Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors

Sai Zhang
University of Washington, USA
szhang@cs.washington.edu

Michael D. Ernst
University of Washington, USA
mernst@cs.washington.edu

ABSTRACT

This paper presents a technique to detect inadequate (i.e., missing or ambiguous) diagnostic messages for configuration errors issued by a configurable software system.

The technique injects configuration errors into the software under test, monitors the software outcomes under the injected configuration errors, and uses natural language processing to analyze the output diagnostic message caused by each configuration error. The technique reports diagnostic messages that may be unhelpful in diagnosing a configuration error.

We implemented the technique for Java in a tool, ConfDiagDetector. In an evaluation on 4 real-world, mature configurable systems, ConfDiagDetector reported 43 distinct inadequate diagnostic messages (25 missing and 18 ambiguous). 30 of the detected messages have been confirmed by their developers, and 12 more have been identified as inadequate by users in a user study. On average, ConfDiagDetector required 5 minutes of programmer time and 3 minutes of compute time to detect each inadequate diagnostic message.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging.
General Terms: Reliability, Experimentation.
Keywords: Software configuration errors, diagnostic messages, empirical studies.

1. INTRODUCTION

Software configuration errors (also known as misconfigurations) are errors in which the software code and the input are correct, but the software does not behave as desired because an incorrect value is used for a configuration option [55, 60, 62, 67]. Such errors can manifest themselves as crashes, erroneous output, hangs, or silent failures. In practice, software configuration errors have a significant impact on software reliability.

Software configuration errors are pervasive. They accounted for the majority of user-visible failures in Yahoo's mission-critical Zookeeper service [45] and caused about 31% of all failures at a commercial storage company [62]. At Google, a considerable amount of production failures arise not due to bugs in the software, but bugs in the configuration settings (i.e., configuration errors) that control the software [16]. Even worse, software configuration errors can have serious impacts. For example, an incorrect configuration value left Facebook inaccessible for about 2 hours [15]. The entire `.se` domain of Sweden was unavailable for about 1 hour, due to a DNS misconfiguration problem [42]. A misconfiguration made Microsoft's Azure platform unavailable for about 2.5 hours [35]. Each of these incidents affected millions of users.

Unfortunately, software configuration errors are difficult to debug and fix. Fixing a configuration error, or even understanding how a configuration error arises, is often a non-trivial task. For example, troubleshooting a configuration error in the CentOS kernel requires the user to gain deep understanding about the exhibited symptom, and to re-install kernel modules and also modify configuration option values in several places to get it to work [62]. Techniques to help escape from "configuration hell" are in high demand [16].

1.1 Detection of Inadequate Diagnostic Messages

Good diagnostic messages are essential in improving software reliability and diagnosability. In many cases, diagnostic messages are the sole data source available to understand an exhibited error. Developers often attempt to map diagnostic message content to source code statements and work backwards to infer what possible conditions might have led to the error. Diagnostic messages are also important for software users, who depend on the symptom described in a diagnostic message when devising solutions. When software misconfiguration happens, the software system should provide a clear diagnostic message relevant to the root cause — the misconfigured option. A diagnostic message that pinpoints the misconfigured option would be best.

Unfortunately, many configurable software systems lack adequate diagnostic messages. When a software system issues a diagnostic message, it is often cryptic, hard to understand, or even misleading [25, 62]. A field study indicated that up to 25% of a software maintainer's time is spent following blind alleys suggested by poorly constructed and unclear messages [5].

Our technique, called ConfDiagDetector, helps *software developers* improve the diagnosability of a configurable software system. ConfDiagDetector is not a configuration error detection or debugging tool. It takes a radically different approach than prior configuration management and troubleshooting techniques [2–4, 13, 20, 49, 55, 61, 63, 63, 65–67]. Instead of debugging an exhibited configuration error, ConfDiagDetector *proactively* detects inadequate (i.e., missing or ambiguous) diagnostic messages for potential software configuration errors at development time, before they actually occur in the field. Each detected message, if issued by the software, is likely to be unhelpful to diagnose the exhibited configuration error.

