Automatically Generating Predicates and Solutions for Configuration Troubleshooting

Ya-Yunn Su
NEC Laboratories America
yysu@nec-labs.com

Jason Flinn
University of Michigan
jflinn@umich.edu

Abstract

Technical support contributes 17% of the total cost of ownership of today's desktop computers [11], and troubleshooting misconfigurations is a large part of technical support. For information systems, administrative expenses, made up almost entirely of people costs, represent 60–80% of the total cost of ownership [5]. Prior work [20, 23] has created promising tools that automate troubleshooting, thereby saving user time and money. However, these tools assume the existence of *predicates*, which are test cases painstakingly written by an expert. Since both experts and time are in short supply, obtaining predicates is difficult. In this paper, we propose a new method of creating predicates that infers predicates by observing the actions of ordinary users troubleshooting misconfigurations. We report on the results of a user study that evaluates our proposed method. The main results were: (1) our method inferred predicates for all configuration bugs studied with very few false positives, (2) using multiple traces improved results, and, surprisingly, (3) our method identified the correct *solutions* applied by users who fixed the misconfigurations in the study.

1 Introduction

Troubleshooting software misconfigurations is a difficult, frustrating job that consists of such tasks as editing configuration files, modifying file and directory access rights, checking software dependencies, and upgrading dynamic libraries. Troubleshooting is also costly: technical support represents 17% of the total cost of ownership of desktop computers [11], and administrative costs are 60–80% of the total cost of ownership of information systems [5].

Consequently, the research community has developed several systems that automate troubleshooting [20, 23]. However, these tools require *predicates*, which are test cases that evaluate the correctness of software configuration on a computer system. Unfortunately, creating predicates is currently a manual task that requires the participation of expert users with domain knowledge about the application being tested. Manual creation of predicates is time-consuming, and expert time is precious.

This paper shows how to substantially reduce the time and effort needed to deploy automated troubleshooting by *automatically* generating predicates from traces of ordinary users troubleshooting their computers. Our method is based on the insight that users who manually troubleshoot a misconfiguration generally use one or more commands to test the correctness of their system. If we can extract these commands and learn a function that maps command output to correct and incorrect configuration states, we can generate a predicate that can be used by an automated troubleshooting tool. Predicate extraction requires no extra effort on the part of the user who is troubleshooting a problem; it simply lets other users benefit from her efforts.

Our current prototype uses a modified Unix shell to observe user input and system output during troubleshooting. We chose a command line interface to simplify development and evaluation by limiting the types of input and output we need to observe. GUI applications pose additional challenges, and we discuss ways in which we might address those challenges in Section 7.1. We also use a modified Linux kernel to track the causal relationship between processes, files, and other kernel objects.

We use two heuristics to find commands that can be used as predicates. First, users generally test the system state multiple times while fixing a configuration problem: once to reproduce a configuration problem and possibly several other times to test potential solutions. Second, testing commands should generate some output that lets the user know whether or not the current configuration is correct. We therefore examine output consisting of data printed to the terminal, exit codes, and the set of kernel objects modified by a command execution. This output should be qualitatively different before and after a configuration problem is fixed. Thus, repeated commands with significant differences in output between two executions are flagged as predicates.

For example, a user could test whether a local Apache server is working correctly by using `wget` to retrieve the server's home page. When the user first runs the command, an HTTP error message is displayed. When

the user runs the command after fixing the problem, the home page is correctly displayed. Our method thus identifies the `wget` command as a predicate.

Our method also identifies potential solutions found during troubleshooting. We observe that, in general, a command that is part of a solution will causally affect the last execution of a predicate but not prior executions of the same predicate. Further, the output of the two executions of the predicate should differ because the problem is fixed during the latter execution but not during the former one. Thus, we use the causal information tracked by our modified kernel to select candidate solutions. We then sort our candidate solutions by the number of times they appear in the set of all traces. As with prior troubleshooting systems such as PeerPressure [21], we observe that the majority of users is usually right, so common solutions are more correct than uncommon ones.

We envision that our system can be deployed across a community of users of an application or an operating system. As users encounter and solve new configuration problems, our tool will collect traces of their activity, analyze the traces, and upload canonicalized and anonymized predicates and solutions to a centralized repository. Other people executing automated configuration management tools can then benefit from the experience of these prior users by downloading and trying commonly used predicates and solutions. Note that since tools such as AutoBash have the ability to try multiple solutions and deterministically roll back unsuccessful ones, our solution database does not have to identify the exact solution to a problem, just the most likely ones. This kind of community sharing of experience has been previously proposed for use with tools that prevent deadlocks [10] and enable better understanding of software error messages [8].

We evaluated the effectiveness of our method by conducting a user study in which we recorded the actions of twelve people fixing four configuration problems for the CVS version control system and the Apache Web server. We then used our method to extract predicates from these recordings. Aggregating across all traces, our method found 22 predicates that correctly tested the state of these systems, while generating only 2 false positives. Further, our method was able to correctly identify the solution that fixed each misconfiguration.

2 Related work

The research community has proposed several different approaches to automate troubleshooting. Some systems, such as Chronus [23] and AutoBash [20], search through the space of possible configurations to find a correct one. Chronus searches backwards in time to find the instance in which a misconfiguration was introduced. AutoBash applies actions previously taken to fix a problem on one computer to fix similar problems on other computers. Both systems use predicates that test software correctness to guide the search. Initially, one or more predicates evaluate to false, indicating a misconfiguration. Both systems use checkpoint and rollback to search for a state in which all predicates evaluate to true; a system in this state is assumed to be configured correctly. Using checkpoints that are created prior to executing predicates, both systems ensure predicates do not permanently affect the state of the system because execution is rolled back after the predicate completes.

Clearly, successful troubleshooting relies on having good predicates. Without predicates that distinguish correct and incorrect configurations, the automated search process will fail. Both Chronus and AutoBash previously assumed that experts would create such predicates by hand, which limits their applicability. In this paper, we show how predicates can be *automatically* generated by observing ordinary users troubleshooting configuration problems. We also show how we can simultaneously generate candidate solutions, which can be tried by systems such as AutoBash during the search.

Other automated troubleshooting tools, such as Strider [22] and PeerPressure [21], take a state-based approach in which the static configuration state on one computer is compared with that on other computers to identify significant deviations. Since a state-based approach does not need predicates, our work is not directly applicable to such systems. One of the principles guiding the design of these systems is that the majority of users (and, hence, the most frequent configuration states) are usually correct. We apply this same principle in our work to identify candidate solutions, and the results of our user study support this hypothesis.

