Automated Diagnosis of Software Configuration Errors

Sai Zhang Michael D. Ernst
Computer Science & Engineering
University of Washington, USA
{szhang, mernst}@cs.washington.edu

***Abstract*—The behavior of a software system often depends on how that system is configured. Small configuration errors can lead to hard-to-diagnose undesired behaviors. We present a technique (and its tool implementation, called ConfDiagnoser) to identify the root cause of a configuration error — a single configuration option that can be changed to produce desired behavior. Our technique uses static analysis, dynamic profiling, and statistical analysis to link the undesired behavior to specific configuration options. It differs from existing approaches in two key aspects: it does not require users to provide a testing oracle (to check whether the software functions correctly) and thus is fully automated; and it can diagnose both crashing and non-crashing errors.**

We evaluated ConfDiagnoser on 5 non-crashing configuration errors and 9 crashing configuration errors from 5 configurable software systems written in Java. On average, the root cause was ConfDiagnoser's fifth-ranked suggestion; in 10 out of 14 errors, the root cause was one of the top 3 suggestions; and more than half of the time, the root cause was the first suggestion.

I. Introduction

Many software applications support configuration options that allow users to customize their behavior. This flexibility has a cost: when something goes wrong, diagnosing a configuration error can be both time-consuming and frustrating. Technical support contributes 17% of the total cost of ownership of today's software, and troubleshooting misconfigurations is a large part of technical support [11].

Software misconfigurations may lead to incorrect output (i.e., non-crashing errors) or to unexpected termination (i.e., crashing errors). Even when an application outputs an error message, it is often cryptic or misleading [2], [8], [30], [31]. Users may not even think of configuration as a cause of their problem.

A. Motivating Example

We next describe a real scenario in which we used ConfDiagnoser to solve a configuration problem. We received a "bug report" against the Randoop automated test generation tool [16], 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 [15] 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 [36], dynamic information flow tracking [2], 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.

```
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)
```

Fig. 1. The top-ranked configuration option in ConfDiagnoser's error report for the motivating example in Section I-A.

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 [1], [2], [18], [25], [28], [33], [35] can be directly applied.

Our technique (and its tool implementation ConfDiagnoser) can diagnose and correct this problem. We first reproduced the error in a ConfDiagnoser-instrumented 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