ConfDiagDetector embodies two **key ideas**: configuration mutation and NLP text analysis. ConfDiagDetector works by injecting configuration errors into a configurable system, observing the re-

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA
© 2015 ACM. 978-1-4503-3620-8/15/07...$15.00
http://dx.doi.org/10.1145/2771783.2771817

```
  /*
   * Variable howToSave stores the value of the
   * jmeter.save.saveservice.output_format configuration option.
   * XML and CSV are two constants: XML = "XML" and CSV = "CSV"
   * If _xml = true, JMeter later saves its result as an XML file.
   * Otherwise, it saves saves the result as a CSV file.
   */
   if (XML.equals(howToSave)) {
       _xml = true
   } else {
+      if(!CSV.equals(howToSave)) {
+         log.warn(OUTPUT_FORMAT_PROP + " has unexepected value: '"
+            + howToSave + "' - assuming 'csv' format");
+     }
      _xml = false;
   }
```

Figure 1: A missing diagnostic message detected by ConfDiagDetector in JMeter 2.9. Lines starting with "+" are the improved diagnostic message added by the JMeter developers. The report generated by ConfDiagDetector for this missing diagnostic message is shown in Figure 2, and the submitted issue report and fixes are available at https://issues.apache.org/bugzilla/show_bug.cgi?id=55623. Note that ConfDiagDetector does not require or work on source code; the source code above is only for illustrative purposes.

```
Injected configuration error:
   jmeter.save.saveservice.output_format = TXT
Test case:
   jmeter -n -t ../threadgroup.jmx -l  ../output.jtl -j ../test.log
Diagnostic message: N/A
Manual description for the jmeter.save.saveservice.output_format option:
   Help determine how result data will be saved. Legitimate values:
   xml, csv, db. Only xml and csv are currently supported
```

Figure 2: The report generated by ConfDiagDetector for the missing diagnostic message in JMeter 2.9, shown in Figure 1.

sulting failures, and using NLP text analysis to check whether the software issues an informative diagnostic message relevant to the root-cause configuration option. If not, ConfDiagDetector reports the diagnostic message as inadequate.

Figure 1 shows a real-world missing diagnostic message detected by ConfDiagDetector. JMeter provides a configuration option, named `jmeter.save.saveservice.output_format`, for users to specify the output file format. The valid values are "`XML`" and "`CSV`". When a JMeter user provides an unsupported value, such as "`TXT`", for `jmeter.save.saveservice.output_format`, JMeter treats the unsupported value as the default value (i.e., "`CSV`") without notifying the user. This behavior silently violates the user's intention and expectations, and the system does not give any useful feedback. Upon receiving ConfDiagDetector's report (Figure 2), the JMeter developers improved this inadequate diagnostic message: they added an additional check with an informative message that logs the configuration name and its value.

Figure 3 shows a real-world ambiguous diagnostic message detected by ConfDiagDetector in Apache Derby [12]. When a user provides an incorrect value, such as "`hello`", for the `derby.stream.error.method` configuration option, the Derby database system crashes. The stacktrace contains an obscure diagnostic message, "Unable to establish connection", that is not obviously related to the root-cause option. A user may be misled to think about other possible failure causes. Our ConfDiagDetector technique uses natural language processing (Section 2.3) to determine that the diagnostic message is unrelated to the documentation for the configuration option, and it generates the error report in Figure 3.

```
Injected configuration error:
   derby.stream.error.method = hello
Test case:
   java -jar derbyrun.jar ij query_example.sql
Diagnostic message:
   IJ ERROR: Unable to establish connection
Manual description for the derby.stream.error.method option:
   Specifies a static method that returns a stream to which the
   Derby error log is written.
```

Figure 3: A report generated by ConfDiagDetector for an ambiguous diagnostic message detected in Apache Derby [12]. The Derby developers have confirmed this issue.

```
...
jmeter.save.saveservice.output_format=csv
httpclient3.retrycount=1
jmeter.save.saveservice.label=true
...
```

Figure 4: Part of an example configuration for JMeter 2.9. In JMeter, all configuration options are specified as key–value pairs in a configuration file (`jmeter.properties`). ConfDiagDetector parses this configuration file and injects configuration errors into it by repeatedly modifying each configuration option's value.

