ICSE: G: Fighting Software Configuration Errors with Combined Static and Dynamic Program Analyses

Sai Zhang
Department of Computer Science & Engineering
University of Washington
szhang@cs.washington.edu

ABSTRACT

This paper presents ConfDiagnoser, a general technique for diagnosing software configuration errors. ConfDiagnoser identifies the root cause of a configuration error — a single configuration option that can be changed to produce desired behavior. It uses user demonstration and combined static and dynamic program analyses to link the undesired behavior to specific configuration options. ConfDiagnoser differs from existing approaches in four key aspects: (1) it does not require users to provide a testing oracle (to check whether the software functions correctly) and thus is fully-automated; (2) it can diagnose both crashing and non-crashing errors; (3) it does not require any OS-level support; and (4) it is cognizant of software evolution and can diagnose configuration errors introduced during software evolution.

We demonstrated ConfDiagnoser's accuracy and speed on 22 real-world software configuration errors from 7 configurable software systems. For 17 errors, the root cause was one of ConfDiagnoser's top 3 suggestions. Overall, ConfDiagnoser produced significantly better results than existing techniques. ConfDiagnoser runs in just a few minutes, making it an attractive alternative to manual debugging.

1. PROBLEM

The behavior of a configurable software system often depends on how that system is configured. Software configuration errors are errors in which the software code and the input are correct, but the an incorrect value is used for a configuration option so that the software does not behave as desired. Such errors may lead the software to terminate unexpectedly (i.e., crashing errors), or simply produce erroneous output (i.e., non-crashing errors).

In practice, software configuration errors are *prevalent*, *severe*, and *hard to debug*, but they are *actionable* for users to fix.

Prevalent. Technical support contributes 17% of the total cost of ownership of today's software, and troubleshooting misconfigurations is a large part of technical support [12]. A recent analysis of Yahoo's mission-critical Zookeeper service showed that software misconfigurations accounted for the majority of all user-visible failures [21]. Configuration-related issues caused about 31% of all failures at a commercial storage company [31]. The vast majority of production failures at Google arise not due to bugs in the software, but bugs in the configuration settings (i.e., configuration errors) that control the software [5].

Severe. Configuration errors can have disastrous impacts. For example, an outage in Facebook due to an incorrect configuration value left the website inaccessible for about 2 hours [4]. The entire `.se` domain of Sweden was unavailable for about 1 hour, due to a DNS misconfiguration problem [20]. A misconfiguration made Microsoft's public cloud platform, Azure, unavailable for about two and a half hours [15]. Each such incident affected millions of users.

Hard to debug. Configuration errors are difficult to diagnose. Even when a software system outputs an error message, it is often cryptic or misleading [3, 7, 30, 31]. Users may not even think of configuration as a cause of their problem. A configuration error usually requires great expertise to understand the error root causes. For example, a configuration error in the CentOS kernel prevented a user from mounting a newly-created file system [31]. The user needed deep understanding about the exhibited symptom, and had to re-install kernel modules and also modify configuration option values in several places to get it to work. Techniques to help escape from "configuration hell" are highly demanded [5].

Actionable. Unlike software bugs, which can only be fixed by experienced software developers, fixing a software configuration error is *actionable* for software end-users or system administrators. These users are not the software developers, and cannot access (much less understand) the source code; but they can fix a configuration error by simply changing the values of certain configuration options.

2. MOTIVATING EXAMPLES

We next describe two real scenarios in which we used ConfDiagnoser to resolve software configuration errors.

Configuration Errors in a Single Software Version.

Randoop [17] is a popular automated test generation tool developed and maintained in the PLSE group at the University of Washington (`http://www.cs.washington.edu/research/plse`). We received a "bug report" against Randoop, from a testing expert who had been using Randoop for quite a while. The "bug report" indicated that Randoop terminated normally but failed to generate tests for the NanoXML [16] program.