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

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. }
```

Fig. 2. Simplified code excerpt from Randoop [16] corresponding to the configuration problem reported in Figure 1.

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.

B. Diagnosing Configuration Errors

Correcting 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. This paper addresses the former task: finding the root cause of a configuration error.

Our technique is designed to be used by system administrators and end-users when they encounter an error that they do not know how to fix. It uses three steps to link the undesired behavior to specific root cause configuration options:

- **Configuration Propagation Analysis**. For each configuration option, ConfDiagnoser uses a lightweight dependence analysis, called thin slicing [20], to statically identify the predicates it affects in the source code.
- **Configuration Behavior Profiling**. 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.
- **Configuration Deviation Analysis**. ConfDiagnoser selects, from a pre-built database, correct execution profiles that are as similar as possible to the undesired one. Then, it identifies 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 pre-built database, which contains profiles from known correct executions. We envision that the software developers build this database at release time. The database can be further enriched by software users as more correct executions are accumulated. In our experiments (Section IV), we built a database of 6–16 execution profiles by running examples from software user manuals, FAQs, discussion mailing list, forum posts, and published papers. We found that even such a small database worked remarkably well for error diagnosis.

Compared to previous approaches [2], [18], [25], [28], [32], [36], ConfDiagnoser has several notable features:

- **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 [2], and dynamic slicing [36]. Our technique also provides an *explanation* of why a configuration option is suspicious.
- **It can diagnose both non-crashing and crashing errors**. Most previous techniques [2], [18], [21], [28] focus exclusively on configuration errors that cause a crash, an error message, or a stack trace. By contrast, ConfDiagnoser diagnoses configuration problems that manifest themselves as either visible or silent failures.
- **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 [21], [28].

C. Evaluation

We evaluated ConfDiagnoser on 14 real configuration errors (9 crashing errors and 5 non-crashing errors) from 5 projects. 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 over half of all cases, the root cause was ConfDiagnoser's first report. Assuming the database of correct execution profiles already exists, ConfDiagnoser takes less than 4 minutes on average to diagnose one error. ConfDiagnoser's accuracy and speed make it an attractive alternative to manual debugging.

We compared ConfDiagnoser to a previous technique, called ConfAnalyzer [18], which uses dynamic information flow analysis to reason about the root cause of a configuration error. ConfDiagnoser produced better results for non-crashing errors and similar results for crashing errors.

We also compared ConfDiagnoser to two techniques leveraging existing fault localization techniques [10], [14] to diagnose configuration errors. ConfDiagnoser substantially outperformed both of them.

Finally, we evaluated two internal design choices of ConfDiagnoser. First, we show that using thin slicing [20] to

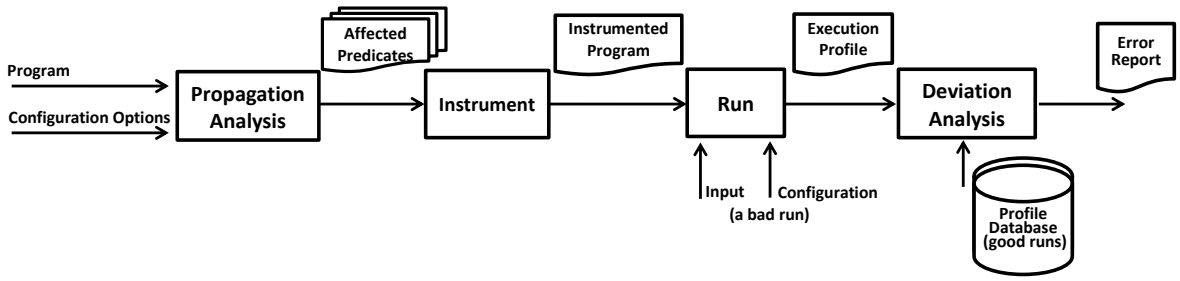

Fig. 3. The workflow of our configuration error diagnosis technique. "Propagation Analysis" is described in Section II-B. The "Instrument" and "Run" components correspond to the Configuration Behavior Profiling step in Section II-C. "Deviation Analysis" is described in Section II-D.

compute the affected predicates yielded more accurate diagnosis than using full slicing [7]. Second, we show that varying the execution profile selection strategy can result in substantially different results. The similarity-based selection strategy used in ConfDiagnoser outperformed the other two strategies.

D. Contributions

This paper makes the following contributions:

- **Technique.** We present a technique to diagnose software configuration errors. Our technique uses static analysis, dynamic profiling, and statistical analysis to link the undesired behavior to specific configuration options (Section II).
- **Implementation.** We implemented our technique in a tool, called ConfDiagnoser, for Java software (Section III). It is available at http://config-errors.googlecode.com.
- **Evaluation.** We applied ConfDiagnoser to diagnose 14 configuration errors in 5 configurable Java software projects. The results show the usefulness of the proposed technique (Section IV).

II. Technique

We model a configuration as a set of key-value pairs, where the keys are strings and the values have arbitrary type. This abstraction is offered by the POSIX system environment, the Java Properties API, and the Windows Registry.

A. Overview

Figure 3 sketches the high-level workflow of our technique. Our technique takes as input a Java program and its configuration options. It first performs a propagation analysis to identify the affected predicates for each configuration option (Section II-B). After that, our technique *selectively* instruments the program at the affected predicates. To diagnose an error, a user runs the instrumented program with the error-revealing input and configuration to obtain an execution profile (Section II-C). Then, our technique analyzes the obtained execution profile to identify the behaviorally-deviated predicates and their root causes, and reports these to the user (Section II-D).

B. Configuration Propagation Analysis

