addresses these limitations and complements our previous techniques. 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 validation assertions that 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 demonstrates that it is highly effective at detecting and hiding both activated and latent mistakes.**

***Keywords*-validation; model; operator mistake; internet service;**

I. 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. Thus, ensuring high service availability is challenging ([1], [2]).

Our work seeks to alleviate one important source of service failures: *operator mistakes*. Several studies have shown that mistakes are a significant source of unavailability [1]–[5]. For instance, Oppenheimer et al. [1] show that mistakes were responsible for 19-36% of failures, and, for 2 out of 3 services, were the dominant source of failures and the largest contributor to time to repair. Similarly, Oliveira et al. [5] report that operator mistakes are responsible for a large fraction of the problems in database administration. Both corroborate an older study of Tandem systems where mistakes were a dominant reason for outages [3].

In our previous work, we proposed operator action *validation* as an approach for detecting mistakes while hiding them from the service and its users ([5], [6]). In this approach, a validation framework creates an isolated extension of the online service in which operator actions are performed and later validated. Before the operator acts on a service component, the component is moved to this extension. After the operator activity is completed, the correctness of the operator's actions is validated by comparing the behavior of the component with that from either a trace or an online component. If validation succeeds, the system moves the component back online; otherwise, it alerts the operator. While this validation strategy can detect and hide a large class of mistakes, it has three important limitations: (1) it requires known instances of correct behavior for comparison; (2) it provides no guidance in pinpointing mistakes; and (3) it fails to detect latent mistakes.

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

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

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

We also implemented a prototype model-based validation system for an academic yet realistic online auction service [7]. This service is smaller in scale than the Ask.com system but allows us to evaluate model-based validation more extensively using mistake injection. This system detected 10 out of 11

This work was supported in part by NSF grant CNS-0509007.

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

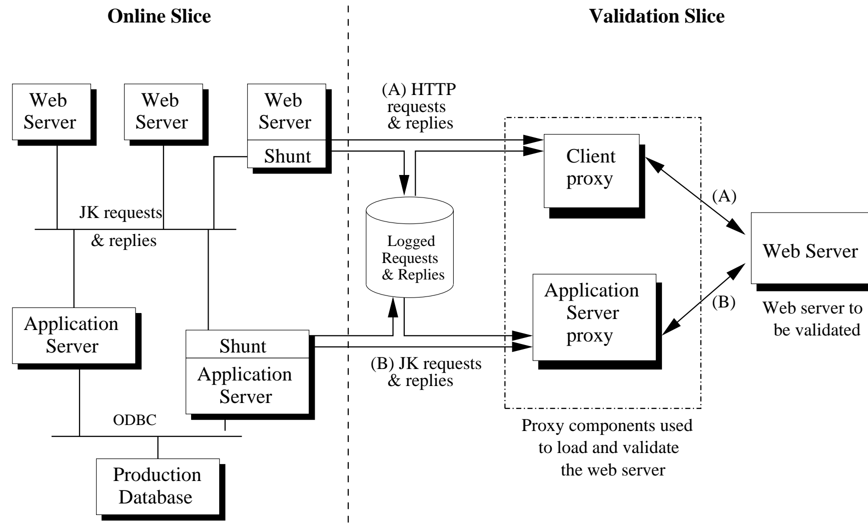

Fig. 1. *The auction service with validation. In this particular case, a single component, a Web server, is being validated inside the validation slice. The validation harness uses one or more client proxies to load the Web server and one or more application server proxies to field requests for dynamic content from the Web server. The proxies can either use previously logged data or a duplicate of the current workload of a functionally equivalent component in the online slice to load the Web server being validated.*

plausible yet not easily anticipated mistakes injected during a variety of operator tasks. All of these would have been missed using our previous validation strategies. Model-based validation would also have detected all 28 previously observed mistake instances that our previous validation strategies were able to detect [6].

The above implementations have three main parts. (1) A small set of models that we decided were simple yet allowed clear description of system correctness. We describe these models in Section IV. (2) An exploratory language called *A* designed to make it easy and convenient to express correctness assertions derived from the models. We briefly describe *A* in Section V. And (3) a runtime system that maps the monitored states of a service to *A* program objects and runs *A* validation programs. We describe this runtime system in Section VI.