1.2 ConfDiagDetector's Design

ConfDiagDetector is designed to help *software developers* improve configuration error reporting for configurable software systems. ConfDiagDetector operates in three steps (illustrated in Figure 5) to find missing and ambiguous diagnostic messages:

- **Configuration Mutation.** ConfDiagDetector first infers the likely data type of each configuration option from an existing, well-formed example configuration such as shown in Figure 4. Then, it mutates the existing configuration by repeatedly replacing the value of each option with random values or values from a pre-defined pool, as detailed in Section 2.1.
  For the JMeter example in Figure 1, ConfDiagDetector inferred the data type of configuration option `jmeter.save.saveservice.output_format` as File Type[1], and then replaced its existing value (i.e., "`CSV`") with a different one, "`TXT`", that was randomly selected from a pre-defined value pool.
- **Execution Observation.** ConfDiagDetector uses *system tests* to determine the software's behavior. After applying each mutated configuration (containing exactly one modified configuration option value) to the software, ConfDiagDetector executes system tests one by one and monitors the test result and the system output. For each test failure as determined by its testing oracle, ConfDiagDetector collects the issued diagnostic message from the console or logs. The ConfDiagDetector report (Figures 2 and 3) shows how to run the error-triggering test.
  For our experiments (Section 4.2), we converted each usage example in the software user manual into a runnable system test.
- **Diagnostic Message Analysis.** For each failed test, if a diagnostic message is missing, ConfDiagDetector generates a report for developers, such as the JMeter example shown in Figure 2.
  Otherwise, ConfDiagDetector determines the message's adequacy by using a natural language processing technique [33] to analyze the issued diagnostic message and the user manual. Specifically, ConfDiagDetector checks whether the issued message has a similar semantic meaning to the description of the modified configuration option in the user manual. If not, ConfDiagDetector generates a report, such as the example shown in Figure 3, containing the mutated configuration option value, how

[1]The set of data types ConfDiagDetector infers is shown in Figure 6.

to run the test, the output diagnostic message, and the configuration option description in the user manual.

ConfDiagDetector's report (Figures 2 and 3) is a good starting point for developers to improve a configurable software system's diagnosability. Developers can reproduce the software behavior, understand which configuration errors caused which problems, and choose to improve the diagnostic message or the user manual description.

1.3 Comparison with Existing Techniques

Many techniques have been developed to troubleshoot anomalies caused by configuration errors [4, 61, 65], diagnose certain types of configuration errors [2, 3], automate configuration tasks [30, 66, 67], and suggest fixes for a configuration error [49, 59], but none of them helps developers identify inadequate diagnostic messages for software configuration errors. While previous research has mitigated the impact of configuration errors, the best way to help users troubleshoot a misconfiguration is for the software to issue a helpful error report. Following this principle, ConfDiagDetector improves the error diagnosability of a configurable software system by detecting the potentially inadequate diagnostic messages. It differs from previous techniques in four key aspects:

- **Goal: it detects inadequate diagnostic messages.** Most existing techniques assist developers in diagnosing an *exhibited* configuration error [3, 49, 66, 67]. By contrast, ConfDiagDetector *proactively* detects inadequate diagnostic messages before a configuration error actually arises in the field.
- **Analysis: it analyzes diagnostic messages in natural language.** Existing configuration management and troubleshooting techniques primarily focus on analyzing source code [61] or execution traces [66, 67]. By contrast, ConfDiagDetector analyzes diagnostic messages and evaluates their adequacy using natural language processing techniques.
- **Requirements: it requires no source code or prior information.** Most existing configuration management, error detection, and troubleshooting techniques require source code [61], a comprehensive defect history [51], or a large set of usage data [65]. By contrast, ConfDiagDetector eliminates these requirements and operates on a single binary version, together with a well-formed example configuration and test cases. Test cases can be created from examples in the manual, if they are not already available.
- **Portability: it requires no OS-level support.** ConfDiagDetector requires no modifications to the JVM or OS. This makes ConfDiagDetector more portable and easier to apply than competing techniques such as OS-level configuration management [49, 55].

1.4 Evaluation of ConfDiagDetector

We implemented ConfDiagDetector and evaluated it on 4 real-world, mature configurable software systems written in Java. ConfDiagDetector's injected configuration errors triggered many software failures, which it automatically grouped into 50 equivalence classes: in 25 classes, the failure produced a diagnostic message, and in 25 classes, the failure produced no diagnostic message. ConfDiagDetector classified 7 messages as adequate and the other 43 messages as inadequate (18 ambiguous and 25 missing). We also conducted a user study to determine the adequacy of each message flagged by ConfDiagDetector. The 3 study participants labeled 42 messages as inadequate (all but one message classified by ConfDiagDetector as inadequate) and 8 messages as adequate, showing ConfDiagDetector's high accuracy (2% false positive rate and no false negatives).

We reported all 43 detected inadequate diagnostic messages to the software developers. As of Aug. 2014, the developers had confirmed 30 of them.