Nagaraja et al. [15] observed that operator mistakes are an important factor in the unavailability of on-line services and conducted experiments with human operators. Like our user study, their experiments asked operators to perform maintenance tasks and troubleshoot misconfigurations for their three-tiered Internet service. However, their study did not share our goal of identifying predicates for automated troubleshooting.

The software testing research community faces a problem similar to ours, namely that manually writing test cases for software testing is tedious and time-consuming. Two solutions proposed by that community are automatic test case generation and test oracles. However, automatic test case generation [3, 4, 6, 9, 12, 17] requires a specification for the program being tested. A test oracle determines whether a system behaves correctly for test execution. Researchers also have developed ways to automate test oracles for reactive systems from specifications [18, 19] and for GUI applications using formal

models [13]. Unfortunately, generating a specification or formal model requires substantial expert participation. Since the need for such expertise is exactly what we are trying to eliminate, both methods are inappropriate for our purpose, except when such specifications and models already have been created for another purpose.

Snitch [14] is a misconfiguration troubleshooting tool that analyzes application state to find possible root causes. It uses an always-on tracing environment that records application state changes and uses exit codes as *outcome markers* to identify faulty application traces. Our work differ from Snitch in two ways. First, our work generates test cases that can be used to verify system configuration, while Snitch finds root causes of misconfigurations. Since Snitch only records application state changes, its traces do not contain sufficient information to generate predicates. In contrast, our traces record all user commands; from these commands, we select which ones can be used as predicates. Second, Snitch only uses exit codes to analyze outcomes. Our work traces the causal relationship between commands executed by the user to better understand relationships among those commands. We also use semantic information from screen output to analyze the outcome of commands.

3 Design principles

We followed three goals in designing our automated predicate extraction system.

3.1 Minimize false positives

Automatically inferring user intentions is a hard problem. It is unreasonable to expect that we can perfectly identify potential predicates and solutions in every case. Thus, we must balance false positives (in which we extract predicates that incorrectly test configuration state) and false negatives (in which we fail to extract a correct predicate from a user trace).

In our domain, false positives are much worse than false negatives. A false positive creates an incorrect predicate that could potentially prevent the automated troubleshooting tool from finding a solution or, even worse, cause the tool to apply an incorrect solution. While a false negative may miss a potentially useful test case, we can afford to be conservative because we can aggregate the results of many troubleshooting traces. With multiple traces, a predicate missed in one trace may still be identified using another trace. Thus, in our design, we bias our system toward generating few false positives, even though this bias leads to a higher rate of false negatives.

3.2 Be as unobtrusive as possible

Troubleshooting is already a tedious, frustrating task. We do not want to make it even worse. If we require users to do more work than they would normally need to do to troubleshoot their system, they will likely choose not to use our method. For instance, as described in the next section, we initially considered asking users to specify which commands they used as predicates or to specify rules that could be used to determine which commands are predicates. However, we eventually decided that these requirements were too burdensome.

Instead, our design requires only the most minimal involvement of the user. In particular, the user must start a shell that they will use to perform the troubleshooting. The shell does all the work necessary to record input, output, and causal dependencies. It then canonicalizes the trace to replace identifiers such as userids, host names, and home directories with generic identifiers. For example, the userid, `yysu`, would be replaced with `USERID`, and her home directory would be replaced with `HOMEDIR`. The traces can then be shipped to a centralized server for processing. The server might be run by a company's IT department, or it might be an open-source repository.

As an added incentive, our shell provides functionality that is useful during troubleshooting, such as checkpoint and rollback [20]. However, this added functionality is not essential to the ideas in this paper.

3.3 Generate complete predicates

Users often test software configurations with complex, multi-step test cases. For example, to test the CVS repository, a user might import a test project, check it back out again to a new directory, and use the Unix `diff` tool to compare the new and old versions.

A predicate extraction method should identify all steps in multi-step predicates. For instance, if it only detected the latter two steps, the predicate it generates would be incorrect. Applying the predicate on another computer will always lead to predicate failure since that computer will not have the test project in the repository. Asking user to identify missing steps is intrusive, violating our previous design goal. Therefore, we use a different method, causal dependency tracking, to identify missing steps. Once we identify a repeated command that has different qualitative output, we identify all prior steps on which that command depends. We refer to such commands as *preconditions*, and we include them as part of the extracted predicate.

4 A failed approach and the lessons learned

In this section, we describe our first, unsuccessful approach to inferring predicates and the lessons we learned.

Our initial idea was to record the actions of users troubleshooting misconfigurations and ask them to classify

which of the commands they had entered were predicates. We hoped to use this data as a training set for machine learning. Our goal was to develop a classifier that would correctly identify predicates from subsequent user traces. We hoped that a machine learning approach would meet our design goals by generating few false positives and not requiring any user feedback during normal operation (i.e., once the training set had been collected).

Unfortunately, this initial approach ran into two pitfalls: First, it was difficult for users to classify their own commands. Although we explained the concept of predicates to our users, it was still difficult for them to select predicates from a list of the commands that they entered. We found that users would often classify a command as a predicate only if it generated some output that was a recognizable error condition. Different users would classify the same command differently, and the same users would classify identical commands as being predicates in some cases but not in others. Thus, the training set was very noisy, making it difficult to use for machine learning.

Second, for many predicates identified by users, we found it difficult to determine which output represented success and which represented failure. For example, many users identified `ls -l` as a predicate because they would inspect a directory's contents to examine whether specific files were present with the appropriate permissions. To evaluate this predicate automatically, we would need to determine what the user expected to see in the directory. For instance, the user might be searching for a test project that he had just imported into CVS and verifying that it was not globally readable. For such commands, the evaluation of the predicate is extremely context-dependent.

These two pitfalls caused us to re-examine our approach. We observed that users often take an *action-based approach*, a *state-based approach*, or a combination of both to troubleshoot a configuration problem. An action-based approach means that the user interacts with the misconfigured application to learn the behavior of the application. For example, when troubleshooting a CVS problem, the user might `cvs import` a new module or `cvs checkout` a module. The user learns about the misconfiguration by examining the error and progress-reporting messages displayed on the screen. A state-based approach means that the user passively examines relevant state on which the application depends. For example, when troubleshooting the same CVS problem, the user would see if the CVS repository directory exists, what the CVS repository file permissions are, list the user and group information, etc.

