Abstract. Operator mistakes have been identified as a significant source of unavailability in Internet services. In this paper, we propose a new language, \( A \), for service engineers to write assertions about expected behaviors, proper configurations, and proper structural characteristics. This formalized specification of correct behavior can be used to bolster system understanding, as well as help to flag operator mistakes in a distributed system. Operator mistakes can be caused by anything from static misconfiguration to physical placement of wires and machines. This language, along with its associated runtime system, seeks to be flexible and robust enough to deal with the wide array of operator mistakes while maintaining a simple interface for designers or programmers.

1. Introduction

Large and complex distributed systems are expanding in importance as users’ dependency on Internet services grows. Reasoning about correct behavior of such systems is difficult because these systems are comprised of complex conglomerates of distributed hardware and software, such as load balancers, loggers, databases, and application servers.

Unlike traditional monolithic applications, these systems require considerable human interaction to keep them operational, and these interactions are a significant source of unavailability [10, 11, 17, 20, 21, 24]. More recently, we found that mistakes are responsible for a large fraction of the problems in database administration [20].

In this paper, we propose a new method of reducing the above class of errors. Our approach uses a new assertion language, called \( A \), combined with a runtime monitoring system to directly support abstractions that allow system engineers to both (1) describe correct behavior and (2) reason about human-computer interactions in complex distributed systems. The goals of the \( A \) language and runtime include both reducing the number and impact of operator mistakes, as well as reducing the human cost of constructing models of correctness for distributed services.

For years, administrators have developed individual, ad-hoc methods of managing operator mistakes and the resulting system errors. Indeed, often significant IT budgets are devoted to developing, maintaining and managing in-house software to operate large-scale systems. We believe that a language specifically tailored to the task of operator-system interaction could improve this process.

Instead of a collection of ad-hoc scripts tailored to a specific site and environment, a language-based approach will allow a concise, understandable, and verifiable method of specifying correctness. Codifying the process in a specific language would also ease the generalization and parameterizing of operator tasks, thus allowing for the re-use of software to manage operator interactions across different organizations. For example, an organization could re-use existing parameterized libraries to manage the interactions with load-balancers, databases and application servers instead of having to develop custom scripts as they do today.

In this paper we thus argue:

- The abstractions in \( A \) are suited for verifying correct behavior in distributed systems;
- The \( A \) runtime supports these abstractions; and
- It is easier to use the \( A \) language and runtime for preventing and catching mistakes than ad-hoc solutions.

\( A \) provides a way to name components of a service, access the state of the components (both dynamic attributes and static properties), and conveniently write assertions about these states. The runtime system provides a gateway between \( A \) programs and a service monitoring infrastructure that provides dynamically measured information about the service to the \( A \) programs. To abstract human actions, the language has constructs to allow a programmer to describe discrete steps of an operator’s task as well how their models may evolve during the execution of the task. One example might be the task of load balancing - switching a load balancing policy - from a completely balanced one to a weighted one.

We also consider that while modeling performance captures dynamic behaviors, many mistakes result in improper static configurations. In fact, a good number of mistakes resulting in security and latent performance faults fit this pattern. \( A \) and its runtime system have specific constructs that represent configuration state (e.g., a start-up file) as well as its audit state (e.g., a log). We have found that validating both static and dynamic properties was critical for completeness; that is, both are needed to cover a wide range of mistakes.

In the remainder of the paper we introduce the reader to the \( A \) language and describe the constructs that make \( A \) uniquely suited to help system designers formalize their ideas of correct behavior. Section 2 details \( A \), with Section 2.3 introducing the running example mistake used throughout this paper. Section 3 provides a brief overview of the supporting runtime system. Section 4 overviews our evaluation methodologies and results. Section 5 describes related research, with Section 5.2 describing current solutions and their inadequacies. Finally, Section 6 looks at ongoing work and future directions.