We present our evaluation of model-based validation in Section VII and discuss our experience in Section VIII.

Our main contributions include:

- 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; and
- Describing the effort involved in applying model-based validation to the Ask.com crawler and demonstrating that it is scalable in practice along all critical dimensions, including service knowledge and modeling, development of validation programs, runtime state collection, and runtime execution of assertions.

II. Validation Background

The core idea of validation is to verify operator actions under realistic workloads in a realistic but isolated validation environment [6]. Mistakes can then be caught before becoming visible to the users. To achieve realism, the validation environment is hosted by the online system itself (Figure 1). In particular, a service with validation is divided into two slices, an online slice that hosts online components and a validation slice where components can be operated on and validated before being re-integrated into the online slice. The validation slice contains a testing harness that can be used to load the components to be validated and to check their correctness. To achieve isolation, the components placed in the validation slice are *masked* (isolated) from the online slice using layer 2 and 3 virtual networking. Server nodes can be moved between slices without changing configuration parameters of the nodes or the software components that they host.

Validation proceeds as follows. Suppose an operator needs to operate on a service component (e.g., to upgrade its software). Before starting, the operator uses a script to move the server hosting the component from the online slice to the validation slice. This takes the node offline and completely masks it from all online components. The operator can now work on the component without affecting the online system. After completing her task, the operator surrounds the masked component with proxies that give the illusion that the masked component is in a complete system. She then places a validation 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 compares 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), 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 environ-

ments 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 production systems [5]. Also, it is often difficult to apply realistic workloads in an offline testing environment. Thus, even with careful testing, operators can make mistakes when changing or deploying their changes to the online system. Validation closes this gap between offline testing and the online system, although the two approaches can be complementary: validation could be applied as the last step in a testing/validation process before exposing an operator action to the online system.

In [5], we revisited trace-based and replica-based validation for database servers, whereas Tan *et al.* [8] revisited replica-based validation for file servers. We also proposed a primitive version of model-based validation in [5]. 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.

III. 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., [9]–[11]. Unfortunately, many tasks cannot be automated, creating the possibility of operator mistakes. Recommendation systems, e.g., [12], attempt to prevent mistakes by guiding the operators' actions. As an extra safety net, previous validation systems ([5], [6], [8]) hide and detect certain types of mistakes by confining the operators' actions to a sandboxed environment. When mistakes are made, recovery/undo systems ([10], [13]–[15]) can be used to bring the service (or the sandbox) back to a proper state. Finally, monitoring/auditing systems, e.g., [16]–[23], attempt to detect (and sometimes diagnose) performance and behavioral problems regardless of their root causes.

Our work is orthogonal and complementary to recommendation and recovery/undo systems. It is also related to monitoring/auditing systems like PSpec [20], Pip [21], and D^3S [23] that use assertion checking to help debug software, with the latter two focusing on distributed systems. However, because they are concerned with detecting bugs, and not operator mistakes, these systems did not consider some important static and structural issues, such as improper system configurations and latent security problems.

In contrast, the focus of [16] was performance-debugging systems with as little application-specific knowledge as possible by viewing the system as a black box and examining bottlenecks. Pinpoint [19] and Magpie [17] 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 declare correct behavior, which is critical when no previous samples exist—a common problem in the face of operator actions. The operator task is a first-class concept in model-based validation and our assertion language, allowing us to relate mistakes to specific tasks.

Along the lines of software testing, a recent related work is ConfErr [24], 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 [25]. 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.

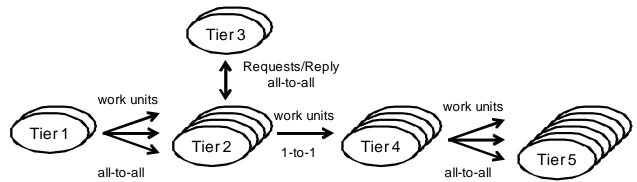

Fig. 2. Partial flow model for the crawler.