Commands used in both action-based and state-based approaches can be used for testing. We observe that if we use commands used in an action-based approach as predicates, it is often easier to classify their return value.

	Action-based commands	State-based commands
Test system state	`wget` `cvs`	`ls` `groups`
Do not test system state	`chmod` `usermod`	read config file clear terminal screen

This table shows examples of various commands that fall into one of four possible combinations. Our methodology finds commands in the shaded quadrant.

Table 1. Command taxonomy

When the system state is incorrect, processes that execute these commands often have different exit values or display error messages. On the other hand, commands used in a state-based approach are often more difficult to evaluate because determining correctness requires semantic information or knowledge about the problem itself to reason about observable output.

This led us to develop the taxonomy in Table 1. The horizontal axis classifies commands according to how easy it is to determine their success or failure. On the left, we list commands that are action-based; these are easy to classify because they generate error messages, return different exit values, or modify a different set of objects when they fail. On the right, we list state-based commands such as `ls` that are hard to classify based on their output. The vertical axis classifies commands according to whether they are used to test system state.

We observed that only commands that fall in the top left quadrant are truly useful for automated troubleshooting. For a state-based command such as *ls*, the automated troubleshooting system will find it very hard to tell whether a difference in screen output is due to a misconfiguration or simply an inconsequential difference in system state (such as two users choosing different names for a test project).

This realization led us to a new approach. Rather than first try to identify predicates and then determine how to classify their output, we decided to instead *identify only repeated commands that have output that can be classified easily*. While this approach will miss some potential predicates, the omitted predicates are likely not useful for automated troubleshooting anyway because of the difficulty in classifying their output. Our prototype, described in the next section, uses this new approach.

5 Implementation

We first give an overview of our automated troubleshooting tool followed by our base algorithm, which we use to find single-step predicates.. All commands found by our base algorithm are action-based commands, as these commands exhibit output that our base algorithm could classify as success or failure. We then describe a

refinement to the base algorithm that allows us to also identify multi-step predicates.

5.1 Overview

We envision that when a user or system administrator encounters a configuration problem, she will launch our troubleshooting shell to fix the problem. Our shell records all the commands she types and uses the algorithms described in Sections 5.2 and 5.3 to determine which commands are predicates and solutions. The user can then save the predicates and solutions for later and share them with an online repository. To help classify predicates and solutions, the user may identify the specific application she is troubleshooting.

Later, if another user runs into a configuration problem for the same application, she can download the solutions and predicates generated by other users. Our previous work, AutoBash [20], uses speculative execution to try potential solutions. After executing each solution, AutoBash tests if the system is working by executing predicates. If all predicates evaluate to true, AutoBash declares that the configuration problem is fixed and commits the solution execution. Operating system support for speculative execution [16] enables the safe roll back of state changes made by predicates and failed solutions.

5.2 Base algorithm

Our algorithm searches for repeated commands that differ in at least two out of the following output features:

- **A zero or non-zero exit value.** When a shell creates a process to execute a command, the return value of the process, typically specified using the `exit` system call, is returned to the shell. Conventionally, a Unix command returns a non-zero exit code to indicate failure and zero to indicate success. Our troubleshooting shell simply records the exit value for each command.

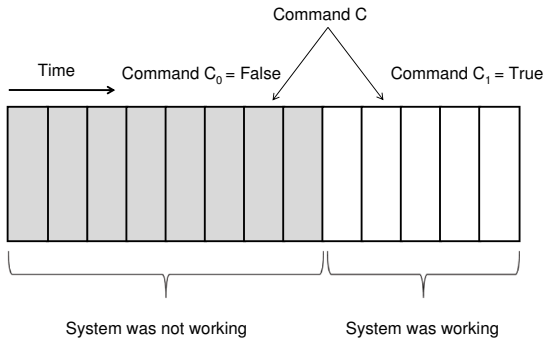
- **The presence of error messages in screen output.** Human beings are good at understanding screen output that gives feedback on the progress or result of executing a command. However, the screen output may contain unstructured text that is difficult for computers to analyze. Our hypothesis is that we only need to search for the presence of certain positive or negative keywords in the screen output to determine if a repeated command has different output. Searching for specific words is much simpler than trying to understand arbitrary screen output from the application and is often sufficient to know that two executions of a repeated command are different. Further, such semantic clues have been successfully applied in other domains to help search for programming bugs [7].

  One alternative would be to say that two commands differ if there is any difference in their screen output. However, many commands such as `date` often generate minor differences in output that do not reflect whether the command succeeded or failed.

  Our shell intercepts all screen output displayed to the user and writes it to a log file. Our implementation is very similar to the Unix `script` tool in that it uses Unix's pseudo-terminal interface to log output transparently to the user. Our base algorithm asynchronously searches for semantic clues in the log file. Currently, it only searches for the presence of the word "error" or any system error messages as defined in the file `errno.h`; e.g., "Permission denied" or "No such file or directory." When *any* error message is found in the screen output, the command is considered to contain an error message. Otherwise, the command is classified as not containing error message. Thus, two executions of the same command are considered to differ in this output feature if one generates an error message and the other does not.

- **The command's output set.** We define a command's *output set* to be the set of all files, file metadata, directories, directory entries, and processes modified by the execution of the command. Our shell uses our modified Linux kernel to trace the causal effects (output set) of command execution. Initially, only the process executing a command is in the output set. Our modified kernel intercepts system calls to observe the interaction of processes with kernel objects such as files, directories, other processes, pipes, UNIX sockets, signals, and other forms of IPC. When such objects are modified by the command's execution, they are added to the output set. When another process interacts with an object already in the output set, that process is added to the output set. For instance, if a command forks a child process that modifies a file, both the child and file are added to the output set. If another process reads the file, that process is also added to the output set. Transient objects, such as temporary files created and deleted during command execution, are not considered part of the command's output set.

  Our current implementation tracks the output set for each command line. Therefore, when a user executes a task with a sequence of commands connected by a pipe, the troubleshooting shell creates only one output set. We chose this implementation because commands connected by a pipe are causally related by the pipe, so their output sets are usually the same.



This figure shows a command C that is executed twice. The first time is shown as C_0, and the second is shown as C_1. If C_0 and C_1 have different output features, C_0 would return false and C_1 would return true.