We compared ConfDiagDetector with an alternative approach that does not use natural language processing techniques; it had a much higher false positive rate (16%, vs. 2% for ConfDiagDetector). We also compared ConfDiagDetector with an approach that determines the adequacy of a diagnostic message based on a Google search. ConfDiagDetector has higher accuracy (12% vs. 2% false positive rate).

ConfDiagDetector is fast enough for practical use, taking 5 minutes of developer time and 3 minutes of compute time, on average, to detect each inadequate diagnostic message.

1.5 Contributions

The main contributions of this paper are:

- **Technique.** We present a technique to proactively detect inadequate messages for software configuration errors. Our technique uses configuration mutation, dynamic monitoring, and natural language processing techniques to identify such messages (Section 2).
- **Tool.** We implemented our proposed technique in a tool, called ConfDiagDetector, for Java software (Section 3).
- **Evaluation.** We applied ConfDiagDetector to 4 configurable software systems, and compared it with two alternative techniques. The results show the accuracy and efficiency of ConfDiagDetector (Section 4).

2. TECHNIQUE

Figure 5 sketches the high-level workflow of ConfDiagDetector. ConfDiagDetector operates in three steps: Configuration Mutation (Section 2.1), Execution Observation (Section 2.2), and Diagnostic Message Analysis (Section 2.3).

2.1 Configuration Mutation

ConfDiagDetector uses two substeps to mutate a configuration. First, it infers the likely data type of each configuration option (Section 2.1.1). Second, it replaces the value of each configuration option with random values (Section 2.1.2).

2.1.1 Inferring Configuration Option Type

Inferring likely types for each configuration option helps the replacement values generated in the next substep bypass simple checks in a program (e.g., checking that a port number option has an integer type). ConfDiagDetector does not assume the availability of source code nor historical usage data [61, 65].

Figure 6 lists the data types that ConfDiagDetector infers. Each data type has an associated regular expression for syntactic matching. For each configuration option, ConfDiagDetector first matches the value in the example configuration against each regular expression. For each match, ConfDiagDetector validates the concrete option value to verify the guessed type. For example, by regular expression matching, a string starting with "http" or "https" or "www" is a potential `URL` type. ConfDiagDetector further validates the concrete value (such as "www.google.com") by verifying the existence of the corresponding website.

If multiple data types are inferred, ConfDiagDetector uses the more specific property. For example, being a `String` is logically implied by being of `URL` type, so if a configuration option value is of both `URL` and `String` types, ConfDiagDetector uses `URL`. If there are multiple inferred data types but no subsumption exists among them, ConfDiagDetector uses the least upper bound in the lattice as the

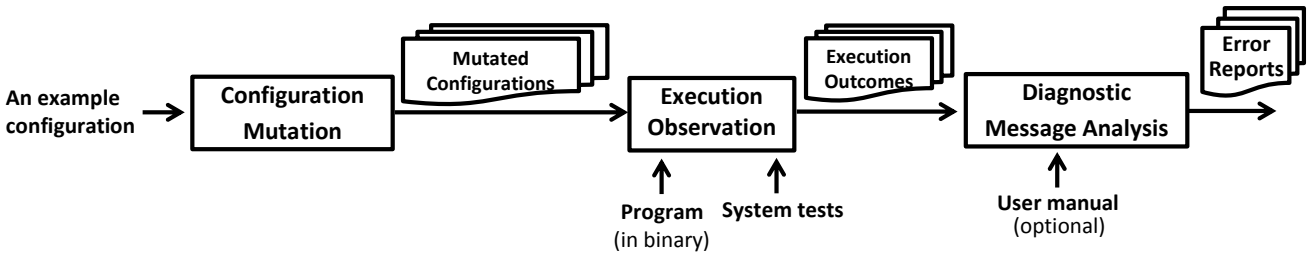

Figure 5: ConfDiagDetector's workflow. The "Configuration Mutation" step is described in Section 2.1, the "Execution Observation" step is described in Section 2.2, and the "Diagnostic Message Analysis" step is described in Section 2.3.

Data Type	Example Configurations (taken from JMeter 2.9)
Integer	summariser.interval = 180
Float	http.version = 2.0
Boolean	sampleresult.timestamp.start = true
File Path	upgrade_properties = /bin/upgrade.properties
Java Class	xml.parser = org.apache.xerces.parsers.SAXParser
File Type	output_format = XML
URL	ns = http://biz.aol.com/schema/2006-12-18
IP Address	remote_hosts = 127.0.0.1
Charset	sampleresult.default.encoding = ISO-8859-1
Language	locales.add = en
String	summariser.name = summary

Figure 6: Data types supported by ConfDiagDetector.