2. The A Assertion Language

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

2.1 Overview

\( A \) ’s primary constructs are assertions, elements, and tasks. These three abstractions allow the programmer to (1) describe correctness, (2) access system state, and (3) reason about temporal state changes with respect to operator actions.

The assert construct was directly inspired by its use in procedural languages such as C and Java. Anecdotally, maintaining a large set of assertions about correct conditions has been critical toward...
writing correct programs. Indeed, [15], devotes an entire chapter to the subject of how to craft assertions for C programs. Some of the key ideas from that work that cross-over into the domain of correctness for the systems we describe are that assertions should (1) be side-effect free, (2) exhibit fail-fast behavior, and (3) should be explicitly labeled to the high-level conditions they are testing.

Elements in A were modeled after aggregate types, such as the C struct. In our case, an element describes the current system state of devices such as load balancers and databases. It is the runtime's job to map the language-level declared state to the current values of the devices. For example, the number of jobs in the run-queue of the OS on a particular machine might be a field of an element. A special stat variable type captures the notion of statistically sampled values, such as load average, for our class of system devices. In addition, because we are concerned with operator mistakes, elements can also describe the configuration state of devices; e.g. values in a configuration file.

Tasks represent sequential execution of an operator's actions. The language-level construct allows for the programmer to specify to wait for operators to complete actions. Tasks also allow the operator mistakes, elements can also describe the configuration state of devices; e.g. values in a configuration file.

Execution in A follows a discrete event model, similar in spirit to triggered databases or user-interface libraries (e.g. the Tk widget library [23]). The run-time takes monitored and measured values from real devices as well as input from human operators and forwards them to a central location. Assertions checks and wait statements in tasks map to a scheduled event stream executed at the central location. Operators are then notified when assertions fail.

2.2 Language Idioms

The decision to design a new language should not come lightly. There are many considerations that need to be evaluated. There is overhead associated with designing and implementing a niche language as well as a developer learning curve. In the case of model based validation, we found that the advantages of using a language like A outweighed the aforementioned overheads.

We believe there are certain ideas that contribute to the success of model based validation in the domain of distributed systems. If these ideas are incorporated into any language - either as a set of library calls in an existing language, or part of actual constructs within a domain specific language such as A, the result should be a flexible platform on which to develop and codify our models. To this end, we have chosen to use A for its simplicity and readability, as well as its ability to demonstrate these tenets we believe to be important.

In our experience with modeling real distributed systems, we found the following elements to be crucial to the process.

Threshold Comparisons threshold comparison is the idea that quantities are "equal enough". When dealing with dynamic quantities such as "flow" or cpu utilization, equating two values in the conventional sense (e.g. 1.00==1.01 evaluates to false) may be too unforgiving. Allowing a threshold for which values are equal (and consequently, greater than or less than) allows for small deviations to be acceptable. This is desirable in most of the comparisons one would make in modeling computer systems.

Group Equality group equality and element wise comparisons: In a distributed environment with potentially hundreds to thousands of entities, the need to make sweeping statements about how similar entities relate. A very good example would be a load balancing requirement in which one would want to ensure that a set of properties across all similar entities are equivalent. Coupled with the above threshold comparison, this can be powerful expression. Similarly, element wise comparisons allow for comparisons of each element to the same value - e.g. ensuring that all processes remain below 80.

Aggregation In large scale systems with 10,000+ monitored properties, dealing with individual property values is neither desired nor possible. Allowing developers to aggregate values increases the manageability of monitoring heuristics.

dynamic group membership: It should not be the model developer's job to sift through thousands of machines to determine which machines have been reserved for which purpose. A dynamic group membership allows the developer make assertions about components by their function and to not be bothered with the task of knowing which machines fit these roles at runtime.

While these ideas can certainly be captured by a library of methods within an existing programming language, we believe the simplicity and intuitiveness of the A language to be a great boon to model developers.