Although the reported problem is deterministic and fully reproducible, it is a silent, non-crashing failure and is challenging to diagnose. Differing from a crashing error, Randoop did not exhibit a crashing point, dump a stack trace, output an error message, or indicate suspicious program variables that may have incorrect values. Lacking such information makes many techniques such as dynamic slicing [37], dynamic information flow tracking [3], and failure trace analysis [18] inapplicable. In addition, for this scenario, the person who reported the bug had already minimized the bug report: if any part of the configuration or input is removed, Randoop either crashes or no longer exhibits this error. This further makes search-based fault isolation techniques such as delta debugging [32] ineffective.

In fact, this bug report does not reveal a real bug in the Randoop code. Its root cause is that the user failed to set one configuration option. Despite the simplicity of the solution, to the best of our knowledge, no previous configuration error diagnosis technique [2, 3, 18, 26, 29, 36] can be directly applied.

Our ConfDiagnoser technique can diagnose and correct this problem. We first reproduced the error in a ConfDiagnoser-instrumented

```
Suspicious configuration option: maxsize

It affects the behavior of predicate:
"newSequence.size() > GenInputsAbstract.maxsize"
(line 312, class: randoop.ForwardGenerator)

This predicate evaluates to true:
  3.3% of the time in normal runs (3830 observations)
  32.5% of the time in the undesired run (2898 observations)
```

Figure 1: The top-ranked configuration option in ConfDiagnoser's error report for the motivating example in Section 2.

In class: randoop.main.GenInputsAbstract

```
     //The maxsize configuration option. Default value: 100.
157. public static int maxsize = readFromCommandLine();
```

In class: randoop.ForwardGenerator

```
99.  public ExecutableSequence step() {
100.   ExecutableSequence eSeq = createNewUniqueSequence();
101.   AbstractGenerator.currSeq = eSeq.sequence;
102.   eSeq.execute(executionVisitor);
103.   processSequence(eSeq);
104.   if (eSeq.sequence.hasActiveFlags()) {
105.     componentManager.addGeneratedSequence(eSeq.sequence);
106.   }
107.   return eSeq;
108. }

310. private ExecutableSequence createNewUniqueSequence() {
311.   Sequence newSequence = ...; //create a sequence
312.   if (newSequence.size() > GenInputsAbstract.maxsize) {
313.     return null;
314.   }
315.   if (this.allSequences.contains(newSequence)) {
316.     return null;
317.   }
318.   return new ExecutableSequence(newSequence);
319. }
```

Figure 2: Simplified code excerpt from Randoop [17] corresponding to the configuration problem reported in Figure 1.

Randoop version, then ConfDiagnoser diagnosed the error's root cause by analyzing the recorded execution profile. ConfDiagnoser produced a report (Figure 1) in the form of an ordered list of suspicious configuration options that should be inspected. The error report in Figure 1 suggests that a configuration option named `maxsize` is the most likely one. The report also provides relevant information to explain why: a program predicate affected by `maxsize` behaves dramatically differently between the recorded undesired execution and the correct executions found in ConfDiagnoser's database.

Figure 2 shows the relevant code snippet in Randoop. When Randoop generates a new test (line 100, in the form of a method-call sequence), Randoop compares its length with `maxsize` (default value: 100). If the generated sequence's length exceeds this pre-defined limit, Randoop discards it to avoid length explosion in further test generation. Although `maxsize`'s default value was carefully chosen by the Randoop developers and works well for many programs (including those used to test Randoop during its development), the generated sequences for NanoXML are much longer than usual and using `maxsize`'s default value results in 32.5% of the generated sequences being discarded (including sequences that the user wishes to retain). ConfDiagnoser captures such abnormal behavior from Randoop's silent failure, pinpoints the `maxsize` option, and suggests the user to change its value. The problem is resolved if the user changes `maxsize` to a larger value, for example 1000.