IV. 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 and the auction service.

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

Describing an Internet service as a flow of requests is not a new concept, e.g., [19], [21]. However, a flow model immediately brings to mind a number of correctness characteristics that should be validated. First, it dictates that the service engineer must reason about *connectivity*, i.e., how components are connected to one another and how requests should flow through the system. We believe that checking system connectivity should be quite useful, since many mistakes observed in our previous study [6] led to incorrect connectivity.

A second characteristic is *flow preservation*. Requests can only enter the flow graph at known sources and then leave the

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

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

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

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

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

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.

V. THE *A* ASSERTION LANGUAGE

Our model-based validation programs are written in a new language called *A* [26]. We use *A* because 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. Within the scope of this study, however, a library written in C++ or Java would have served equally well. Thus, in this section, we focus only on general features that we found to be useful for reasoning about service configurations and behaviors and not on specific details of *A* . Figure 3 gives a small example *A* program to help make our description more concrete.

```
00: #include "connected.lib";
01: task add_app_server {name="Add an App Server";}
02: {
     // Define an element to represent the DB server
03:   validation db::DBServer(IP="dbserver.domain.tld")
         with config mySQLCfg;
     // Define an element group to represent all
     // app 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
     // DB 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: }
```

Fig. 3. Sample *A* program to validate the task of adding an application server to the auction service.

The fundamental idea behind *A* is to capture and expose the state of a running service as a set of language objects. *A* supports a number of features to ease the task of reasoning about service correctness. For example, each *A* program comprises a set of *assertions* written about *elements*. Elements are typed objects representing the monitored state of running service components. Elements contain information about components' configuration parameters and log output in addition to the components' running state (Figure 3, lines 3 and 4). 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 (*stat* object), or another element representing the state of a sub-component. Stat objects 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. For example, the `EQUAL` in line 8 is one such operator, which identifies statistical outliers, rather than a strict equality comparison.

Each element is dynamically *bound* to a specific component, e.g., the database server running on the host dbserver.domain.tld (line 3). Mapping the state of a component to the corresponding element in an *A* program is the job of the runtime system (Section VI).

Each *A* element can optionally have one or more attached *configuration* and/or *log* objects. Each attached configuration object refers to a set of static configuration parameters for the service component bound to the element. Each attached log object refers to a stream of output logged by the component

bound to the element. Exposing configuration parameters is particularly important because our previous study of operator mistakes shows that misconfiguration comprise a major class of mistakes.

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 4). 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, it is very simple to assert that all application servers should be relatively load balanced, regardless of the number of servers currently running (line 8).

Assertions are Boolean expressions on elements that represent beliefs about how a service should be configured and how it should behave (line 8). Assertions are executed periodically with configurable frequencies.

Assertions can be organized into *tasks* (line 1), which are constructs designed to model human-machine interactions. Tasks allow validation programs to wait for expected operator actions (lines 5 and 6) 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 0 and 7).

VI. 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. It 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 4. The Integrator maintains a database of active service components, called the *Component Database*, against which *A* elements can be dynamically bound. In the Ask.com system, all service components communicate with a distributed directory proxy that maintains dynamic membership information and are linked to a remote control and monitoring package called CAPP. The Component Database is maintained by periodically obtaining the set of active service components from the directory proxy. The components' attributes are obtained either from the directory proxy or by contacting the components themselves through CAPP.

The Integrator also periodically contacts each active service component through the CAPP interface to monitor time series

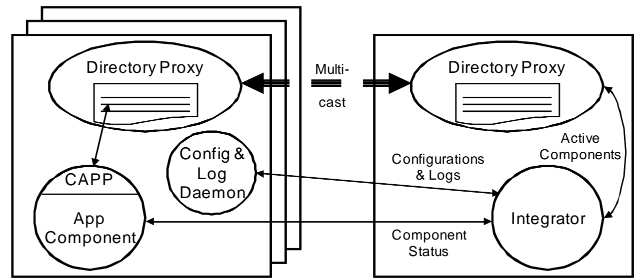

Fig. 4. The Ask.com *A* runtime system.

attributes, such as the number of requests serviced and CPU utilization. This monitoring data is maintained in the *Stat Database*.