2.3 Example System, Mistake, and Program

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

One of the main applications of LVS is an advanced load balancing solution. A commonly seen LVS operator misconfiguration [14] occurs when LVS is set up in the direct routing mode with Web servers on the back-end. A popular method to achieve direct routing involves assigning the same IP address that the LVS uses to receive requests from the clients to a virtual interface of a loopback device on each Web server. This causes requests to be handled by the LVS machine, with responses handled directly by the individual Web servers. The caveat is that Web servers must ignore ARP requests for the loopback devices. Clearly, the effect of a misconfiguration here will lead to all Web servers and the load balancer answering ARP requests for the shared IP address, leading to a race condition. The manifestation of this misconfiguration is that some client requests might be sent directly to the Web servers, while others go through the load balancer, resulting in an unbalanced load on the Web servers.

Although the above is only one specific example in a long list of possible mistakes that can be introduced into a distributed system, it is a fairly representative one with several interesting qualities. Firstly, the mistake has a real, tangible impact on the distributed system. Secondly, the mistake is fairly common and simple to achieve. Finally, the impact of the mistake may or may not result in a latent error.

Figure 1 shows a small A program designed to catch mistakes when manipulating an LVS server. At a high-level, lines 1-10 type the configuration state we care about, in this case the LVS and Web server (Apache) configuration files. Lines 11-12 instantiate the elements types against real servers, and lines 13-19 are assertions to describe correct behavior. We give a brief, bottom-up description of these language constructs in the remainder of this section.

2.4 Elements

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

**Primitive types.** A supports a standard set of primitive types including int, double, and string with the typical operators.

**Stat type.** A stat object represents the tail of a stream of temporally sampled values. The CPU utilization of the Web servers are just such a value.

A stat object can be aggregated statistically into a single value via operators such as average, median, standard deviation, min, and max. Stat objects can also be compared using the standard relational operators. A statement like “All web server CPU utilization should be less than X” is thus simple to write in A. Statistical equality is calculated by using the Box and Jenkins [4] outlier model.

**Configuration and log types.** Since static configurations dictate how a majority of distributed components behave, it is natural to include configuration constructs. Each element may have an attachment of one or more objects of the configuration or log types. Each attached configuration object refers to a set of static information about the service component bound to the element. The definition of a configuration type involves the specification of a set of files or program outputs (Line 2) that can be parsed to obtain the desired static characteristics, a set of drivers that can be used to translate the configuration files into the XML format, and a set of XPath queries to extract the desired characteristics.

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

**Configuration Aggregates.** It is often useful to name an aggregate set of configuration parameters. For example, the weight of each of the workers seen by LVS is referred to as a _set_. (Line 3). A programmer can later refer to all weights using this one parameter.

### 2.5 Assertions

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

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

**Name.** A common problem with assertions directives in C and Java is that they can become opaque; sometimes even the author cannot remember the higher-level reason a particular assertion was included. It thus becomes difficult to know if the assertion or program is faulty. We hope to mitigate this effect by at least forcing programmers to name the assertions. For example, the name _balanced_ in line 13 has a simple intuitive meaning.

**Expression.** An expression returns a boolean value. If the statement evaluates to _true_, the system is correct. If the expression evaluates to _false_, the code in the action block (the _else_ clause) is evaluated. In the listing, Line 13 shows an assertion that models the balanced nature of all Web servers.

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

**Action Block.** The action block (Line 15) refers to what the runtime system should do in the event that the particular assertion has failed. We leave action, i.e. adjusting the system in response to failed assertions, as future work.

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

**Aggregates.** Relations in the expression part of an assertion can accept aggregate element types as arguments. For example, the _EQUAL_ operator can take an aggregate element (e.g., a set of replicas). The _EQUAL_ operation applied to a stat field of an aggregate element returns true if all stat objects in the field are equal, using the definition of statistical types. The _balanced_ assertion shows this usage.