Configuration Errors in Software Evolution.

We next briefly describe a different scenario where software configuration errors arise. Continual change is a fact of life for software systems. During software evolution, developers may change how the configuration options behave. When upgrading to a new software version, users may need to re-configure the software by changing the values of certain configuration options.

Take the popular JMeter performance testing tool as an example. In version 2.8, the testing report is saved as an XML file after running an example command (`jmeter -n -t ../threadgroup.jmx -l ../output.jtl -j ../test.log`) from the user manual. However, after upgrading to version 2.9, the same command saves the testing report in a CSV file. Further, all JMeter regression tests pass on the updated version. The new JMeter version behaves as *designed* but *differently* than a user was expecting.

ConfDiagnoser can also help users diagnose such configuration errors introduced in software evolution. For the JMeter example, a user first demonstrates the different behaviors on two ConfDiagnoser-instrumented JMeter versions. Then, ConfDiagnoser analyzes the recorded execution traces produced by the two instrumented versions, and outputs a ranked list of suspicious configuration options that may need to be changed. At the top of the list is the `output_format` option with a default value of `CSV` in version 2.9. To resolve this problem, users only need to change its value to `XML`.

3. DIAGNOSIS TECHNIQUE

Diagnosing a configuration error can be divided into two separate tasks: identifying which specific configuration option is responsible for the unexpected behavior, and determining a better value for the configuration option. ConfDiagnoser addresses the former task: finding the root cause of a configuration error.

In ConfDiagnoser, we model a configuration as a set of key-value pairs, where the keys are strings and the values have arbitrary type.

3.1 Overview

Figure 3 sketches the high-level workflow of ConfDiagnoser. ConfDiagnoser takes as input a Java program and its configuration options, and uses three steps to link the undesired behavior to specific root cause configuration options:

1. **Configuration Propagation Analysis**. For each configuration option, ConfDiagnoser uses a lightweight dependence analysis, called thin slicing [23], to statically identify the predicates it affects in the source code.
2. **Instrumentation and User Demonstration**. ConfDiagnoser selectively instruments the program-to-diagnose so that it records the run-time behaviors of affected predicates in an execution profile. When the user encounters a suspected configuration error, the user reproduces the error using the instrumented version of the program.
3. **Root Cause Analysis**. ConfDiagnoser compares the undesired execution profile with a set of correct execution profiles to identify the predicates whose dynamic behaviors deviate the most between correct and undesired executions. The behavioral differences in the recorded predicates provide evidence for what predicates in a program might be behaving abnormally and why. For each deviated predicate, ConfDiagnoser further identifies its affecting configuration options as the likely root causes. Finally, it outputs a ranked list of suspicious configuration options and explanations.

An important component in ConfDiagnoser is the set of correct execution profiles. ConfDiagnoser uses different correct execution profiles when diagnosing different types of configuration errors:

- **Diagnosing configuration errors in a single software version.** We envision that the software developers provide a set of correct execution profiles at release time. The profile set can be further enriched by software users as more correct executions are accumulated. In our experiments (Section 4), we built a database of 6–16 execution profiles by running examples from software user

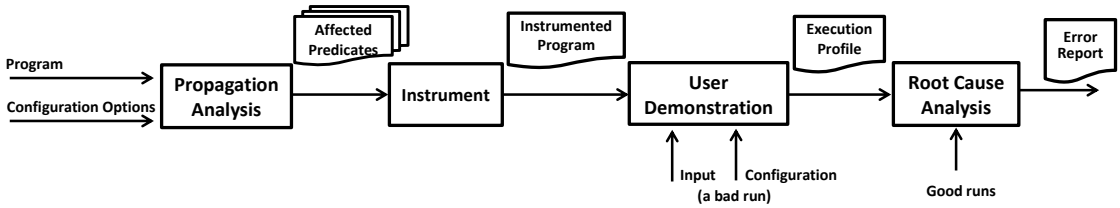

Figure 3: ConfDiagnoser's workflow. The "Propagation Analysis" step is described in Section 3.2. The "Instrument" and "User Demonstration" steps are described in Section 3.3. The "Root Cause Analysis" step is described in Section 3.4. In the "Root Cause Analysis" step, how to obtain the good runs is different in diagnosing configuration errors in a single software version and diagnosing configuration errors in software evolution. The details are described in Section 3.1.