For each configuration option, Configuration Propagation Analysis statically determines its affected *predicates*. 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 because the outcome of a program predicate can only be either true or false. Nevertheless, a program predicate is not the only abstraction our technique can use. Our experiments (Section IV) empirically demonstrate that choosing other abstractions, such as monitoring statement-level coverage or method-level invariants, yields less accurate results.

To identify the predicates affected by a configuration option, a straightforward way is to use program slicing [7] to compute a forward slice from the initialization statement of a configuration option. Unfortunately, traditional full slicing [7] 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 [20], which includes only statements that are *directly* affected by a configuration option. Different from traditional slicing [7], 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. Thin slicing is attractive because it 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. Section IV empirically demonstrates that thin slicing is a better choice than traditional full slicing for our purposes.

C. Configuration Behavior Profiling

This step instruments the tested program offline by inserting code to record how often each predicate evaluates to true at run time.

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 how many times it evaluated to true. For example, suppose the predicate on line 312 has been executed 100 times, of which 30 times it evaluated to true. ConfDiagnoser creates the following predicate profile:
⟨`maxsize`, `newSequence.size() > maxsize`, 100, 30⟩.

Such predicate profiles are by no means complete in recording the whole execution. However, they capture sufficient information to reason about the causal effects of configurations and how a configuration option relates to software's behavior, as shown by our experiments (Section IV). Collecting these profiles imposes only moderate performance impact.

D. Configuration Deviation Analysis

ConfDiagnoser starts error diagnosis after obtaining the execution profile from an undesired execution. It selects similar profiles from known correct executions (Section II-D1), compares each selected profile with the undesired one to identify the most behaviorally-deviated predicates (Section II-D2), and then determines the likely root cause options (Section II-D3).

1) Selecting Similar Execution Profiles for Comparison: ConfDiagnoser's database contains profiles from known correct executions. These execution profiles can be dramatically different from another. To avoid reporting irrelevant differences when determining how and why the observed execution profile behaves differently from the correct ones, ConfDiagnoser first compares the undesired profile with the correct profiles, then selects a set of similar ones as the basis of diagnosis.

ConfDiagnoser first converts each execution profile e into a n-dimensional vector $v_e = \langle r_{e1}, r_{e2}, ..., r_{en} \rangle$, where n is the number of predicates affected by configuration options and each r_{ei} is a ratio representing how often the i-th predicate profile evaluated to true at run time. If a predicate has never been executed in an execution, ConfDiagnoser uses 0 as its ratio.

ConfDiagnoser computes the similarity of two execution profiles e and f by computing the cosine similarity from information retrieval [29] of v_e and v_f.

$$Similarity(e, f) = cos\_sim(v_e, v_f) = \frac{\sum_{i=1}^{n} r_{ei} \times r_{fi}}{\sqrt{\sum_{i=1}^{n} r_{ei}^2} \times \sqrt{\sum_{i=1}^{n} r_{fi}^2}}$$

This similarity metric compares two execution profiles based on control flow taken (approximated by how often each predicate evaluated to true). Its value ranges from 0 meaning completely different predicate behavior, to 1 meaning the same predicate behavior, and in-between values indicating intermediate similarity.

A crashing error sometimes happens soon after the program is launched, so the resulting execution profile is much smaller than most correct execution profiles. To avoid comparing un-executed predicates, when diagnosing a crashing error, ConfDiagnoser reduces each correct execution profile by only retaining the predicates executed by the crashing profile, and then uses the reduced profile for comparison.

Given an undesired execution profile, ConfDiagnoser selects all execution profiles (or the reduced profiles for a crashing error) from the database with a $Similarity$ value above a threshold (default value: 0.9, as used in our experiments).

2) Identifying Deviated Predicates: Our automated error diagnosis approach compares an undesired execution profile with a set of *similar* and *correct* execution profiles. The behavioral differences in the recorded predicates provide evidence for what parts of a program might be incorrect and why.

ConfDiagnoser characterizes the dynamic behavior of a predicate by how often it was evaluated (i.e., the number of observed executions), and how often it evaluated to true (i.e., the true ratio). The true ratio is more important, but it is less dependable the fewer times the predicate has been evaluated.

We define the following ϕ metric, which combines sensitivity (informally, the need for multiple observations) and specificity (informally, the true ratio) in a standard way by computing their harmonic mean.

$$\phi(e, p) = \frac{2}{1/trueRatio(e, p) + 1/totalExecNum(e, p)}$$

In $\phi(e, p)$, $trueRatio(e, p)$ is the ratio of executions of the predicate p that evaluated to true in e, and $totalExecNum(e, p)$ is the the total number of executions of predicate p in e. To smooth corner cases, if a predicate p is not executed in e, i.e., $totalExecNum(e, p) = 0$, then $\phi(e, p)$ returns 0; and if a predicate p's true ratio is 0, i.e., $trueRatio(e, p) = 0$, then $\phi(e, p)$ returns $1/totalExecNum(e, p)$.

The following $Deviation$ metric compares a predicate p across two execution profiles e and f. A larger $Deviation$ value indicates that the behavior is more different.

$$Deviation(p, e, f) = |\phi(e, p) - \phi(f, p)|$$