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

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

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

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

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

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

VII. Evaluation

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

A. Ask.com Crawler

1) Crawler and Validation System: The Ask.com crawler comprises 5 different tiers, where each tier has hundreds to thousands of replicated components. All communication is inter-tier and follows either a one-to-one or all-to-all pattern (see Figure 2). Inter-tier messages are carried by distributed persistent communication channels similar to the Unix named pipe, except that each channel supports an arbitrary number of distributed senders and receivers. The 6 element types mentioned in Section VI correspond to components in the 5 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.

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

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

The validation environment is somewhat different than the ideal environment described in Section II. In particular, we do not have the capability of dynamically creating a validation slice. Rather, the system can be configured to crawl an internal testbed that emulates the Web for validation. Changing from the validation configuration to one that crawls the real Internet requires copying the tier 4 software to a different set of machines and, typically, changing a small number of configuration parameters for tier 1 to increase the crawling rate. The first change is entirely script-based so that direct human action is quite limited.

The testbed that emulates the Web comprises a cluster running a large number of Web servers. The URLs and content served by these servers are realistic since they derive from traces of actual crawls. On the other hand, the emulation has two limitations. First, the parallelism achievable against the testbed is much smaller than that against the real Internet. Second, the Web servers in the testbed could not emulate many of the error conditions that can and do occur on the Internet.

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

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

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

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. This is an estimate because our validation system was not ready for real use at the time. We view the very last test run before entering production mode as a validation run. During these runs, there were a handful of mistakes and/or software problems that were easily identifiable by the normal monitoring system. There were also 6 instances (4 due to operator mistakes and 2 due to software/design bugs) that were not detected until after considerable post-run analysis.[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

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

executed the *A* program described above on a live Integrator and replayed the logs from the 6 cases to feed the Integrator with monitoring data. We consider the two cases that were not operator mistakes because similar misbehavior could have been caused by mistakes (and because it is interesting to see if model-based validation can detect such misbehavior).

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

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

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

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

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

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

Summary. Model-based validation quickly detected 5 out of the 6 problems described above. We find that the flow model and the corresponding *A* program are quite effective at detecting mistakes/problems that lead to statistically significant violations of flow principles, such as flow preservation and expected composition. These are the *most important* mistakes since they can have serious business consequences. On the

other hand, because we are currently checking work flows at a coarse-grained level, mistakes that lead to changes in the flow that are within the expected variance thresholds cannot be detected. Fortunately, these "small" mistakes typically do not have significant impact.

B. Auction Service

1) Auction Service and Validation System: We also defined models and wrote validation programs for an EBay [7] like auction service. 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 and an access control matrix model.

We then used the above models together with results from previous works exploring the nature of operator mistakes ([1], [5], [6], [27]–[30]) to guide the writing of 12 *A* libraries containing 49 assertions (749 lines of code). Three libraries (14 assertions) were written to check the flow model. These libraries contain assertions on inter-tier connectivity and flow capacity. Three libraries (18 assertions) deal with the correct running of components within tiers. They also contain assertions about the consistency of related configuration parameters of each component type. Three libraries (9 assertions) deal with 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 a Web server, (2) adding an application server, (3) adding a load balancer, and (4) adding a database to the DBMS. 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;[3] 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.

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 LVS load balancer, 2 Apache Web servers, 2 Tomcat application servers, and 1 MySQL database server, each with a 1.2 GHz CPU and 512 MB of RAM. The service was loaded using a client emulator. The overall load imposed on the system was 40 requests/second (roughly 50% of the maximum achievable throughput). The components under validation received 20 requests/second.

The Integrator was hosted on 1 additional machine. 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 (average CPU utilization of 2.7% when the entire service was under validation).

Table I summarizes our experiments. We injected 3 categories of mistakes: those affecting the connectivity of multiple components; those affecting the capacity of a subset of components; and those affecting the security of the system. One mistake was observed in a previous study [6]. Some mistakes were reported in a survey of database administrators we conducted [5]. All others were synthetically generated but motivated by previous work on operator mistakes ([1], [5], [6], [27]–[30]). 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 [31] that may occur when LVS is set up in direct-routing mode. In order to use this mode, Web servers must be configured to ignore ARP requests for the loopback devices. Failure to do so would result in a race condition where some requests would bypass the load balancer and go directly to the Web servers. This mistake is particularly interesting because it illustrates the importance of configuration assertions.