Figure 1. The insight behind the base algorithm

> Recently, our research group developed an alternative method for tracking output sets that does not require using a modified kernel [1]. This method uses system call tracing tools such as `strace` to generate the output set in a manner similar to that described above. In the future, we plan to use this alternative method to eliminate the dependency on kernel modifications.
>
> Two executions of the same command are considered to have different output sets if any object is a member of one output set but not the other. This comparison does not include the specific modifications made to the object. For instance, the output sets of two executions of the command `touch foo` are equivalent because they both contain the metadata of file `foo`, even though the two executions change the modification time to different values.

Figure 1 shows the insight behind the base algorithm. If a repeated command has two or more different output features when executed at two different points in time, it is very likely that the command is a predicate that is being used to test system state. We hypothesize that users are likely to create a test case to demonstrate a problem, attempt to fix the problem, and then re-execute the test case to see if the fix worked. If this hypothesis is true, then the last execution of the repeated command should represent a successful test and have recognizably different output than prior, unsuccessful tests. This troubleshooting behavior was previously reported in computer-controlled manufacturing systems [2] and happened in 28 out of 48 traces in our user study described in Section 6.

Based on this observation, we say that a predicate evaluates to true (showing a correct configuration) if two or more output features match those in the final execution of the command. If two or more differ, we say the predicate evaluates to false and shows a misconfiguration. Note that some predicates that return a non-zero exit code and generate an error message should still evaluate to true. For instance, a user may be testing whether an unauthorized user is able to access files in a CVS repository. In this case, an error message actually indicates that the system is configured correctly.

Identifying repeated commands as the same command based on strict string comparison is somewhat problematical because it may leave out critical information. First, a command may be executed as different users; e.g., once as an ordinary user and once as root. This can lead to substantially different output even if a misconfiguration was not corrected between the two executions. Second, the same command may be executed in two different working directories, leading to markedly different output.

We solved this problem by having our shell record the userid and working directory for each command. If either of these variables differ, we consider two commands to be different. This is a conservative approach; e.g., the working directory has no effect on the value returned by `date`. While our conservative approach may miss good predicates, it is consistent with our design goal of preferring false negatives to false positives.

Some known environment variables, such as the parent process id, are ignored by our shell because we have judged them extremely unlikely to influence command execution. However, unknown environment variables introduced during troubleshooting are always recorded and used to differentiate commands. It may be that further experience will allow us to refine our heuristic for which environment variables to exclude. However, our evaluation in Section 6 shows that comparing only the userid, working directory, and any new environment variables introduced during troubleshooting generated many correct predicates. Further, no false positives were generated due to environment variables excluded from the comparison.

5.3 Finding preconditions

The base algorithm is very effective in identifying single-step predicates that test system state. However, we noticed that in some cases a predicate relies on some prior commands to change the system state before it can work correctly. We call these prior commands on which a predicate depends *preconditions*. A predicate that relies on preconditions cannot be used without those preconditions by automated troubleshooting tools as it would always return the same result. For example, if a user fixed a CVS problem starting from an empty repository and the base algorithm identified `cvs checkout` to be a predicate, this predicate would work only if the command `cvs import` had first been executed. The `cvs import` com-

mand is a precondition for that predicate because, without it, the predicate will always return false.

To find preconditions, we first search for commands that have causal effects on a predicate. We identify causal relationships by tracking the output set of each command, even after that command terminates. If a subsequent command becomes a member of a prior command's output set, then we say that the prior command has a causal effect on the latter one. For instance, the command `echo hi > foo` would have a causal effect on the subsequent command `cat foo`.

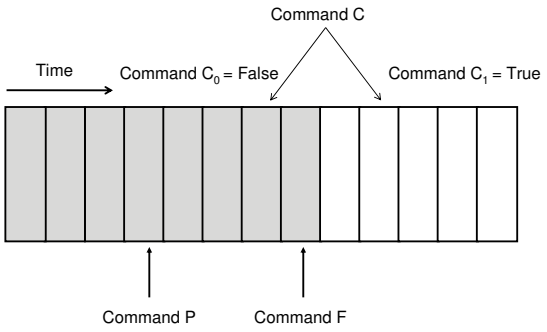
Besides including prior commands that have causal effects on predicates, we also include prior commands that add or remove environment variables. Since the base algorithm compares environment variables when comparing commands, adding or removing environment variables is considered to have an effect on all subsequent commands. This is more conservative than strictly needed, as environment variables could be added but not read by some or all later commands executed in the same shell. However, identifying such situations requires application-specific semantic knowledge.

Not all prior commands with causal effects on predicates are preconditions. If the user has successfully fixed the configuration problem, the commands comprising a solution will have a causal effect on the last execution of the predicate. Including a solution as a precondition would cause a predicate to always return true, rendering the predicate ineffective. Note that even if a solution embedded in a predicate inadvertently fixes a problem, the effects are undone when the automated troubleshooting system rolls back system state after predicate execution.

We differentiate between preconditions and solutions by first finding all commands that causally affect all executions of a predicate. Within these commands, the heuristic uses two rules to determine if a command is a precondition or a solution. First, a command that has causal effects on both successful and failed predicates is a precondition. Second, a command that only has causal effects on successful predicates and is executed chronologically after all failed predicates is a solution.

Figure 2 shows the insight that allows the second rule to distinguish between a solution and a precondition. The insight is that users will usually first make sure preconditions are executed before trying to run a test case that reproduces a misconfiguration (since the test can not succeed without the precondition). Therefore, commands that occur after failed predicates and before successful ones are likely to be solutions.

Consider an example in which both user1 and user2 are authorized to access a CVS repository, but only user 1 is in the CVS group. Figure 3 shows the four commands the user executes and the causal relationship be-



This figure demonstrates how the precondition heuristic works. Assume the base algorithm determines that C_0 returns false and C_1 returns true. If both a command P and a command F have causal effects on C_1 only, the precondition heuristic determines that the command P is a precondition for command C and the command F is a solution. The reason is that command F is likely the solution that causes the state transition that causes C_0 and C_1 to return different results.

Figure 2. The insight behind the precondition heuristic

tween commands. First, the base algorithm would determine that "`cvs co` as user2" is a predicate. The first predicate execution is considered to return false and the second one is considered to return true. Both the "`cvs import` as user1" command and the "`usermod -G CVSgroup user2`" command, which adds user2 to CVS group, have causal effects on the second predicate execution. The precondition uses the chronological order to determine that "`cvs import` as user1" is a precondition and "`usermod -G CVSgroup user2`" is a solution.