Often $1/trueRatio(e, p) \gg 1/totalExecNum(e, p)$, and then the value of $Deviation(p, e, f)$ depends primarily on p's true ratio difference between execution profiles e and f.

ConfDiagnoser computes the $Deviation$ value for each predicate p appearing in two execution profiles e and f, and ranks them in decreasing order. The two execution profiles e and f are for the same program, so they have exactly the same set of predicates affected by configuration options.

3) Linking Predicates to Root Causes: ConfDiagnoser links the behaviorally-deviated predicates to their root cause configuration options by using the results of thin slicing (computed by the Configuration Propagation Analysis step in Section II-B). ConfDiagnoser identifies the affecting configuration options for each deviated predicate, and treats the configuration option affecting a higher-ranked deviated predicate as the more likely root cause. If a predicate is affected by multiple configuration options, ConfDiagnoser prefers options whose initialization statements are *closer* to the deviated predicate (in terms of breath-first search distance in the dependence graph of thin slicing). This heuristic is based on the intuition that statements closer to the predicate seem more likely to be relevant to its behavior.

When multiple correct execution profiles are selected for comparison, ConfDiagnoser first produces a ranked list of root

cause configuration options for each comparison pair, and then outputs a final list by using majority voting over all ranking lists. In the final ranking list, one configuration option ranks higher than another if it ranks higher in more than half of the ranking lists. Our implementation breaks possible cycles by arbitrarily ranking the involved options, but this did not occur in our experiments.

In the final output report (e.g., Figure 1), ConfDiagnoser generates a brief explanation for each behaviorally-deviated predicate by showing the difference between the predicate's true ratio during correct executions from the database and during the undesired execution.

E. Discussion

This paper focuses specifically on configuration errors, assuming the application code is correct but the software is inappropriately configured so that it does not behave as desired. We next discuss some design issues in ConfDiagnoser.

Differences between program inputs and configuration options. We took the list of configuration options for each subject program from its manual. Typically, the manual calls an input a configuration option when it controls a program's control flow rather than producing result data. A configuration option is often supplied via a command-line flag or configuration file.

Why not use profiles from unit test executions? ConfDiagnoser's database stores correct profiles from complete executions that start at the main method. ConfDiagnoser does not use profiles from unit test executions, which check the correctness of a single program component and produce an incomplete execution profile that is not representative of the whole program workflow.

Why not store profiles from failing executions in the database? We envision the profile database is built by developers at release time. It is more natural and easier for a developer to provide correct execution profiles, instead of anticipating and enumerating the possible errors a user may encounter.

What if a similar execution profile is not available? ConfDiagnoser's effectiveness largely depends on the availability of similar execution profiles from the database. For a given undesired execution profile, lacking a similar profile in ConfDiagnoser's database may lead ConfDiagnoser to produce less useful results. It also indicates inadequacy of the tests from which the database was constructed. Future work should remedy this problem. One possible approach is to synthesize a new execution, either by generating a new input for the program [34] or by mutating an existing execution [22].

III. Implementation

We implemented a tool, called ConfDiagnoser, on top of the WALA framework [24]. Our tool analyzes Java bytecode. It statically computes the affected predicates for each configuration option, assigns a unique ID for each affected predicate, and then performs offline instrumentation. The runtime behavior of all affected predicates is recorded in a file.

For a Java program, ConfDiagnoser does not analyze the standard JDK library and all the dependent libraries. We believe this approximation is reasonable, since a configuration option set on client software usually does not affect the behaviors of its dependent libraries.

Program (version)	LOC	#Config Options	#Profiles
Randoop (1.3.2)	18587	57	12
Weka Decision Trees (3.6.7)	3810	14	12
JChord (2.1)	23391	79	6
Synoptic (trunk, 04/17/2012)	19153	37	6
Soot (2.5.0)	159273	49	16

Fig. 4. Subject programs. Column "LOC" is the number of lines of code, as counted by CLOC [4]. Column "#Config Options" is the number of configuration options. Column "#Profiles" is the number of execution profiles in the pre-built database.

IV. Evaluation

Our evaluation answers the following research questions:

- How effective is ConfDiagnoser in error diagnosis? ConfDiagnoser's effectiveness can be reflected by:
  - the absolute ranking of the actual root cause in ConfDiagnoser's output (Section IV-C1).
  - the time cost of error diagnosis (Section IV-C2).
  - comparison with a previous configuration error diagnosis technique (Section IV-C3).
  - comparison with two fault localization techniques (Section IV-C4).
- What are the effects of using full slicing [7] rather than thin slicing [20] to identify the affected predicates? What are the effects of varying comparison execution profiles? These are two internal design choices (Section IV-C5).

A. Subject Programs