### 2.6 Libraries

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

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

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

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

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

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

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

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

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

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

3. The A Runtime System

Recall the most critical job of the run-time is to reflect the real system state in the A run-time. For example, the current values of system configuration files and sampled state of running devices must be brought into a single location where the assertions are evaluated. This task is basically system monitoring, which is vast topic in and of itself. In this section we just briefly describe our current implementation.

Configuration is obtained through a series of drivers. These drivers take the vastly different configuration formats and create XML equivalents. A programs can then process these XML fragments in any way. Dynamic information is gathered by a set of monitors, which capture the dynamic state of running service components.

All information is then collected by the Integrator, and brought to a central location where the tasks and assertions are executed and evaluated. Figure 2 shows the overall architecture of our prototype.

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

A key abstraction of the runtime is the Monitoring Stream Object (MSO). There are two categories of MSOs, stat and log, which map directly to A stat and log objects.

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

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

3.1 Assertion Scheduling

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

The assertion scheduler distinguishes between two types of assertions, global assertions that have a lifetime that exceeds that of any one task, and task-specific assertions, that have lifetimes associated with specific tasks.

3.2 A Run through the A System

With a cursory understanding of the language and the runtime, it is now possible to understand how A programs are processed. Even before our A program is loaded into the runtime system, there are many things occurring. There are monitors on each machine reporting information back to the Integrator, which is keeping some historical information and doing statistical analysis.

Once our program is loaded, the runtime immediately attempts to bind the variables it encounters, namely: balancer and wsall. In the case of the former, it finds the one reporting host whose identity matches the argument given “domain.tld”. In the latter case, the runtime, matches wsall with all Web servers. These variables are now bound, and their mappings are kept as references. The assertions are then loaded into the assertion scheduler. At regular intervals, as specified in the assertion or by default values, the runtime executes an assertion by comparing the current system state as reported by the monitors on the bound machines. If the expression evaluates to false, the system will then perform the operations listed in the action block of the assertion.

If the misconfiguration listed in Section 2.3 were to be performed, the assertion listed on Line 16 would fire immediately after the configuration file was saved. If corrected, the assertion listed on Line 13 (balanced) may never have the opportunity to fire. It should be noted that the assertion is still evaluated, so even other causes that make the Web servers unbalanced would also be caught, making one assertion useful in many situations.

![Figure 2. Architecture of the current A runtime system. Shaded circles inside the service components (Web server, application servers, database server) represent monitors. Shaded circles in the directory of active monitors represent monitor callback objects. Hexagons inside the MSO Store represent monitoring stream objects (MSOs). The dynamic streaming of monitoring data to the receiving MSOs is only shown for one monitor for clarity.](image-url)
4. Evaluation

To evaluate our approach, we implemented a prototype runtime system and wrote a set of A programs to model what we believed to be correct operation in our evaluation environment - which was a three-tier auction service with a LVS load balancer front-end and a load generator as a client.

We then performed mistake injection experiments to observe how well mistakes were caught. To ensure our mistakes were of comparable complexity and subtlety to real mistakes, we used a combination of mistakes observed from human factors studies in our previous work, those reported by observations in the literature, and those reported by database administrators in the course of a survey.

The assertions found in Listing 1 represent only a small component of the libraries and programs we used to evaluate our methodologies for reducing operator mistakes. While these methods are outside the scope of this paper, we will discuss our findings in the context of our LVS case study.

We injected the mistake of allowing a Web server to answer ARP requests for its loopback device, which is the default behavior. 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. Previous techniques [18, 20] would have overlooked this mistake, due to its latent nature. However, in the interest of completeness, we decided to perform another 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 utilization should be uniform across the Web servers also failed.

Besides catching the above mistake, we were able to catch 10 of the 11 mistakes we injected - none of which could be found in previous works on validation [18]. This is a very encouraging result because it shows that a high-level understanding of the system, as encoded in our language, will catch a large fraction of unanticipated mistakes.