However, our heuristic is not foolproof. If a precondition is executed after failed predicates, it has the same causal effect as a solution. Thus, it is difficult to determine whether it is a precondition or a solution without the command semantics. Consider the same example, but with the order of "`cvs import` as user1"and "`cvs co` as user2" reversed, as shown in Figure 4. Both "`cvs import` as user1" and "`usermod -G CVSgroup user2`" have causal effects on the successful predicate and were executed after the failed predicate. Lacking semantic information about CVS, the precondition heuristic would mark both commands as part of the solution.

We believe that the second scenario is less likely to occur than the first because a user will typically set up the preconditions before executing a test case. In our user study, this heuristic worked in every case. However, we can also filter out incorrect solutions if they occur less frequently than correct ones, as described in the next section.

Our heuristic can work correctly even if a solution requires multiple commands. Consider a solution that requires two commands A and B, both of which occur chronologically after all failed predicates. Since both A and B have causal effects on the predicate, our heuristic

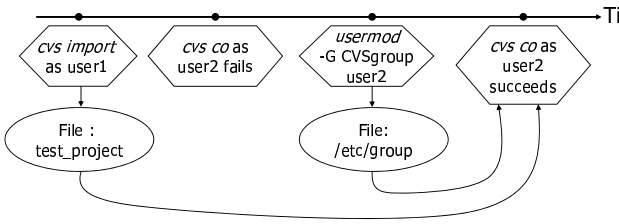


Figure 3. Precondition heuristic example

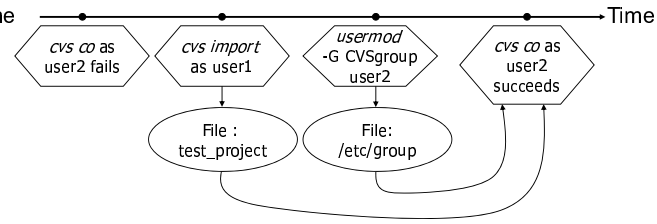


Figure 4. Example of the precondition heuristic failing

would identify both as part of the solution.

5.4 Inferring solutions from user traces

Not all solutions identified by the precondition heuristic are the correct solution for the specific configuration problem the user is solving. Sometimes, a user does not fix the problem. Other times, a user fixes the problem incorrectly. To filter out such erroneous solutions, we rely on the observation made by PeerPressure [21] and other troubleshooting systems that the mass of users is typically right. Thus, solutions that occur more frequently are more likely to be correct.

We therefore rank solutions by the frequency that they occur in multiple traces. Effectively, solutions are ranked by their popularity; a solution is considered more popular if more users apply it to successfully fix their configuration problems. In our evaluation, this heuristic worked well because the most popular solutions found by users were the ones that solved the actual configuration problems that we introduced.

Counting the frequency of a solution's popularity is complicated by the fact that different commands have the same effect. For example, when users in our study solved an Apache configuration problem caused by the execution bit on a user's home directory (e.g., /home/USERID), being set incorrectly, their solutions included `chmod 755 /home/USERID`, `chmod 755 USERID/`, and `chmod og+rx USERID`. Although these commands were syntactically different, they all had the same effect.

To better measure the popularity of a solution, we group solutions by their *state deltas*. The state delta captures the *difference* in the state of the system caused by the execution of a command by calculating the difference for each entity in the command's output set. For example, the state delta for command `chmod a+r test.pl` includes only the change in file permissions for test.pl. The solution-ranking heuristic first groups solutions based on state deltas so that all solutions having the same state delta are in the same group. It then ranks the groups by their cardinality.

	CVS configuration problems
1	Repository not properly initialized
2	User not added to CVS group
	Apache Web server configuration problems
1	Apache cannot search a user's home directory due to incorrect permissions
2	Apache cannot read CGI scripts due to incorrect permissions

Table 2. Description of injected configuration problems

Here is an example of how the solution-ranking heuristic ranks the three solutions: `chmod 755 /home/USERID`, `chmod 755 USERID/`, and `chmod -R 777 USERID/`. The the first two commands are placed into one group because they both have the same state delta (changing the permission of the directory /home/USERID to 755). The third solution is put in a separate, less popular group.

6 Evaluation

To evaluate our methods for extracting predicates and solutions, we conducted a user study in which participants were asked to fix four configuration bugs: two for the CVS version control system and two for the Apache web server. These bugs are shown in Table 2. While these bugs may seem relatively simple, several users were unable to solve one or more of these bugs in the allotted time (15 minutes). We initially designed a user study with a few more complex bugs, but our initial feedback from trial users showed that the complex bugs were too hard to solve in a reasonable amount of time. We also decided to impose the 15-minute limit based on our trial study. Since we ask each user to solve a total of six problems (including two sample problems) and the study required some up-front explanation and paperwork, users needed to commit over two hours to participate. Even with the 15-minute limit, one user declined to complete all problems in the study.

6.1 Methodology

A total of twelve users with varying skills participated in our study: two experienced system administrators and

User	CVS version control			Apache Web server		
	Experience level	Prob 1 Fixed?	Prob 2 Fixed?	Experience level	Prob 1 Fixed?	Prob 2 Fixed?
A	Expert	N/A	Y	Expert	Y	Y
B	Novice	Y	Y	Intermediate	N	Y
C	Intermediate	Y	Y	Novice	Y	Y
D	Expert	Y	Y	Expert	Y	Y
E	Beginner	N	N	Expert	Y	N
F	Intermediate	Y	Y	Expert	Y	Y
G	Novice	Y	N/A	Beginner	N	N
H	Intermediate	Y	Y	Expert	Y	Y
I	Intermediate	Y	Y	Expert	Y	Y
J	Novice	Y	Y	Expert	Y	Y
K	Expert	Y	N	Intermediate	N	Y
L	Intermediate	Y	Y	Novice	N	Y
Total fixed		10	9		8	10

This table shows the experience of our participants and the number of problems solved.