manuals, FAQs, discussion mailing list, forum posts, and published papers. We found that even such a small number of correct execution profiles worked remarkably well for error diagnosis.

- **Diagnosing configuration errors in software evolution.** ConfDiagnoser uses the desired behavior on the instrumented old software version with the same input and configuration as the correct execution profile. It compares the undesired execution profile on the new version with this correct execution profile.

3.2 Configuration Propagation Analysis

ConfDiagnoser takes as input a Java program and its configuration options. For each configuration option, ConfDiagnoser statically computes a forward thin slice [23] from its initialization statement to identify the predicates it may affect in the source code.

In our context, a predicate is a Boolean expression in a conditional or loop statement, whose evaluation result determines whether to execute the following statement or not. A predicate's run-time outcome affects the program control flow. ConfDiagnoser focuses on identifying and monitoring configuration option-affected control flow rather than the values, for two reasons. First, control flow often propagates the majority of configuration-related effects and determines a program's execution path, while the value of a specific expression may be largely input-dependent. Second, it simplifies reporting since a program predicate's outcome can only be either true or false.

To identify the predicates affected by a configuration option, a straightforward way is to use program slicing [6] to compute a forward slice from the initialization statement of a configuration option. Unfortunately, traditional full slicing [6] is impractical because it includes too much of the program. This is due to conservatism (for example, in handling pointers) and to following both data and control dependences. Figure 2 illustrates this problem. Traditional slicing concludes that the predicates in lines 104, 312, and 315 are affected by the configuration option `maxsize`. However, the predicates in lines 104 and 315, though possibly affected by `maxsize`, are actually irrelevant to `maxsize`'s value. That is, the value of `maxsize` controls the length of a generated sequence rather than deciding whether a sequence has an active flag (line 104) or a sequence has been executed before (line 315).

To address this limitation, our technique uses thin slicing [23], which includes only statements that are *directly* affected by a configuration option. Different from traditional slicing [6], thin slicing focuses on data flow from the seed (here, a seed is the initialization statement of a configuration option), ignoring control flow dependencies as well as uses of base pointers. This property separates pointer computations from the flow of configuration option values and naturally connects a configuration option with its directly affected statements. For example, in the code excerpt of Figure 2, a forward thin slice computed for `maxsize` only includes the predicate in line 312. In [33], we have empirically demonstrated that thin slicing is a better choice than traditional full slicing for our purposes.

3.3 Instrumentation and User Demonstration

ConfDiagnoser instruments the tested program offline by inserting code to monitor each affected predicate's outcome at run time. It inserts 2 statements, one before and one after each affected predicate, to count how often the predicate is evaluated and how often the predicate is evaluated to true, respectively.

When the user encounters a suspected configuration error, the user reproduces the error using the instrumented version of the program. Executing the instrumented program produces an execution profile, which consists of a set of predicate profiles. Each predicate profile is a 4-tuple consisting of a configuration option, one of its affected predicates, the predicate's execution count, and its evaluation results as recorded at run time.

3.4 Root Cause Analysis

ConfDiagnoser starts error diagnosis after obtaining the execution profile from an undesired execution.