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. For completeness, we performed 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

[3]Elements can be bound to components in both the online and validation slices so that the behavior of masked components can bed checked against that of active components.

TABLE I
SUMMARY OF MISTAKE-INJECTION EXPERIMENTS.

Category	Mistake	Impact	Task	Detected?
Connectivity	"LVS ARP problem": Web server not configured to ignore ARP requests. (**synthetic but well-known** [31])	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** [1])	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** [1], [28])	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** [29])	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** [27])	It causes the system not to work at the maximum capacity.	Application server addition	Y
	Wrong front-end load balancer policy is activated. (**synthetic** [30])	Undesired/inappropriate distribution of load across Web servers.	Load balancer addition	Y
	Web server load balancer misconfigured. (**synthetic** [30])	Undesired/inappropriate distribution of load across application servers.	Web server addition	Y
	DBMS performance parameters configured sub-optimally. (**survey** [5])	Service overall performance might be jeopardized.	Database creation	Y
Security	Database administrator account not assigned a password. (**observed** [6])	Serious security vulnerability.	Database creation	Y
	Allowing any machine to access the database remotely. (**survey** [5])	Security vulnerability and possibility of data corruption due to nonmalicious yet unauthorized data access.	Database creation	Y
	Allowing an ordinary user to grant/revoke privileges to/from other users. (**survey** [5])	Serious security vulnerability.	Database creation	Y

server, bypassing the load balancer. This time, the configuration assertion and the assertions about balanced CPU and memory utilization failed.

Membership protocol mistakes. In the auction service, the application servers periodically send heartbeat messages to the Web servers; accordingly, an Apache module (JK) keeps a list of available application servers based on the heartbeats. As this is a local extension, the module must be recompiled to support this membership protocol. One mistake was not compiling the JK module of a new Web server with support for it. The affected JK module would rely on a static list of application servers.

Replica and trace-based validation cannot deal with the above latent mistake because it will not be activated until the set of Application servers changes after the affected Web server has been brought online. During the experiment, the A program we used for model-based validation also did not detect this mistake. We could have detected this mistake by adding support for checking a configuration attribute. However, for the purposes of our evaluation, we deem this mistake as not detected.

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 triggered 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 detected 10 out of the 11 injected mistakes shown in Table I. 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).

VIII. 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 says *yes*. 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 it with lead engineers led to the adoption of the flow model

as the overall abstraction of the system. This review involved collaboration with component developers to define the attributes/properties that should be monitored. The initial discussion with each 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.

During this experience, 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. A model-based validation system can be quickly deployed using an initial set of high-level models and corresponding validation programs. This initial infrastructure will likely only catch gross mistakes; however, as these kinds of mistakes are the critical ones, it is worthwhile to deploy initial validation programs as soon as possible.

Subsequently, service engineers can iteratively develop more detailed validation programs which should be able to catch mistakes that have less obvious impact on the system. In addition, the more detailed assertions help the operator identify 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 few) of the service engineer(s) to learn within a reasonable amount of time.

IX. 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 different systems: the real-life Ask.com crawler and a smaller scale auction service. We showed how three relatively simple models can drive the systematic design and implementation of effective validation programs. Correctness conditions derived from the models were easy to express in compact *A* 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 5/6 of these problems. We also performed mistake-injection experiments with the auction service. To ensure realism in the injected mistakes we used a combination of mistakes observed from human-factors studies, reported in the literature, and reported by database administrators in a survey. We found our model-based validation approach highly effective, catching 10/11 injected mistakes, none of which could be found using trace and replica-based validation.

X. Acknowledgment

We thank Kiran Nagaraja for his participation in the conception and early prototyping of model-based validation. We thank Ask.com, especially Tao Yang, Hong Tang, Srinath Rao, and the Crawler Development Group, for supporting the prototyping of model-based validation in the Ask.com crawler.