We evaluated ConfDiagnoser on 5 Java programs shown in Figure 4. Randoop [16] is an automated test generator for Java programs. Weka [27] is a toolkit that implements machine learning algorithms. Our evaluation uses only its decision tree module. JChord [9] is a program analysis platform that enables users to design, implement, and evaluate static and dynamic program analyses for Java. Synoptic [23] mines a finite state machine model representation of a system from logs. Soot [19] is a Java optimization framework for analyzing and transforming Java bytecode.

1) Configuration Errors: We collected 14 configuration errors, listed in Figure 5. This paper evaluates all the configuration errors we found; we did not select only errors on which ConfDiagnoser works well. The misconfigured values include enumerated types, numerical ranges, regular expressions, and strings. The 5 non-crashing errors are collected from actual bug reports, mailing list posts, and our own experience. The 9 crashing errors, taken from [18], were previously used to evaluate the ConfAnalyzer tool. All 14 configuration errors are minimal: if any part of the configuration or input is removed, the software either crashes or no longer exhibits the undesired behavior.

B. Evaluation Procedure

For each subject program, we constructed a profile database by running existing (correct) examples from its user manual, FAQs, discussion mailing list, forum posts, and published

Error ID. Program	Root Cause Configuration Option	#Options	Program Output	ConfDiagnoser		ConfAnalyzer	Coverage Analysis	Invariant Analysis	ConfDiagnoser w/ Full Slicing
			Rank	#Profiles	Rank	Rank	Rank	Rank	Rank
Non-crashing errors									
1. Randoop	`maxsize`	57	N	10 / 12	1	X	13	N	46
2. Weka	`m_numFolds`	14	N	2 / 12	1	X	4	5	9
3. JChord	`chord.datarace.exclude.eqth`	79	N	2 / 6	3	X	38	2	73
4. Synoptic	`partitionRegExp`	37	N	2 / 6	1	X	1	N	6
5. Soot	`keep_line_number`	49	N	6 / 16	2	X	46	N	N
Average		47.2	23.6	3.6 / 10.4	1.6	23.6	20.4	15.7	31.7
Crashing errors									
6. JChord	`chord.main.class`	79	1	4 / 6	1	1	1	4	5
7. JChord	`chord.main.class`	79	1	5 / 6	1	1	1	4	5
8. JChord	`chord.run.analyses`	79	1	5 / 6	17	1	17	22	21
9. JChord	`chord.ctxt.kind`	79	1	3 / 6	1	3	25	30	75
10. JChord	`chord.print.rels`	79	1	2 / 6	15	1	20	25	24
11. JChord	`chord.print.classes`	79	1	4 / 6	16	1	13	17	22
12. JChord	`chord.scope.kind`	79	1	5 / 6	1	1	1	N	10
13. JChord	`chord.reflect.kind`	79	1	6 / 6	1	3	5	9	11
14. JChord	`chord.class.path`	79	1	4 / 6	8	N	21	26	6
Average		79	1	4.2 / 6	6.7	5.7	11.5	19.5	19.8

Fig. 6. Experimental results in diagnosing software configuration errors. Column "Root Cause Configuration Option" shows the actual root cause configuration option. Column "#Options" shows the number of configuration options, taken from Figure 4. Column "Program Output" shows the rank of the root cause as indicated by the program's output, such as an error message. Column "ConfDiagnoser" shows the results of using our technique. Column "#Profiles" shows the number of similar execution profiles selected from the pre-built database for comparison, and the number of executions in the database. For each technique, Column "Rank" shows the absolute rank of the actual root cause in its output (lower is better). "X" means the technique is not applicable (i.e., requiring a crashing point), and "N" means the technique does not identify the actual root cause. When computing the average rank, each "X" or "N" is treated as half of the number of configuration options, because a user would need to examine on average half of the options to find the root cause. Column "ConfAnalyzer" shows the results of using a previous technique [18] (Section IV-C3); the data in this column is taken from [18]. Columns "Coverage Analysis" and "Invariant Analysis" show the results of using two fault localization techniques as described in Section IV-C4. Column "ConfDiagnoser w/ Full Slicing" shows the results of using full slicing [7] to compute the affected predicates (Section IV-C5).

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

Fig. 5. The 14 configuration errors used in the evaluation.

papers [16], [18]. We spent 3 hours per program, on average, and obtained 6–16 execution profiles. The average size of the profile database is 35MB, and the largest one (Randoop's database) is 72MB.

We made a simple syntactic change to JChord, which affected 24 lines of code. This change does not modify JChord's semantics; rather, it just encapsulates scattered configuration option initialization statements as static class fields, which simplifies specifying the seed statement in performing slicing.

When diagnosing a configuration error, we first reproduced the error on a ConfDiagnoser-instrumented program to obtain the execution profile. Then, we ran ConfDiagnoser on the obtained execution profile to identify its root causes.

Our experiments were run on a 2.67GHz Intel Core PC with 4GB physical memory (2GB was allocated for the JVM), running Windows 7.

C. Results

1) Accuracy in Diagnosing Configuration Errors: As shown in Figure 6, ConfDiagnoser is highly effective in pinpointing the root cause of misconfigurations. For all 5 non-crashing errors and 5 of the 9 crashing errors, it lists the actual root cause as one of the top 3 options.

a) Non-crashing configuration errors.: ConfDiagnoser is particularly effective in diagnosing non-crashing configuration errors, which are not supported by most other tools. The average rank of the root cause in ConfDiagnoser's output is 1.6. The primary reason is ConfDiagnoser's ability to identify the behaviorally-deviated predicates through execution profile comparison. The top-ranked deviated predicates often provide useful clues about what parts of a program might be abnormal and why.

We use the non-crashing error in Weka as an example to illustrate this point. Weka's decision tree implementation is highly tuned, achieving 70–90% accuracy on its included examples. However, its accuracy drops to 62% on a different dataset we experimented on. We used ConfDiagnoser to diagnose this problem by first building a database by running Weka on its examples, and then obtaining the undesired execution profile by running it on our dataset. As a result, ConfDiagnoser outputs the following report (only the top option is shown):