ConfDiagnoser compares the undesired execution profile with a set of correct execution profiles (or a single correct execution profile from the old software version) to identify the predicates whose dynamic behaviors deviate the most between the selected execution profiles and the undesired execution profile. The behavioral differences in the recorded predicates provide evidence for which parts of a program might be abnormal and why. For each behaviorally-deviated predicate, ConfDiagnoser identifies its affecting configuration options as the likely root causes by using the results of thin slicing (computed by the Configuration Propagation Analysis step). ConfDiagnoser treats the configuration option affecting a higher-ranked deviated predicate as the more likely root cause, and outputs a ranked list of suspicious options.

When diagnosing configuration errors in a single program version, ConfDiagnoser selects from the pre-built profile database, correct execution profiles that are as similar as possible to the undesired one. Selecting similar execution profiles avoids reporting irrelevant differences when determining how and why the observed execution profile behaves differently from the correct ones. When diagnosing configuration errors in software evolution, ConfDiagnoser uses the single execution profile obtained by user demonstration on the instrumented old software version for comparison.

When characterizing the dynamic behavior of a predicate, ConfDiagnoser combines how often it is evaluated (i.e., the number of observed executions) and how often it evaluated to true (i.e., the true ratio) by computing their harmonic mean. For space reasons, we omit the metric for execution trace comparison, the statistical algorithm for identifying behaviorally-deviated predicates, and the error diagnosis algorithm. The interested reader can refer to [33, 34] for details.

4. EXPERIMENTS

Program (version)	LOC	#Configuration Options
Randoop [19] (1.3.2)	18587	57
Weka Decision Trees [28] (3.6.7)	3810	14
JChord [9] (2.1)	23391	79
Synoptic [25] (trunk, 04/17/2012)	19153	37
Soot [22] (2.5.0)	159273	49
JMeter [10] (2.8)	91979	55
Javalanche [8] (0.36)	25144	35

Figure 4: Subject programs used in the evaluation.

Configuration errors in a single software version

Error ID	Program	Description
Non-crashing errors		
1	Randoop	No tests generated
2	Weka	Low accuracy of the decision tree
3	JChord	No datarace reported for a racy program
4	Synoptic	Generate an incorrect model
5	Soot	Source code line number is missing
Crashing errors		
6	JChord	No main class is specified
7	JChord	No main method in the specified class
8	JChord	Running a nonexistent analysis
9	JChord	Invalid context-sensitive analysis name
10	JChord	Printing nonexistent relations
11	JChord	Disassembling nonexistent classes
12	JChord	Invalid scope kind
13	JChord	Invalid reflection kind
14	JChord	Wrong classpath

Configuration errors in software evolution

Error ID	Program	Description
15	Randoop	Poor performance in test generation
16	Weka	A different error message when Weka crashes
17	Synoptic	Initial model not saved
18	Synoptic	Generated model not saved as JPEG file
19	JChord	Bytecode parsed incorrectly
20	JChord	Method names not printed in the console
21	JMeter	Results saved to a file with a different format
22	Javalanche	No mutants generated

Figure 5: The 22 configuration errors used in the evaluation. For configuration errors in software evolution, all errors except for error#16 are non-crashing errors.

We evaluated ConfDiagnoser on 7 configurable Java software (Figure 4). For each software, we searched its forums, FAQ pages, and the literature of configuration error diagnosis research to find actual configuration problems that users have experienced with it. We collected 22 configuration errors (listed in Figure 5), in which the misconfigured values cover various data types, such as enumerated types, numerical ranges, regular expressions, and text entries.

The 12 non-crashing errors are collected from actual bug reports, mailing list posts, and our own experience. The 10 crashing errors, taken from [18], were used to evaluate the ConfAnalyzer tool.

For each program that contains single-version configuration errors, we spent 3 hours, on average, to build a database containing 6–16 correct execution profiles. For the program that contains configuration errors introduced by software evolution, we downloaded the two versions in which the old version functions correctly while the new version reveals unintended behavior. We demonstrated the behavioral difference across the two versions and collected a correct execution profile and an undesired execution profile.