5. Related Work

The related works fall into three broad categories: current monitoring and correctness systems; language work for formal modeling, performance verification, and event programming; and systems level availability.

As previously mentioned, a large part of the motivation in our work are the studies done by Gray [10], and the somewhat modernized update by Oppenheimer [21]. These works investigate exactly what their respective titles would imply - how systems fail, and what measure can be done to lessen failures and their impacts. Gray looks specifically at fault tolerant systems, but states that the ratios among causes of failures is comparable to non fault tolerant or conventional systems. Of particular interest is that system administration accounted for 42% of all failures - the highest of all causes. This figure is sub divided into maintenance, operations and configuration (in order of frequency). Almost 20 years later, Oppenheimer’s work shows that not much has changed. As machines multiply and are connected by networks, the failures that occur are still largely caused by operators.

It is clear in the decades since Gray’s work, we can see that there is still much work to be done in the realm of operator caused failures. Even after identifying operators as the main source of failures, Gray offers solutions of applying hardware fault tolerant techniques of software components. He leaves the failures caused by system configuration, operations, and configuration open. Oppenheimer, on the other hand, recognizes that concentrating on operators can reduce failure rates. It is here that we have made a contribution - giving designers and operators alike, the tools to detect and prevent failures.

5.1 Previous Validation Strategies

The A language and runtime are part of a larger body of work called Model Based Validation. We briefly overview validation in this section.

The core idea of validation [18] is to verify operator actions under realistic workloads in an isolated validation environment. Mistakes can then be caught before becoming visible to the rest of the system and users.

In [18], we proposed two validation approaches: trace-based and replica-based validation. In trace-based validation, for each masked component to be validated, requests and replies passing through the shunts of an equivalent live component are logged and later replayed. During the replay, the logged replies can be compared to the replies produced by the masked component. In replica-based validation, the current offered load on the live service is used, where requests passing through the shunts of an equivalent live component are duplicated and forwarded in real-time to the validation harness to drive the masked component. The shunts also capture the replies generated by the live component and forward them to the harness, which compares them against the replies coming from the masked component.

Unfortunately, trace-based and replica-based validation are only applicable when the output of a masked component can be compared against that of a known correct instance. Many operator actions can correctly lead to a masked component behaving differently than all current/know instances, posing a bootstrapping problem. An example in the context of databases is a change to the database schema (a task that is cited as one of the most common DBA tasks in our survey). After the DBA changes the schema (e.g., by deleting a column) in the validation environment, the masked database no longer mirrors the online database and so may correctly produce different answers to the same query. The same applies to a previously collected trace. In these cases, the models created with the A and validated in A’s runtime system can help to mitigate failures caused by operator actions. This process is known as Model Based Validation.

5.2 Current Monitoring and Correctness Tools.

Assuring correct behavior of distributed systems is often assigned to an organization’s system administrators. The Internet is littered with monitoring and alerting scripts that can assist them with the monumental task of keeping a smoothly operating system. Anecdotal evidence suggests the coverage and behavior of these tools is often random and chaotic in nature. Their usage and output leave many things open to interpretation and even the smallest of organizations may have many of them cobbled together to create a suitable solution. In any event, these system administration scripts suffer from a number of problems that show them to be sub-optimal for ensuring correct behavior of distributed systems. 1) They lack continuity: the same script used to monitor and identify problems in network traffic may vastly differ from a script used to monitor CPU utilization, even though one may be tightly related to the other. 2) Manageability/Readability: Adding new alerts or monitors can involve editing multiple scripts/tools. Determining what components, alerts, monitors, exist also involves multiple scripts/tools in multiple formats. 4) Interpretation: Many tools provide the means by which information is gathered, but interpretation of this information is largely left to a person. Different people have differing opinions of correctness.
A common theme among these monitoring and correctness tools is that they, by and large, are a reactive measure. When things go wrong, there is a hurried push to see that those particular occurrences do not happen again. The models encoded in A force designers to be more proactive in insuring correct behavior in their systems, and to think at length about the properties that are important to the correct operation of their systems.