```
Suspicious configuration option: m_numFolds

It affects the behavior of predicate:
"numFold < numInstances() % numFolds"
(line 1354, class: weka.core.Instances)

This predicate evaluates to true:
  20% of the time in normal runs (4 observations)
  70% of the time in the undesired run (10 observations)
```

The above report reveals an important fact about the low accuracy. The predicate `numFold < numInstances() % numFolds` controls the depth of a decision tree. Its true ratio is substantially higher in the undesired execution than in normal executions. A higher true ratio leads to a deeper tree that is more likely to overfit the training data and yield low accuracy on the testing data. To resolve this problem, we changed `m_numFolds` value from 2 to 3 to reduce the tree depth, and gained a 5% performance increase.

b) Crashing configuration errors.: Crashing errors are often easy for a user to diagnose. This is because a crashing error often produces a stack trace or error message with valuable diagnosis clues. In fact, for the crashing errors selected by the ConfAnalyzer authors, the user is always led to the root cause by the program output, without the need for further analysis. For error #6, JChord throws a `NoClassDefFoundError` when loading a user-specified class. This error reminds the user that a non-existent class might be provided. For error #7, JChord outputs an error message of "Could not find main class [...] or main method in that class" that explicitly informs the user that the configuration option to specify a main class might be wrong. For errors #8–13, JChord outputs the relevant configuration option in its error message. For error #14, JChord throws a `ClassNotFoundException` (for the main class) that reminds the user to check the classpath setting.

ConfDiagnoser is more effective in diagnosing non-crashing errors (average rank: 1.6) than crashing errors (average rank: 6.7). Most of the crashing errors occur soon after the program is launched, so ConfDiagnoser lacks enough predicate behavior observations. Many predicates are only executed once, so their $Deviation$ scores (Section II-D2) turn out to be the same; and the BFS-distance-based heuristic to resolve ties (Section II-D3) only works half the time.

2) Performance of ConfDiagnoser: We measured ConfDiagnoser's performance in two ways: the time cost in diagnosing an error and the overhead introduced in reproducing an error in a ConfDiagnoser-instrumented program.

As shown in Figure 7, the performance of ConfDiagnoser is reasonable. On average, it uses less than 4 minutes to diagnose each configuration error (including the time to compute thin slices and the time to recommend suspicious configuration options). Computing thin slices for all configuration options is expensive. However, this step is one-time effort per program and the computed slices can be cached to share across diagnoses.

The performance overhead to reproduce the buggy behavior varies among applications. The current tool implementation imposes a substantial slowdown when reproducing errors 3, 9, 10, and 11 in a ConfDiagnoser-instrumented version. This is due to ConfDiagnoser's naive, inefficient instrumentation code, which we have made no effort to optimize. Even so, an error can be reproduced in less than 4 minutes on average, with a worst case of 13 minutes.

Error ID. Program	Run-time Slowdown (×)	ConfDiagnoser time (seconds)	
		Thin Slicing	Error Diagnosis
Non-crashing errors			
1. Randoop	1.1	50	< 1
2. Weka	1.2	43	< 1
3. JChord	13.2	147	82
4. Synoptic	3.6	24	< 1
5. Soot	3.1	95	21
Mean	2.9	72	21
Crashing errors			
6. JChord	2.4	147	79
7. JChord	1.4	147	75
8. JChord	1.5	147	17
9. JChord	28.5	147	30
10. JChord	13.7	147	13
11. JChord	65.1	147	10
12. JChord	1.6	147	83
13. JChord	1.9	147	8
14. JChord	1.4	147	80
Mean	4.3	147	44

Fig. 7. ConfDiagnoser's performance. The run-time slowdown column shows the cost of reproducing the error in an instrumented version of the subject program, and the mean is the geometric mean. The ConfDiagnoser time has been divided into two parts — computing thin slices and diagnosing an error — and the mean is the arithmetic mean.

3) Comparison with a Previous Technique: We compared ConfDiagnoser with ConfAnalyzer, a dynamic information flow-based technique [18]. We chose ConfAnalyzer because it is the most recent technique and also one of the most precise configuration error diagnosis techniques in the literature. ConfAnalyzer tracks the flow of labeled objects through the program dynamically, and treats a configuration option as a root cause if its value may flow to a crashing point. ConfAnalyzer works well for most of the crashing errors (all of which are from the ConfAnalyzer paper [18]), though as described above these are easy to diagnose even without tool support. However, ConfAnalyzer cannot diagnose non-crashing errors.

The experimental results of ConfAnalyzer are shown in Figure 6 (column "ConfAnalyzer"). For the 9 crashing errors, ConfDiagnoser produced better results for 3 of them, the same results for another 3, and worse results for the remaining 3.

ConfAnalyzer performs best on the easiest crashing errors: those having short execution paths, when an error exhibits almost immediately after the software is launched. In such cases, only a small number of configuration options are initialized and few of them can flow to the crashing point. ConfDiagnoser fails to produce a good diagnosis for these errors, because it can not identify the statistically-behaviorally-deviated predicates based on the limited observation of program behaviors.

ConfAnalyzer outputs less accurate or no results for errors where the root cause option value flows into containers or system calls (e.g., error #14 in Figure 6). ConfDiagnoser can reason (to some extent) about the *consequence* of such a misconfiguration based on the observed predicate behaviors.

4) Comparison with Two Fault Localization Techniques: Another possible way to diagnose a configuration error is to leverage existing fault localization techniques, by treating the undesired execution as a failing run and all correct executions (in the database) as passing runs. We next compare ConfDiagnoser with two state-of-the-art techniques:

- **Statement-level Coverage Analysis**. This technique treats statements covered by the undesired execution profile as potentially buggy, and statements covered the correct execution profiles as correct. Then, it leverages a well-known fault localization technique, Tarantula [10], to rank the likelihood of each statement being buggy, and queries the results of thin slicing to identify its affecting configuration options as the root causes. The results are essentially the same for a variant of coverage analysis: using thin slicing to compute all affected statements, and only monitoring the coverage of such affected statements.
- **Method-level Invariant Analysis**. This technique stores invariants detected by Daikon [5] from correct executions in the database. It treats a method as having suspicious behavior if its observed invariants from the undesired execution are different from the invariants stored in the database [14]. This technique ranks a method's suspiciousness by the number of different invariants, and queries the results of thin slicing to identify its affecting configuration options as the root causes.

In Coverage Analysis, the statement-level granularity is *too fine-grained*. Many statements have exactly the same coverage in the failing/passing executions, and thus have the same suspiciousness score as computed by Tarantula [10]. Furthermore, Tarantula only records whether a statement has been executed or not but does not record how a statement is executed (e.g., how often a predicate evaluates to true). The combination of these two factors causes the low accuracy.