Table 3. Participant summary

ten graduate students, all from the University of Michigan. We asked participants to assess their experience level. Table 3 is a summary of our participants. For CVS, three participants (A,D, and K) rated themselves as experts, meaning that the participant had diagnosed and fixed misconfigurations several times for that application. Five participants (C, F, H, I, and L) rated themselves as intermediates, meaning that the participant had fixed configurations for that application at least once, and three participants (B, G, and J) rated themselves as novices, meaning that the participant had used the application. Two participants (user E for CVS and user G for Apache) were listed as beginners because they were unfamiliar with the application. Both beginner participants were not asked to complete that portion of the study. For Apache, seven participants were experts (A, D, E, F, H, I, and J), two were intermediates (B and K), and two were novices (C and L). Additionally, user A did not complete CVS bug 1 because of an error we made in setting up the environment, and user G declined to do CVS problem 2.

For each application, each participant was first given a sample misconfiguration to fix. While fixing the sample, they could ask us questions. The participant then was asked to fix two additional misconfigurations without any guidance from us. We did not tell the participants any information about the bugs, so they needed to find the bug, identify the root cause, and fix it. Participants were given a maximum of 15 minutes to fix each problem. For each problem, 8–10 participants were able to fix the bug, but 1–3 were not.

For each task, our modified shell recorded traces of user input, system output, and the causality data (output sets) tracked by the kernel. The overhead of tracking the output set was shown to be negligible in our prior work [20]. We implemented the algorithms to analyze these traces in Python. The computational cost is small; the time to analyze all traces averaged only 123 ms on a desktop computer with a 2.7 GHz processor and 1 GB memory. The shortest trace required 3 ms to analyze and the longest took 861 ms.

6.2 Quality of predicates generated

Table 4 and Table 5 summarize the predicates generated for each application. Each table lists the number of predicates extracted, whether the extracted predicates were correct, and the total number of commands entered by the user for that trace.

For both CVS bugs, we generated four correct predicates from ten traces, with no false positives. For Apache, we generated six correct predicates for the first bug and eight for the second, with one false positive for each bug. We are pleased that the results matched our design goal of minimizing false positives, while still extracting several useful predicates for each bug.

6.2.1 Predicates generated for CVS problem 1

The first CVS problem was caused by the CVS repository not being properly initialized, and the solution is to initialize it using `cvs init`. Table 6 lists the predicates we extracted. We break each predicate into two parts: the predicate (Pred), the repeated command that determines if this predicate returns true or false, and the precondition (Precond), the set of commands that need to be executed before the predicate. We also list the solution (Sol) identified by our methodology. All predicates

User	Predicate	
B	Precond	`export CVSROOT="/home/cvsroot"`
	Pred	`cvs import -m "Msg" yoyo/test_project yoyo start`
	Sol	`cvs -d /home/cvsroot init`
F	Precond	`export CVSROOT=/home/cvsroot`
	Pred	`cvs import test_project`
	Sol	`cvs -d /home/cvsroot init`
K	Pred	`cvs -d /home/cvsroot import cvsroot`
	Sol	`cvs -d /home/cvsroot init`
L	Pred	`cvs -d "/home/cvsroot" import test_project`
	Sol	`cvs -d /home/cvsroot init`

Each predicate contains one or more steps. The last step of a predicate is listed as Pred, and the remaining steps are listed as Precond. We also list the solution (Sol) identified for each predicate.

Table 6. Correct predicates for CVS problem 1

User	CVS problem 1			CVS problem 2		
	# of Pred	Correct?	Total # of cmds	# of Pred	Correct?	Total # of cmds
A	—	—	—	1	Yes	44
B	1	Yes	105	1	Yes	44
C	0	SBA	57	0	NRC	46
D	0	SBA	49	0	SBA	26
F	1	Yes	22	1	Yes	30
G	0	NRC	61	—	—	—
H	0	NRC	58	0	NRC	74
I	0	NRC	18	1	Yes	18
J	0	NRC	65	0	SBA	72
K	1	Yes	55	0	DNF	24
L	1	Yes	40	0	SBA	24

There are three reasons why no predicate was identified in some of the above traces: (1) NRC means that the user did not use a repeated command to test the configuration problem. (2) DNF means that the user did not fix the configuration problem. (3) SBA means that the user used a state-based approach.

Table 4. Summary of predicates generated for CVS

User	Apache problem 1			Apache problem 2		
	# of Pred	Correct?	Total cmds.	# of Pred	Correct?	Total cmds.
A	1	Yes	45	1	Yes	45
B	0	DNF	39	0	NRC	39
C	2	Yes Yes	41	1	Yes	41
D	1	Yes	68	1	Yes	68
E	0	NRC	35	1	No	35
F	0	NRC	33	1	Yes	33
H	1	Yes	32	1	Yes	32
I	1	Yes	22	1	Yes	22
J	0	NRC	40	1	Yes	40
K	1	No	67	1	Yes	67
L	0	DNF	55	0	NRC	55

There are two reasons why no predicate was identified in some of the above traces: (1) NRC means that the user did not use a repeated command to test the configuration problem. (2) DNF means that the user did not fix the configuration problem.

Table 5. Summary of predicates generated for Apache

that we identified involve the user importing a module into the CVS repository.

We did not identify predicates for participants C and D because they used a state-based approach in which they discovered that the CVS repository was not initialized by examining the repository directory. Participants G, H, I, and J used slightly different commands to test the system state before and after fixing the configuration problem, so our algorithm did not identify a predicate. For example, participant H's two `cvs import` commands had different CVS import comments.

6.2.2 Predicates generated for CVS problem 2

The second CVS bug was caused by a user not being in the CVS group, and the solution is to add that user to the group. Table 7 shows the predicates and solutions identified for this bug. All extracted predicates involve the user trying to check out a module from the CVS repository. These are multi-step predicates in which the checkout is proceeded by a CVS import command.

No predicate was generated for six traces. Participants C and H did not use repeated commands. For instance, participant C specified the root directory as /home/cvsroot in one instance and /home/cvsroot/ in another. Participant H used the same command line but with different environment variables. Our algorithm was unable to identify a predicate for participant K because that user did not fix the problem. Finally, no predicate was found for participants D, J, and L because they used a state-based approach (for example, participant D executed `groups` and participant L examined /etc/group).