References

[1] D. Oppenheimer *et al.*, "Why do Internet Services Fail, and What Can Be Done About It?" in *USITS*, 2003.
[2] D. A. Patterson *et al.*, "ROC: Motivation, Definition, Techniques, and Case Studies," UC, Berkeley, Tech. Rep. UCB//CSD-02-1175, Mar. 2002.
[3] J. Gray, "Why do Computers Stop and What Can Be Done About It?" in *SRDS*, 1986.
[4] B. Murphy and B. Levidow, "Windows 2000 Dependability," Microsoft Research, Tech. Rep. MSR-TR-2000-56, June 2000.
[5] F. Oliveira *et al.*, "Understanding and Validating Database System Administration," in *USENIX*, 2006.
[6] K. Nagaraja *et al.*, "Understanding and Dealing with Operator Mistakes in Internet Services," in *OSDI*, 2004.
[7] Rice University, "DynaServer Project," http://www.cs.rice.edu/CS/Systems/DynaServer, 2003.
[8] Y.-L. Tan *et al.*, "Comparison-Based File Server Verification," in *USENIX*, 2005.
[9] P. Anderson *et al.*, "SmartFrog Meets LCFG: Autonomous Reconfiguration with Central Policy Control," in *LISA*, 2003.
[10] Y.-Y. Su *et al.*, "AutoBash: Improving Configuration Management with Operating System Causality Analysis," in *SOSP*, 2007.
[11] W. Zheng *et al.*, "Automatic Configuration of Internet Services," in *EuroSys*, 2007.
[12] R. Bodik *et al.*, "Advanced Tools for Operators at Amazon.com," in *HotAC*, 2006.
[13] A. B. Brown and D. A. Patterson, "Undo for Operators: Building an Undoable E-mail Store," in *USENIX*, 2003.
[14] H. Wang *et al.*, "Automatic Misconfiguration Troubleshooting with PeerPressure," in *OSDI*, 2004.
[15] A. Whitaker *et al.*, "Configuration Debugging as Search: Finding the Needle in the Haystack," in *OSDI*, 2004.
[16] M. K. Aguilera *et al.*, "Performance Debugging for Distributed Systems of Black Boxes," in *SOSP*, 2003.
[17] P. Barham *et al.*, "Magpie: Real-Time Modelling and Performance-Aware Systems," in *HotOS IX*, 2003.
[18] C. Verbowski *et al.*, "LiveOps: Systems Management as a Service," in *LISA*, 2006.
[19] M. Y. Chen *et al.*, "Path-Based Failure and Evolution Management," in *NSDI*, 2004.
[20] S. E. Perl and W. E. Weihl, "Performance Assertion Checking," in *SOSP*, 1993.
[21] P. Reynolds *et al.*, "Pip: Detecting the Unexpected in Distributed Systems," in *NSDI*, 2006.
[22] C. Verbowski *et al.*, "Flight Data Recorder: Monitoring Persistent-State Interactions to Improve Systems Management," in *OSDI*, 2006.
[23] X. Liu *et al.*, "D^{3}S: Debugging Deployed Distributed Systems," in *NSDI*, 2008.
[24] L. Keller *et al.*, "ConfErr: A Tool for Assessing Resilience to Human Configuration Errors," in *DSN*, 2008.
[25] A. A. Neto and M. Vieira, "Towards Assessing the Security of DBMS Configurations," in *DSN*, 2008.
[26] A. Tjang *et al.*, "A: An Assertion Language for Distributed Systems," in *PLOS*, 2006.
[27] R. Barrett *et al.*, "Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices," in *CSCW*, 2004.
[28] G. W. Herbert, "Failure from the Field: Complexity Kills," in *EASY*, 2002.
[29] P. Maglio and E. Kandogan, "Error Messages: What's the Problem?" *ACM Queue*, Nov. 2004.
[30] A. Wool, "A Quantitative Study of Firewall Configuration Errors," *IEEE Computer*, 2004.
[31] Linux.org, "Linux Virtual Server," http://www.linuxvirtualserver.org, 2006.