A seeks to solve these problem areas by providing a clear and concise method for system designers and administrators to formalize policies of correct behavior. The assertions in A are expressive enough to encode multi-component dependencies and simple enough to be easily and readily readable and editable. A features elements, element groups, dynamic bindings, and configuration structures—all of which make great strides in simplicity and uniformity.

When we look at our LVS case, traditional monitoring and alerting tools might used to detect excessive CPU usage in an overloaded state, or a saturated link. Usually by this time, the effects of the misconfiguration have been exposed to end users as slow response times, timeouts, or other undesirable behavior. The root cause, however, would remain hidden to administrators.

5.3 Modeling, Verification and Event Languages

Other systems have combined assertion languages with dynamic assertion checking. Both PSpec [25] and Pip [26] used assertion checking for performance debugging. While Pip focuses on distributed systems, it also seeks to verify their communication structure. Pip, like A and its associated runtime system, allows for the to declarative expression of expectations in the distributed systems. Its focus is on the communications structure, timing, and resource consumption. While A is concerned about similar features of the system, PIP’s end goal is to allow programmers to debug and explore unexpected behavior in the system.

Similar to Pip, earlier work by [25] also looked at system designers writing performance assertion for debugging and testing. PSpec includes a set of tools write and check assertions or predicates about monitoring logs. Temporal properties of distributed systems are also a central part of PSpec. PSpec specification are performance assertions grouped together with some initial declarations. The compiled assertions, along with monitored log data and a solver, evaluator, and checker come together to assist designers or system maintainers debug performance problems.

Because the above works are concerned with dynamic, run time problems as opposed to detecting operator mistakes, these systems did not consider some important static and structural issues, such as improper system configurations and latent security problems. The works described above were designed for and targeted debugging and identifying performance issues. Operator tasks are first-class concepts in our assertion language, allowing us to detect these types of problems and relate them to specific tasks.

There have also been recent languages and systems designed for temporal, event-based programming. For example, FRP [30] is designed with abstractions for temporal event-based programming. A is not as focused on abstracting temporal events handling as it is on expressing correctness about time-varying state, although it may benefit from some control over time intervals. On the systems side, Drools [12] is an event-based rule engine designed to assist enterprises in codifying their business logic. A prime focus of Drools is execution efficiency, and so it can also serve as a framework for other domain specific languages, such as A. Its use and implementation are somewhat orthogonal to our work.

Other assertion languages have been extensively explored for software debugging. For example, [8] imposes pre-conditions and post-conditions on software components. A supports a similar pre/post-conditions paradigm, but targets operator mistakes rather than software bugs.

Given that formal modeling for verification of large systems is infeasible, alternatives based on runtime monitoring have been proposed. Kim et al. [13] proposed a framework to specify the formal requirements of a program using Linear Temporal Logic, extract information from the program’s execution, and check runtime observations against the specified properties. Sammapun et al. [27] later extended this work by including the capability of verifying timinglessness to check real-time systems. Also in the realm of real-time systems, Mok and Liu [16] proposed an approach to specify timing constraints, monitor them, and catch violations.

Although both our approach and formal languages (such as Z [28]) model systems as a set of valid states separated by transition functions, A does not seek completeness; we do not try to model all possible valid states. Because the program state in A represents a running system, not a theoretical model, it is faced with observability issues that do not exist in formal languages. In general, our approach is more domain-specific and favors practicality and programmability over the provability properties favored by formalisms.