6.2.3 Predicates generated for Apache problem 1

The first Apache bug was caused by Apache not having search permission for the user's home directory, and the solution is to change the directory permissions. Table 8 shows the correct predicates identified for each trace and the corresponding solutions. Almost all predicates download the user's home page. Some preconditions found by the precondition heuristic are not required for the predicate to work but also do not affect the correctness of the predicates. Predicate C-2, `apachectl stop` did not seem to be a correct predicate, so we examined why it was generated. We found that predicate C-2 successfully detected an error in the Apache configuration file introduced by participant C. `apachectl` is a script that controls the Apache process. It does not stop

User	Predicate	
A	Precond	`cvs import test_project head start`
		`cvs co test_project`
		`export CVSROOT=/home/cvsroot`
	Pred	`cvs co test_project`
	Sol	`vi /etc/group`
B	Precond	`cvs -d /home/cvsroot import yoyo/test_project`
	Pred	`cvs -d /home/cvsroot checkout yoyo/test_project`
	Sol	`usermod -G cvsgroup USERID`
F	Precond	`cvs import test_project`
		`cvs co test_project`
		`export CVSROOT=/home/cvsroot`
	Pred	`cvs co test_project`
	Sol	`vim group`
I	Precond	`cvs -d /home/cvsroot import test_project`
	Pred	`cvs -d /home/cvsroot co test_project`
	Sol	`vi /etc/group`

Each predicate contains one or more steps. The last step of a predicate is listed as Pred, and the remaining steps are listed as Precond. We also list the solution (Sol) identified for each predicate.

Table 7. Correct problems for CVS problem 2

User	Predicate	
A	Precond	`wget http://localhost`
		`chmod 777 index.html`
		`chmod 777 public_html/`
	Pred	`wget http://localhost/~USERID /index.html`
	Sol	`chmod 777 USERID`
C-1	Pred	`wget http://localhost/~USERID/`
	Sol	`chmod o+rx /home/USERID`
C-2	Pred	`apachectl stop`
	Sol	`vim /etc/httpd/conf/httpd.conf`
		`chmod o+rx /home/USERID`
D	Precond	`wget localhost`
		`chmod -R 777 public_html/`
	Pred	`wget localhost/~USERID`
	Sol	`chmod -R 777 USERID/`
H	Precond	`mkdir scratch`
		`wget http://localhost`
		`rm index.html`
	Pred	`wget http://localhost/~USERID`
	Sol	`chmod 755 /home/USERID/`
I	Precond	`wget http://localhost`
	Pred	`wget http://localhost/~USERID`
	Sol	`chmod 755 /home/USERID/`

Each predicate contains one or more steps. The last step of a predicate is listed as Pred, and the remaining steps are listed as Precond. We also list the solution (Sol) identified for each predicate.

Table 8. Correct predicates for Apache problem 1

User	Predicate	
A	Pred	`wget http://localhost/cgi-bin/test.pl`
	Sol	`chmod 755 test.pl`
C	Pred	`wget http://localhost/cgi-bin/test.pl`
	Sol	`chmod go+r test.pl`
D	Precond	`wget localhost/cgi-bin/test.pl`
		`vi /var/www/cgi-bin/test.pl`
	Pred	`wget localhost/cgi-bin/test.pl`
	Sol	`vi /var/www/cgi-bin/test.pl`
		`chmod 777 /var/www/cgi-bin/test.pl`
F	Pred	`wget http://localhost/cgi-bin/test.pl`
	Sol	`chmod 755 test.pl`
H	Precond	`mkdir scratch`
	Pred	`wget http://localhost/cgi-bin/test.pl`
	Sol	`chmod 755 /var/www/cgi-bin/test.pl`
I	Pred	`wget http://localhost/cgi-bin/test.pl`
	Sol	`chmod 755 /var/www/cgi-bin/test.pl`
J	Pred	`wget 127.0.0.1/cgi-bin/test.pl`
	Sol	`chmod +r test.pl`
K	Pred	`wget http://localhost/cgi-bin/test.pl`
	Sol	`chmod a+r test.pl`

Each predicate contains one or more steps. The last step of a predicate is listed as Pred, and the remaining steps are listed as Precond. We also list the solution (Sol) identified for each predicate.

Table 9. Correct predicates for Apache problem 2

the Apache process if an error is detected in the configuration file. Thus, this command is indeed a valid predicate.

No predicates were generated for some participants due to reasons similar to those seen in CVS traces. Participants B and L did not fix the problem. Participants E and F executed a command in different directories, so the base algorithm considered that command as not repeated.

One false positive was generated for participant K. Since participant K did not fix the problem, we first expected no predicate be generated. However, this participant edited the Apache configuration file multiple times using `emacs`. The file contains an error message that was displayed only in some executions of `emacs` in which the participant scrolled down. Additionally, the participant sometimes modified the file and other times did not, leading to different output sets. Since two output features changed, our heuristic labeled this command as a predicate.

6.2.4 Predicates generated for Apache problem 2

The second Apache problem was caused by Apache not having read permission for a CGI Perl script and the solution is to change the file permission. The initial state included a default CGI Perl script in Apache's default

	Solution	Freq.
1	`cvs -d /home/cvsroot init` as cvsroot	3
2	`vi /etc/group` as root	3
3	`cvs -d /home/cvsroot init` as root	1
4	`usermod -G cvsgroup USERID` as root	1

Table 10. Solutions ranked for CVS

	Solution	Freq.
1	`chmod 755 /var/www/cgi-bin/test.pl`	6
2	`chmod 755 /home/USERID` as root	2
3	`chmod 777 USERID` as root	1
4	`chmod o+rx /home/USERID` as root	1
5	`chmod -R 777 USERID/` as USERID	1
6	`vim /etc/httpd/conf/httpd.conf` as root	1
7	`chmod 777 /var/www/cgi-bin/test.pl`	1
8	`chmod +r test.pl` as root	1
9	`vi /var/www/cgi-bin/test.pl`	1

Table 11. Solutions ranked for Apache

CGI directory, /var/www/cgi-bin. Table 9 shows the correctly identified predicates, which all involve retrieving the the CGI script. Some traces include preconditions that are not really required, but these preconditions do not affect the correctness of the predicates.

No predicate was generated for participants B and L because they executed `wget` in different directories. The predicate identified for participant E was incorrect. Participant E did not fix the problem. However, participant E used `links` to connect to the Apache Web server. This utility generates files with random names so each invocation has a different output set. Combined with an error message being generated in some instances but not others, this led our heuristic to falsely identify a predicate. Based on our two false positives, we believe it would be beneficial to ask users if the problem was fixed at the end of troubleshooting and not try to generate predicates if it was not. This question may not be too intrusive and would eliminate both false positives.

