ConfDiagnoser: An Automated Configuration Error Diagnosis Tool for Java Software

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

***Abstract*—This paper presents ConfDiagnoser, an automated configuration error diagnosis tool for Java software. ConfDiagnoser identifies the root cause of a configuration error — a single configuration option that can be changed to produce desired behavior. It uses static analysis, dynamic profiling, and statistical analysis to link the undesired behavior to specific configuration options. ConfDiagnoser 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 demonstrated ConfDiagnoser's accuracy and speed on 5 non-crashing configuration errors and 9 crashing configuration errors from 5 configurable software systems.

I. Problem

The behavior of a configurable software system often depends on how that system is configured. Small configuration errors can lead to hard-to-diagnose undesired behaviors. Technical support contributes 17% of the total cost of ownership of today's software, and troubleshooting misconfigurations is a large part of technical support [5]. Software misconfigurations may lead to incorrect output (i.e., non-crashing errors) or to unexpected termination (i.e., crashing errors). Even when a software system outputs an error message, it is often cryptic or misleading [1], [3], [15], [16]. Users may not even think of configuration as a cause of their problem.

II. 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. As sketched in Figure 1, ConfDiagnoser uses three steps to link the undesired behavior to specific root cause configuration options:

1. 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 [9] 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.

2. Configuration Behavior Profiling. 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. Configuration Deviation Analysis. ConfDiagnoser starts error diagnosis after obtaining the execution profile from an undesired execution. 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 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.

In this step, selecting similar execution profiles avoids reporting irrelevant differences when determining how and why the observed execution profile behaves differently from the correct ones. When characterizing the dynamic behavior of a predicate, ConfDiagnoser combines how often it is evaluated

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

ICSE 2013, San Francisco, CA, USA
ACM Student Research Competition

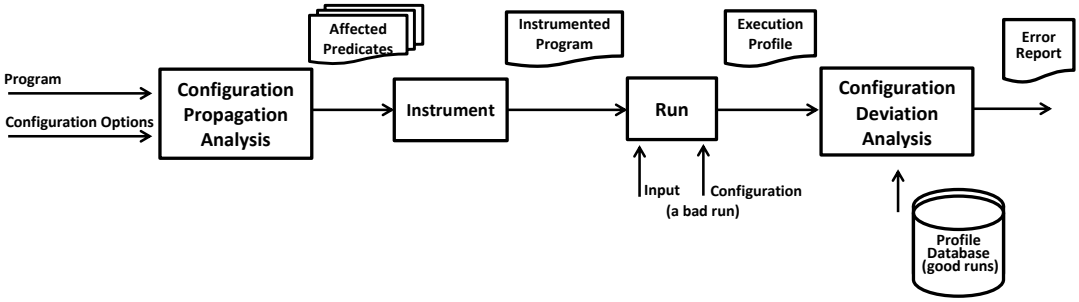

Fig. 1. ConfDiagnoser's workflow. The "Instrument" and "Run" steps correspond to the "Configuration Behavior Profiling" step as described in Section II.

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. 3. The 14 configuration errors used in the evaluation.

Program (version)	LOC	#Options	#Profiles
Randoop [7] (1.3.2)	18587	57	12
Weka Decision Trees [13] (3.6.7)	3810	14	12
JChord [4] (2.1)	23391	79	6
Synoptic [11] (trunk, 04/17/2012)	19153	37	6
Soot [8] (2.5.0)	159273	49	16

Fig. 2. Subject programs. Column "LOC" is the number of lines of code. Column "#Options" is the number of configuration options. Column "#Profiles" is the number of execution profiles in the pre-built database.

(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 [18] for details.

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 III), we built such a database by running examples from software user manuals, FAQs, and forum posts. We found that even a small database of 6–16 execution profiles worked remarkably well for error diagnosis.

Compared to existing approaches [1], [6], [12], [14], [15], [17], [19]–[21], 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 [17], information flow analysis [1], and dynamic slicing [21].
- **It can diagnose both non-crashing and crashing errors**. Most existing techniques [1], [6], [10], [14] 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 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 [10], [14].

III. Experiments

We evaluated ConfDiagnoser on 5 configurable Java software (Figure 2). 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 14 configuration errors, in which the misconfigured values cover various data types, such as enumerated types, numerical ranges, regular expressions, and text entries. They are listed in Figure 3. The 5 non-crashing errors are collected from actual bug reports, mailing list posts, and our own experience. The 9 crashing errors, taken from [6], were used to evaluate the ConfAnalyzer tool. For each program, we spent 3 hours, on average, to build a database containing 6–16 execution profiles.

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. 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 approach.

We compared ConfDiagnoser to an existing tool, called ConfAnalyzer [6]. 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.

Finally, we evaluated an internal design choices of ConfDiagnoser. We show that using thin slicing [9] to compute the affected predicates yielded more accurate diagnosis than using full slicing [2] for 13 out of 14 errors.

REFERENCES

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

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

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

[4] JChord. http://pag.gatech.edu/chord/.

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

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

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

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

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

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

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

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

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

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

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

[16] 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.

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

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

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

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

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