Diagnosing Configuration Errors in a Single Software Version. On average, ConfDiagnoser's 5th report was the root cause; in 10 out of 14 cases, the root cause was ConfDiagnoser's top 3 reports; and in 8 cases, the root cause was ConfDiagnoser's first report.

We compared ConfDiagnoser to an existing tool, called ConfAnalyzer [18]. ConfAnalyzer uses dynamic information flow analysis to reason about the root cause of a configuration error, and can only diagnose crashing configuration errors. As a result, ConfDiagnoser produced better results for 8 errors, equivalent results for 3 errors, and worse results for 3 errors.

We also compared ConfDiagnoser to two general automated debugging techniques and showed that ConfDiagnoser produced significantly better results. All experimental details are described in [33].

Diagnosing Configuration Errors introduced in Software Evolution. ConfDiagnoser is also highly effective in identifying the root-cause configuration options that should be changed in the new program version. The average rank of the root cause in ConfDiagnoser's output is 1.8. For 6 errors, the root-cause configuration option ranks first in ConfDiagnoser's output; for 1 error, the root-cause configuration option ranks third in ConfDiagnoser's output; and the root-cause option ranks sixth for the remaining error. ConfDiagnoser is successful because of its ability to identify the behaviorally-deviated predicates with substantial impacts through execution trace comparison. The top-ranked deviated predicates often provide useful clues about what parts of a program have performed differently.

Performance. For both experiments, ConfDiagnoser spends an average of 4 minutes to recommend configuration options for one error (including the time to compute thin slices and the time to suggest suspicious options). Computing thin slices for all configuration options is non-trivial. However, this step is one-time cost per program and the results can be precomputed. The time used for suggesting configuration options is roughly proportional to the size of the execution trace rather than the size of the subject program.

Experimental conclusions. We have three chief findings. (1) ConfDiagnoser is highly effective in diagnosing software configuration errors; (2) ConfDiagnoser runs in a practical amount of time, making it an attractive approach to manual debugging; and (3) ConfDiagnoser produces more accurate results than existing approaches [11, 14, 18].

5. RELATED WORK

Software configuration errors are time-consuming and frustrating to diagnose. To reduce the time and human effort needed to troubleshoot software misconfigurations, prior research has applied different techniques to the problem of configuration error diagnosis [2, 3, 13, 18, 26, 29, 30]. For example, Chronus [29] relies on a user-provided testing oracle to check the system behavior, and uses virtual machine checkpoint and binary search to find the point in time where the program behavior switched from correct to incorrect. AutoBash [24] fixes a misconfiguration by using OS-level speculative execution to try possible configurations, examine their effects, and roll them back when necessary. PeerPressure [26] statistically compares configuration states in the Windows Registry on different machines. When a registry entry value on a machine exhibiting erroneous behavior differs from the value usually chosen by other machines, PeerPressure flags the value as a potential error. More recently, ConfAid [3] and X-Ray [1] use dynamic taint analysis to diagnose configuration errors by monitoring causality within the program binary as it executes. ConfAnalyzer [18] uses dynamic information flow analysis to precompute possible configuration error diagnoses for every possible crashing point in a program.

Compared to existing approaches [3, 18, 26, 29, 30, 32, 35–37], ConfDiagnoser is different in the following four aspects:

- **It is fully-automated**. ConfDiagnoser does not require a user to specify *when*, *why*, or *how* the program fails. This is different than many well-known automated debugging techniques such as Delta debugging [32], information flow analysis [3], and dynamic slicing [37].

- **It can diagnose both non-crashing and crashing errors**. Most existing techniques [3, 18, 24, 29] focus exclusively on configuration errors that cause a crash point, an error message, or a stack trace. By contrast, ConfDiagnoser diagnoses configuration problems that manifest themselves as either visible or silent failures.
- **It diagnoses configuration errors caused by software evolution**. Most existing configuration error diagnosis techniques identify errors from a single program version [1–3, 18, 24, 26, 29]. By contrast, ConfDiagnoser is cognizant of software evolution and works on two different versions of the same program. It uses the desired behavior of the old software version as a baseline against which to compare new program behavior, and only reasons about the behavioral differences.
- **It requires no OS-level support.** Our technique requires no alterations to the JVM or standard library. This distinguishes our work from competing techniques such as OS-level configuration error troubleshooting [24, 29].