One other related avenue of research from which designers using A can benefit greatly is that of probabilistic risk assessment (PRA) [2]. When designing an assertion model, it would be most helpful to identify many of the possible failures a system can have. Our proactive method of increasing system reliability is to model correct behavior, which is made easier by knowing what aberrant behavior is. Accomplishing the steps of PRA, including Event Tree Analysis and Fault Tree Analysis, can not only give the designer more insight into the system, but also a starting point of writing A programs.

Finally, another related idea is the analysis of network security from the specification of both vulnerabilities and security policies. MuVAL [22] is a framework that uses Datalog (a subset of Prolog) to specify the security model of the components to be analyzed. As a result of the analysis, MuVAL highlights all violations. Whereas MuVAL is specialized in security analysis, our approach is more generic: the A assertions are applicable to all aspects of the system configuration (including security), as well as to dynamic system properties. We delve more into security issues in Section 6.2.

5.4 Systems Level Availability

The work described in this section is related to our work, but largely complementary. Many of the techniques can be used to strengthen the modeling process. Where appropriate and relevant, we have shown how the work can assist the system designer using A.

Flight Data Recorder [29] attempts to simplify the daunting task of analyzing and storing logs of persistant state accesses. As we have noted earlier, misconfigurations by operators account for a large number of failures. Using a domain specific log format with compression, FDR is able to quickly search and query gigabytes of logs. Common systems management queries include the ‘needle in the haystack’ variety: i.e. “Who changed this file today?” or “What was changed in this configuration file”. FDR could easily coexist on a system with the A runtime, providing a backend for assertions involving static properties. This cooperation could allow for a whole new set of assertions to be written, exposing their detail as to where failures originate.

A precursor to Pip was the concept of black-boxes in distributed system performance debugging [1]. Using passive tracing of messages, the black-box algorithms determine where delays occur in specific causal paths. With debugging, its primary goal, by the time systems observe these delays, the failures have already occurred. While this is little consolation if operator error was the
cause of said failure, we can certainly infer more information about failures using similar techniques.

A and the modeling concepts behind it are dependent on designers knowing what correct behavior is. In the rare cases when this is not true, or when designers need more assistance, online modeling can be helpful. Magpie [3] can be used to debug performance, determine capacity, tune systems, and detect anomalies. Much of the functionality requires instrumentation of components. So, if designers really need to understand these details, the effort may be worth it. However, modeling the distributed system in A, and watching the resulting assertion states may be sufficient in detecting operator error, without the additional overhead of instrumentation.

Operator Undo [5] has interesting implications if used in conjunction with an assertion based system like A. Recall that each assertion has an action block that can be used to actuate the distributed system. Operator Undo’s infrastructure and concepts of Rewind, Repair, and Replay would provide a very nice platform for systems. Using the semantics described in their work, an actuation system could be built within the A runtime. Allowing operators to undo mistakes that caused failed assertions could potentially prevent failures from worsening - especially in the absence of a validation harness [20], where rollback is inherent in the system.

A may or may not be able to give operators enough information as to why a failure has occurred. In many cases, the assertion that failed points directly to the root cause of the anomaly, but this is not guaranteed. Pinpoint [6] attempts to find faulty components through client request tracing and middleware instrumentation. Pinpoint utilizes both internal (software assertions and exceptions) and external (end to end) fault detection. Data is clustered and analyzed by a separate engine, and nodes involved in root causes of request failures are identified. We believe A can both benefit from Pinpoint’s infrastructure (for fault detection and data clustering), as well as contribute to Pinpoint’s internal fault detection.

6. Current Status and Future Work

In this paper we introduced a language for determining correct behavior in distributed systems. We overviewed the many unique constructs in the A language that can help system designers model correct behavior. We also showed the event-based and procedural components that lend themselves nicely to modeling operator tasks.

6.1 Open Issues

The question of utility of a new domain specific language versus that of traditional languages is a subjective one. More work must be done to investigate how easily A programs are written, and how effective the resulting programs are at modeling correct behavior. Our initial experiences have shown promise in a simple three tiered Internet service. Examining A in other distributed scenarios is another avenue for further work. Scalability of A programs is an open issue. Our intuition is that A programs can be modularized and then combined. This combination raises consistency and efficiency problems that might need to be resolved.