In Invariant Analysis, the method-level granularity is *too coarse-grained*. Invariant detection techniques like Daikon [5] only check program states at method entries and exits to infer likely pre- and post-conditions, and thus are less sensitive to control flow details within a method (e.g., a predicate's true ratio). In our study, Invariant Analysis failed to diagnose 3 errors. For Synoptic, it failed to infer invariants. For Soot and Randoop, it reported the same invariants over undesired and correct executions, for the method containing behaviorally-deviated predicates.

This experiment suggests that we cannot treat configuration options as just another regular program input, and then directly apply existing fault localization techniques [10], [14] to find the error causes. The primary reason is that, unlike a program input, a configuration option is often used to control a program's control rather than produce result data. Thus, focusing on the behaviors of relevant predicates as our tool does may be a good choice.

5) Evaluation of Two Design Choices in ConfDiagnoser: We investigate the effects of:

- using traditional full slicing [7] rather than thin slicing [20] in the Configuration Propagation Analysis step (Section II-B) to compute the affected predicates. Figure 6 (Column "Full Slicing") shows the results.
- varying the comparison execution profiles from the pre-built database. In particular, we compare the similarity-based selection strategy used in ConfDiagnoser (Section II-D1) with two alternatives: selecting all available profiles in the database, and randomly selecting the same number of profiles as ConfDiagnoser uses from the database. Figure 8 shows the results. For random selection, we performed the experiment 10 times and report the average.

Error ID. Program	Rank of the Actual Root Cause		
	All Profiles	Random Selection	Similarity-Based
Non-crashing errors			
1. Randoop	1	2	1
2. Weka	7	6	1
3. JChord	16	19	3
4. Synoptic	1	1	1
5. Soot	13	13	2
Average	7.6	8.2	1.6
Crashing errors			
6. JChord	1	1	1
7. JChord	1	1	1
8. JChord	17	17	17
9. JChord	1	1	1
10. JChord	15	15	15
11. JChord	16	16	16
12. JChord	25	25	1
13. JChord	1	1	1
14. JChord	9	9	8
Average	9.4	9.4	6.7

Fig. 8. Comparison with different execution profile selection strategies (Section IV-C5). The last column "Similarity-based" is the selection strategy used in ConfDiagnoser, and the data in that column is taken from Figure 6.

As shown in Figure 6 (Column "Full Slicing"), ConfDiagnoser achieves substantially less accurate results when using full slicing. The primary reason is that full slicing includes too many irrelevant statements that are only *indirectly* affected by a configuration option value but not pertinent to the task of error diagnosis. In many cases, monitoring the control flow of such indirectly-affected predicates and then linking their behaviors to configuration options leads to low accuracy. Furthermore, performing full slicing is much more expensive than thin slicing; in our experiments, the full slicing algorithm ran out of memory on Soot.

As shown in Figure 8, varying the selection strategy for correct traces can affect the results, depending on the application being analyzed. Using all available execution profiles or randomly selecting execution profiles is less effective, because they make ConfDiagnoser report many irrelevant differences between an undesired execution and a dramatically different execution. When diagnosing a crashing error, ConfDiagnoser is less sensitive to the comparison execution profiles. This is because crashing profiles are often much smaller, executing fewer predicates before reaching the crashing points; and ConfDiagnoser reduces each correct execution profile before diagnosis (Section II-D1). Thus, many irrelevant differences have already been removed.

Because the trace selection strategy improved the accuracy of ConfDiagnoser, we tried applying it to Coverage Analysis and Invariant Analysis as well. That is, we supplied those analyses not with the full database of good runs, but with the runs most similar to the bad run. This approach degraded the accuracy of the other tools beyond the results shown in Figure 6, even though it helped ConfDiagnoser. The reason is that the suspiciousness of a statement or method is inversely proportional to the number of correct execution profiles that cover it. When using fewer correct execution profiles, more statements or methods have the same suspiciousness scores.

D. Experimental Discussion

Limitations. The experiments indicate several limitations of our technique. First, we only focus on named configuration options with a common key-value semantic, and our implementation and experiments are restricted to Java. Second, we evaluated ConfDiagnoser on configuration errors involving just one mis-configured option. Third, our implementation currently does not support debugging non-deterministic errors. For non-deterministic errors, ConfDiagnoser could potentially leverage a deterministic replay system that can capture an undesired non-deterministic execution and faithfully reproduce it for later analysis. Fourth, ConfDiagnoser's effectiveness largely depends on the availability of a similar but correct execution profile. Using an arbitrary execution profile (as we demonstrated in Section IV-C5 by random selection) may significantly affect the results.

Threats to Validity. There are three major threats to validity in our evaluation. First, the 5 programs and the configuration errors may not be representative. Thus, we can not claim the results can be generalized to an arbitrary program. For example, we did not evaluate ConfDiagnoser to diagnose misconfigurations that cause poor performance. Second, in this paper, we focus specifically on configuration errors, assuming the application code is correct. Furthermore, in our experiments, all 14 errors have been minimized (as end-users often do when reporting an error). ConfDiagnoser might produce different error diagnosis results on buggy application code with non-minimized inputs. A user who did not know whether a program was misbehaving due to a bug in the code or an incorrect configuration option would need to apply multiple debugging techniques. We have not yet formulated guidance regarding when the user should give up on ConfDiagnoser and assume the error is not related to a configuration option. Third, we only compared two dependence analyses (thin slicing and full slicing), three abstraction granularities (at the predicate level, statement level [10], and method level [5]), and three other tools (ConfAnalyzer, Coverage Analysis, and Invariant Analysis) in our evaluation. Using other dependence analyses, abstraction levels, or tools might achieve different results.

Experimental Conclusions. We have three chief findings: **(1)** ConfDiagnoser is effective in diagnosing both crashing and non-crashing configuration errors with a small profile database. **(2)** ConfDiagnoser produces more accurate diagnosis than approaches leveraging existing fault localization techniques [10], [14]. **(3)** Thin slicing and selection of similar comparison traces permit ConfDiagnoser to produce more accurate diagnosis than other approaches.

V. Related Work

This section discusses closely-related work on software configuration error diagnosis, automated debugging, and configuration-aware software analysis techniques.