6.3 Solution ranking results

We took the solutions found by the precondition heuristic from all traces and used our solution-ranking heuristic to rank them by frequency. Table 10 shows results for CVS. The two highest ranked solutions are the correct fixes for the two problems we introduced. Note that the solutions are captured as a state delta, so a command that starts an editor (vi) is really a patch to the edited file. The third solution listed is less correct than the first because it creates the repository as root, giving incorrect permissions. The final solution is as correct as the second for fixing CVS problem 2.

Table 11 shows results for Apache. The two highest ranked solutions fix Apache problems 2 and 1, respectively. Solution 3-5 are less correct than solution 1 because they grant more than the minimum permission required. The 6th solution correctly solves the bug introduced by participant C for Apache problem 1. The remaining 3 solutions are less correct than solution 1 because they either give too much permission or do not fix the problem.

7 Discussion

We first discuss the challenges we encountered and how we would like to address them, followed by discussing the limitations of our approach.

7.1 Challenges

One challenge we encountered relates to canonicalization. Our troubleshooting shell canonicalizes common environment variables, such as home directories and user names. However, applications may also use temporary files, specific file location settings, or other environment variables. More thought may be required on how to handle application-specific variables if application semantic knowledge is not presented.

Our current study simplifies the configuration problem by restricting user input to text form (i.e., by requiring the activity to occur within the scope of a Unix shell). We chose this approach to speed the implementation of our prototype. Writing tools to capture text input and output is easier than writing tools to capture graphical interactions.

We sketch here how we would extend our current approach to handle GUI applications. First, a command roughly maps to an action in the command line interface. However, users launch a GUI application to explore many different configuration actions more flexibly, which makes it hard to find repeated tasks for our base algorithm. Without user input, it is very difficult to break a long GUI session into individual actions that are more likely to repeat. Second, if the user happens to execute one action at a time using a GUI application, we need a more sophisticated way to identify if two GUI sessions are the same. One possible solution is to use state deltas to capture the effect of performing GUI applications and compare such deltas. We would capture output features as follows:

- **Exit value** Since GUI applications are designed to execute several actions, they usually do not return the proper value to the calling shell.
- **Screen output** GUI configuration tools may offer additional semantic information. For instance, error dialogs are a common widget that indicate the failure of an operation. Such dialogs can be queried using the GUI's accessibility APIs (intended to benefit vision-impaired users).
- **Output set** We could capture the output set for a GUI application in the same way as a command.

7.2 Limitations

Our troubleshooting shell assumes that all configuration actions happen under its purview. Configuration problems involving external components, such as printer or network communication, are not handled by our shell because it does not have the ability to track external components' output. Also, one can imagine a precondition command executed before our troubleshooting shell is launched; our shell will not find that precondition as it limits dependency tracking to the period in which it is running.

8 Conclusion

Predicates play an important role for automated configuration management tools such as AutoBash and Chronus. However, writing predicates by hand is tedious, time-consuming, and requires expert knowledge. This work solves the problem of manually writing predicates by automatically inferring predicates and solutions from traces of users fixing configuration problems.

Acknowledgments

We thank our shepherd, Yinglian Xie, and the anonymous reviewers for valuable feedback on this paper. The work has been supported by the National Science Foundation under award CNS-0509093. Jason Flinn is supported by NSF CAREER award CNS-0346686. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, the University of Michigan, or the U.S. government.

References

[1] ATTARIYAN, M., AND FLINN, J. Using causality to diagnose configuration bugs. In *Proceedings of the USENIX Annual Technical Conference* (Boston, MA, June 2008), pp. 171–177.

[2] BEREITER, S., AND MILLER, S. *Troubleshooting and Human Factors in Automated Manufacturing Systems*. Noyes Publications, March 1989.

[3] BOYAPATI, C., KHURSHID, S., AND MARINOV, D. Korat: Automated testing based on Java predicates. In *Proceedings of ACM International Symposium on Software Testing and Analysis ISSTA 2002* (2002).

[4] CHOW, T. S. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering 4*, 3 (1978), 178–187.

[5] COMPUTING RESEARCH ASSOCIATION. Final report of the CRA conference on grand research challenges in information systems. Tech. rep., September 2003.

[6] DALAL, S. R., JAIN, A., KARUNANITHI, N., LEATON, J. M., LOTT, C. M., PATTON, G. C., AND HOROWITZ, B. M. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)* (Los Angeles, California, May 1999).

[7] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Banff, Canada, October 2001), pp. 57–72.

[8] HA, J., ROSSBACH, C. J., DAVIS, J. V., ROY, I., RAMADAN, H. E., PORTER, D. E., CHEN, D. L., AND WITCHEL, E. Improved error reporting for software that uses black-box components. In *Proceedings of the Conference on Programming Language Design and Implementation 2007* (San Diego, CA, 2007).

[9] HAREL, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (1987), 231–274.

[10] JULA, H., TRALAMAZZA, D., ZAMFIR, C., AND CANDEA, G. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation* (San Diego, CA, December 2008).

[11] KAPOOR, A. Web-to-host: Reducing total cost of ownership. Tech. Rep. 200503, The Tolly Group, May 2000.

[12] MARINOV, D., AND KHURSHID, S. Testera: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)* (San Diego, CA, November 2001).

[13] MEMON, A. M., POLLACK, M. E., AND SOFFA, M. L. Automated test oracles for GUIs. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, 2000), pp. 30–39.

[14] MICKENS, J., SZUMMER, M., AND NARAYANAN, D. Snitch: Interactive decision trees for troubleshooting misconfigurations. In *In Proceedings of the 2007 Workshop on Tackling Computer Systems Problems with Machine Learning Techniques* (Cambridge, MA, April 2007).

[15] NAGARAJA, K., OLIVERIA, F., BIANCHINI, R., MARTIN, R., AND NGUYEN, T. Understanding and dealing with operator mistakes in Internet services. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 61–76.

[16] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.

[17] OFFUTT, J., AND ABDURAZIK, A. Generating tests from uml specifications. In *Second International Conference on the Unified Modeling Language (UML99)* (Ocrober 1999).

[18] RICHARDSON, D. J. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)* (Seattle, WA, 94), pp. 138–153.

[19] RICHARDSON, D. J., AHA, S. L., AND O'MALLEY, T. O. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering* (Melbourne, Australia, 1992), pp. 105–118.

[20] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (Stevenson, WA, October 2007), pp. 237–250.

[21] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 245–257.

[22] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the USENIX Large Installation Systems Administration Conference* (October 2003), pp. 159–172.

[23] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 77–90.