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

The second issue relates to the mistake coverage achievable by our approach. At this point, we are trying to characterize in a general manner, the classes of mistakes that are detectable based on the level of abstraction provided by A and its runtime system.

The third issue relates to the possibility of using large A programs for extensive monitoring and diagnosis of the online system. We intend to pursue this possibility, even though having a large number of assertions may result in information overload if many assertions fire at the same time. We have not seen this phenomenon with our current implementation.

The fourth issue relates to the automatic pinpointing of mistakes. Our current approach involves inspecting fired assertions manually. If many assertions fire at the same time, it may be time-consuming to identify the potential problem. However, it may be possible to develop a diagnosis system that can leverage our fired assertions in automatically pinpointing mistakes.

Finally, we left open the issue of actuation—what to do when an assertion fails. Our current prototype allows the execution of arbitrary Java code in the else block of an assertion, but currently we just print error messages. Our future work will consider taking actions to correct operator mistakes when an assertion fails.

6.2 Future Work

Besides the open issues described in the previous sections, there are many avenues to explore using our experiences with A as a foundation. We shall cover just a few of these possibilities that we believe to be promising directions.

In Section 5 we touched upon MulVAL as a framework to evaluate and reason about vulnerabilities in networked components using a rules based engine. Using input from existing vulnerability databases and system administrator knowledge, analyses can be performed on the network. Similar to how models in the A view the distributed system holistically for failures in performance, security models must also consider all system components and interactions among them. While we believe that even incompletely specified models in A can show anomalous behavior and can catch a significant number of real world failures, the security domain will undoubtedly be less forgiving. Incomplete security models can lead to unsafe systems. We hope to determine, at a minimum baseline, the level of complexity a designer generated security model would sufficient. Perhaps any insights gained from such an unforgiving field can lead to even more effective techniques in this work.

Along the same vein, security researchers sometimes struggle to identify common security metrics for systems. We propose a few ideas that can leverage our previous experience with system operators. We posit that system operators, much like they do with distributed system availability, cause a number of security vulnerabilities. Much work has been done to identifying attacks from malicious users and/or software [7, 9, 19], but perhaps a fresher view can be obtained by investigating regular (privileged or unprivileged) users, or treating malicious attackers as incompetent operators (or vice versa). One possible security metric may be how well a system fares (e.g. how many assertions fail, etc) in models created to secure systems in the presence of the aforementioned set of users.

Another open area of research for which we are well suited, at least with respect to our experiences thus far, is the field of operator guidance. Since operator error is such a large portion of all failures, any work done to make operators’ lives easier and their work less error prone would be welcomed. From the infamous animated paper clip to other more subtle works, operator guidance must walk the fine lines between accuracy, completeness, and simplicity. Failing any of these aspects creates tools with limited usefulness at best, annoying hindrances at worst. Using previous work in operator experiments and modeling operator behavior can assist in evaluating and/or designing new methods of operator guidance.

Finally, closely related to operator guidance is a stronger form of control on system operators. While guidance, many times, is merely a series of suggested actions operators should take, a
stronger form of control may require operators to take certain actions, or prevent operators from taking others - perhaps dependent on some external control model or heuristic. With these two extremes on the spectrum available to operators, a third, gentler approach arises - that of operational inertia - that is systems that are minimally self aware that construct obstacles to operator actions that have potentially disastrous effects if executed improperly. This follows the idea that actions that have dire consequences should be harder to accomplish than those that are relatively benign.

6.3 Conclusion

Ultimately, we believe that having a domain specific language like A can help to structure, organize and formalize ideas of correctly operating systems, especially in the context of operator actions. We have shown that the carefully designed constructs in A fit the domain of distributed systems well.

References