Configuration Mutation Rule	Example	
	Before	After
Delete the existing value	format = XML	format =
Randomly select values of the same data type from a pre-defined pool	format = XML	format = TXT
Randomly select values of a different data type from a pre-defined pool	format = XML	format = 123
Randomly inject spelling mistakes	format = XML	format = XLL
Change the case of text	format = XML	format = xml

Figure 7: Rules ConfDiagDetector uses modify a configuration option.

inferred type. For example, "nic.py" can be either a `URL` or a `File Path`, but neither type implies the other, so ConfDiagDetector uses `String`.

2.1.2 Modifying Configuration Options

ConfDiagDetector modifies an existing, well-formed configuration to create new configurations. Each new configuration includes exactly one added or modified configuration option.

For each option in the well-formed configuration, ConfDiagDetector replaces the value with a different value (see Figure 7). ConfDiagDetector applies the first, fourth, and fifth rules of Figure 7 once and applies the second and third rules k times (k is user-settable, default: 3). ConfDiagDetector also adds a new, unsupported configuration option with a contrived option value, such as `unsupported_option=contrived_value`.

Given an existing, well-formed configuration containing n configuration options, ConfDiagDetector generates $n \times (2k+3)+1$ new configurations, each a potential misconfiguration.

2.2 Execution Observation

ConfDiagDetector checks a software system's behavior using its *system tests*. For each mutated configuration (containing exactly one modified configuration option), ConfDiagDetector executes the system tests. ConfDiagDetector ignores all passing tests. For each failing test, ConfDiagDetector collects the diagnostic message from the console output and/or system logs. Section 3 discusses implementation details.

2.3 Diagnostic Message Analysis

If a test fails without issuing any diagnostic message, ConfDiagDetector immediately generates an error report such as the one shown in Figure 2.

Otherwise, ConfDiagDetector analyzes the diagnostic message using a natural language processing technique. The goal is to determine whether the message is informative enough to guide a user to the root cause (i.e., the modified configuration option).

1. If a diagnostic message contains the modified configuration option name or the value, ConfDiagDetector treats the message as adequate.
2. If a software user manual is available and the diagnostic message has a similar semantic meaning to the manual description of the modified configuration option, ConfDiagDetector treats the message as adequate, since the user can find the root-cause configuration option in the user manual based on the message content.
3. Otherwise, ConfDiagDetector generates an error report such as the one shown in Figure 3.

Checking the first case is straightforward. The rest of this section focuses on the second case, assuming a diagnostic message and a software user manual are available.

2.3.1 Natural Language Processing Preliminaries

It is challenging to convert natural-language text such as diagnostic messages and user manuals into unambiguous specifications that computers can process. Recent research techniques in the area of natural language processing (NLP) show promise in "understanding" the semantic meaning of natural-language text. We next briefly introduce the key NLP techniques used in ConfDiagDetector.

Phrase and Clause Parsing. Also known as chunking, this technique enhances the syntax of a sentence by dividing it into a constituent set of words (or phrases) that logically belong together. For example, the phrase "*goes wrong*" in the sentence "*the program goes wrong*" is an atomic logical unit. Current state-of-the-art approaches can classify phrases and clauses with about 90% accuracy in well-written documents [48].

Bag-of-Words Model. A text such as a sentence or a document is represented as the bag (multiset) of its words/phrases, disregarding grammar and even word order but keeping multiplicity. The bag-of-words model is a common representation in document classification, where the frequency of occurrence of each word is used as a feature to distinguish a document.

Lexical Similarity. A straightforward way to determine the semantic similarity of two sentences is to measure the distance between two vectors converted from the bag-of-words model, where

the vector distance reflects the number of lexical units that occur in both input sentences. However, such a lexical similarity method may under- or over-estimate the semantic similarity of the texts. For instance, there is an obvious similarity between the sentences "*the program goes wrong*" and "*the software fails*", but a lexical similarity method would fail to identify any relationship between them.

2.3.2 Checking Semantic Similarity

ConfDiagDetector employs a method developed in the NLP community [33] for measuring the *semantic* similarity of texts by exploiting the information that can be drawn from the similarity of the component words/phrases. Given two input text sentences, ConfDiagDetector first measures the similarity between each individual word/phrase. We denote the similarity between two words w_1 and w_2 as $word\_sim(w_1, w_2)$.