6. CONCLUSION AND FUTURE WORK

This paper presented a general technique (and its tool implementation, called ConfDiagnoser) for diagnosing configuration errors. Our experimental results show that ConfDiagnoser is effective in diagnosing both configuration errors in a single software version and configuration errors introduced in software evolution. The source code of ConfDiagnoser is publicly available at `http://config-errors.googlecode.com`.

Future work should address the following topics:

User study. We plan a user study to evaluate ConfDiagnoser's usefulness to system administrators and end-users.

Fixing configuration errors. After a configuration error is localized, fixing it is often non-trivial. Thus, we plan to apply automated error patching [30] and software self-adaptation techniques [27] to fix configuration errors.

Improving configuration error reporting. A high-quality error report allows software developers to understand and correct the problems they receive. We plan to develop new error reporting mechanisms to make configuration errors more diagnosable.

7. REFERENCES

[1] M. Attariyan, M. Chow, and J. Flinn. X-ray: automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[2] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *USENIX ATC*, 2008.

[3] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

[4] R. Johnson. More details on today's outage. `https://www.facebook.com/notes/facebook-engineering/more-details-on-todaysoutage/431441338919`.

[5] Volatile and Decentralized. `http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html`.

[6] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, 1988.

[7] A. Hubaux, Y. Xiong, and K. Czarnecki. A user survey of configuration challenges in Linux and eCos. In *VaMoS*, 2012.

[8] Javalanche. `https://github.com/david-schuler/javalanche/`.

[9] JChord. `http://pag.gatech.edu/chord/`.

[10] JMeter. `http://jmeter.apache.org/`.

[11] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.

[12] A. Kapoor. Web-to-host: Reducing the total cost of ownership. *Technical Report 200503, The Tolly Group*, May 2000.

[13] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *DSN*, 2008.

[14] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE*, 2003.

[15] Configuration error brings down the Azure cloud platform. `http://www.evolven.com/blog/configuration-error-brings-down-the-azure-cloud\-platform.html`.

[16] NanoXML. `http://nanoxml.sourceforge.net//`.

[17] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, 2007.

[18] A. Rabkin and R. Katz. Precomputing possible configuration error diagnoses. In *ASE*, 2011.

[19] Randoop. `http://code.google.com/p/randoop/`.

[20] Misconfiguration brings down entire .se domain in Sweden. `http://www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden`.

[21] Y. J. Song, F. Junqueira, and B. Reed. Bft for the skeptics. In *Proc. ACM SOSP'09 Work in Progress Session*.

[22] Soot. `http://www.sable.mcgill.ca/soot/`.

[23] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.

[24] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: improving configuration management with operating system causality analysis. In *SOSP*, 2007.

[25] Synoptic. `http://code.google.com/p/synoptic/`.

[26] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.

[27] Y. Wang and J. Mylopoulos. Self-repair through reconfiguration: A requirements engineering approach. In *ASE*, 2009.

[28] Weka. `www.cs.waikato.ac.nz/ml/weka/`.

[29] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI*, 2004.

[30] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *ICSE*, 2012.

[31] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, 2011.

[32] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.

[33] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *ICSE*, 2013.

[34] S. Zhang and M. D. Ernst. Which configuration option should i change? In *ICSE*, 2014.

[35] S. Zhang, Y. Lin, Z. Gu, and J. Zhao. Effective identification of failure-inducing changes: a hybrid approach. In *PASTE*, 2008.

[36] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *ASE*, Nov. 2011.

[37] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE*, 2003.