Software Configuration Error Diagnosis. Software configuration error diagnosis is recognized as an important research problem [1], [2], [12], [18], [25], [28]. Chronus [28] relies on a user-provided testing oracle to check the behavior of the system, and uses virtual machine checkpoint and binary search to find the point in time where the program behavior switched from correct to incorrect. AutoBash [21] fixes a misconfiguration by using OS-level speculative execution to try possible configurations, examine their effects, and roll them back when necessary. PeerPressure [25] uses statistical methods to compare 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 [2] uses dynamic taint analysis to diagnose configuration problems 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.

Our technique is significantly different from the other approaches. First, most previous approaches focus exclusively on configuration errors that lead to a crash or assertion failure [1], [2], [18], [28]. By contrast, our technique can diagnose both *crashing* and *non-crashing* errors. Second, several approaches [2], [28] assume the existence of a testing oracle that can check whether the software functions correctly. However, such oracles are often absent in practice or may not apply to the specific configuration problem. A typical software user should not be expected, to invest the substantial time and effort to create an oracle. By contrast, our technique eliminates this assumption. Third, approaches like PeerPressure [25] benefit from the known schema of the Windows Registry, but cannot detect configuration errors that lie outside the registry. Our technique of analyzing the affected predicate behavior is more general.

Automated Debugging Techniques. Program slicing [7] and taint analysis [3] are two well-known techniques to determine which statements and inputs could affect a particular variable. Despite their effectiveness in diagnosing a crashing configuration error by performing backward reachability analysis from the crashing point [2], [18], these two techniques cannot be directly applied to diagnose a non-crashing error.

Delta debugging [32] is a general algorithm to isolate software error causes. It reduces differences between a working state and a broken state to isolate a set of failure-inducing changes. When using delta debugging, the user must provide a single nearby working state and a testing oracle to check

whether an intermediate program state behaves correctly — both of which are difficult tasks. Furthermore, as a minimization technique, delta debugging is not applicable to data (i.e., configuration option values) that are missing nor ones that are incorrect. By contrast, ConfDiagnoser eliminates these assumptions by comparing the undesired execution profile with correct execution profiles, and then identifying the root cause configuration options to account for the behavioral difference.

Statistical algorithms have been applied to the automated debugging domain. The CBI project [13] analyzes executions collected from deployed software to isolate software failure causes. Significantly different than ConfDiagnoser, CBI correlates a predicate's evaluation result (either true or false) rather than its true ratio and execution frequency with the observed behaviors. In addition, CBI identifies likely buggy statements as its final output, while ConfDiagnoser identifies the behaviorally-deviated program predicates and links the undesired behavior to specific configuration options.

Configuration-Aware Software Analysis and Testing. Empirical studies show that configuration errors are pervasive, costly, and time-consuming to diagnose [8], [31]. To alleviate this problem, researchers have designed various software analysis techniques to understand and test the behavior of a configurable software system [6], [17]. Compared to ConfDiagnoser, those techniques can be used to find new errors in a configurable software system earlier, but cannot identify the root cause of a revealed configuration error. By contrast, our technique is designed to diagnose an exhibited error.

VI. Conclusion and Future Work

This paper presented a practical technique (and its tool implementation, called ConfDiagnoser) for diagnosing configuration errors. Our experimental results show that ConfDiagnoser is effective in diagnosing both crashing and non-crashing configuration errors, and it does so with a small profile database. 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. A challenge will be finding study participants who are familiar with our subject programs.

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 [26] to fix configuration errors.

Improving configuration error reporting. A high-quality error report allows software developers to understand and correct the problems they receive. Unfortunately, the quality of error reports often decreases as software becomes more complex and configurable. We plan to develop new error reporting mechanisms to make configuration errors more diagnosable.

Acknowledgements

This work was supported in part by ABB Corporation and NSF grant CCF-1016701.

References

[1] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *USENIX ATC*, 2008.
[2] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.
[3] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA*, 2007.
[4] CLOC. http://cloc.sourceforge.net/.
[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, 1999.
[6] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Using feature locality: can we leverage history to avoid failures during reconfiguration? In *ASAS*, 2011.
[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, 1988.
[8] A. Hubaux, Y. Xiong, and K. Czarnecki. A user survey of configuration challenges in Linux and eCos. In *VaMoS*, 2012.
[9] JChord. http://pag.gatech.edu/chord/.
[10] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.
[11] A. Kapoor. Web-to-host: Reducing the total cost of ownership. *Technical Report 200503, The Tolly Group*, May 2000.
[12] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *DSN*, 2008.
[13] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.
[14] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE*, 2003.
[15] NanoXML. http://nanoxml.sourceforge.net//.
[16] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, 2007.
[17] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA*, 2008.
[18] A. Rabkin and R. Katz. Precomputing possible configuration error diagnoses. In *ASE*, 2011.
[19] Soot. http://www.sable.mcgill.ca/soot/.
[20] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.
[21] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: improving configuration management with operating system causality analysis. In *SOSP*, 2007.
[22] W. N. Sumner, T. Bao, and X. Zhang. Selecting peers for execution comparison. In *ISSTA*, 2011.
[23] Synoptic. http://code.google.com/p/synoptic/.
[24] WALA. http://sourceforge.net/projects/wala/.
[25] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.
[26] Y. Wang and J. Mylopoulos. Self-repair through reconfiguration: A requirements engineering approach. In *ASE*, 2009.
[27] Weka. www.cs.waikato.ac.nz/ml/weka/.
[28] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI*, 2004.
[29] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1996.
[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, Y. Lin, Z. Gu, and J. Zhao. Effective identification of failure-inducing changes: a hybrid approach. In *PASTE*, 2008.
[34] S. Zhang, D. Saff, Y. Bu, and M. D.Ernst. Combined static and dynamic automated test generation. In *ISSTA*, 2011.
[35] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *ASE*, 2011.
[36] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE*, 2003.