Then, ConfDiagDetector takes into account the *specificity* of words. It gives a higher weight to a semantic matching identified between two specific words (e.g., *fail* and *go wrong*), than between generic concepts (e.g., *get* and *become*). ConfDiagDetector measures word specificity with a corpus-based measurement, where the corpus is the user manual description for all configuration options. ConfDiagDetector weights words using the well-established inverse document frequency (*idf*), defined as the total number of documents that include that word. We selected the *idf* measure based on previous work that theoretically proved its effectiveness in weighting [46]. In our context, a document is the manual description for one configuration option, and the total number of documents is the number of all configuration options.

Given a metric for word-to-word similarity (i.e., $word\_sim(w_1, w_2)$) and a measure of word specificity (using *idf* [46]), ConfDiagDetector uses the following metric to measure the similarity of two texts T_1 and T_2:

$$sim(T_1, T_2) = \frac{1}{2}(sim'(T_1, T_2) + sim'(T_2, T_1))$$

where

$$sim'(T_1, T_2) = \frac{\sum_{w_1 \in T_1} maxSim(w_1, T_2) \times idf(w_1)}{\sum_{w_1 \in T_1} idf(w_1)}$$

$$maxSim(w_1, T_2) = \max_{w_2 \in T_2} word\_sim(w_1, w_2)$$

maxSim computes the semantic similarity of a word with its best match (highest semantic similarity) in another text. sim' combines these similarities for all the words of a text, using the *idf* weighting and normalizing for the length and words in each sentence. *sim* combines those similarities with a simple average; thus, each word of each text is compared with all words of the other text.

The ConfDiagDetector technique can be instantiated using any word similarity measurement technique [6, 7, 17]. Our current implementation uses WordNet [57], a publicly-available library that encodes relations between words or concepts.

In our implementation, $word\_sim(w_1, w_2) = 1$ if w_1 and w_2 are encoded as synonyms in WordNet; otherwise, $word\_sim(w_1, w_2) = 0$. The *sim* similarity score has a value between 0 and 1. Identical text segments have a score of 1, and a score of 0 indicates no semantic overlap between the two texts. ConfDiagDetector treats two sentences T_1 and T_2 as having similar semantic meaning if $sim(T_1, T_2) \geq \delta$. ($\delta$ is user-settable. Our experiments use 0.4 as the default value.)

2.3.3 Determining Message Adequacy

ConfDiagDetector generates an error report for each configuration error that leads to a test failure where the issued diagnostic message is inadequate.

ConfDiagDetector considers a diagnostic message to be adequate if either of the following is true:

1. The diagnostic message's semantic similarity to the root-cause configuration option description is greater than the pre-defined threshold δ, or
2. The diagnostic message has greater semantic similarity to the root-cause configuration option description than to any other configuration option's description.

2.3.4 Generating an Error Report

Two example reports generated by ConfDiagDetector are shown in Figures 2 and 3. Each generated report contains four parts: (1) the injected configuration error, (2) the failed test case, (3) the issued diagnostic message, and (4) the user manual description of the misconfigured option.

ConfDiagDetector clusters all generated reports based on the misconfigured option and the diagnostic message content, and shows only one error report from each cluster to developers. That is, if multiple misconfigurations from changing the same configuration option cause the same diagnostic message, ConfDiagDetector only shows one misconfiguration and the corresponding diagnostic message.

2.4 Discussion

We next discuss some design choices and limitations.

What if system tests are not available? Our approach requires an example configuration and system tests. Most realistic software contains these and/or a user manual from which they can be generated. For example, the manual usually gives examples of how to use the software. In our experiments (Section 4.2), we converted such usage examples into system tests, demonstrating that even a small number of usage examples worked well for detecting inadequate messages.

Why not use unit tests? ConfDiagDetector uses system tests to observe a software system's behavior from the main method. ConfDiagDetector does not use unit tests, which check the correctness of a single program component, may not be representative of the whole program behavior under a misconfiguration, and may not even read configuration options. However, the main execution path reads most configuration options.

Why not statically extract all diagnostic messages that might be issued at run time? Several factors make it infeasible to perform text analysis statically. Diagnostic messages are often dynamically assembled at run time. Configuration options are often read from files or strings, set via reflection, or use other code patterns that are challenging for a static analysis. Thus, it is difficult to determine exactly which configuration options correspond to which messages and link messages to root-cause configuration errors. Furthermore, a static analysis may overestimate the diagnostic messages that are ever produced at run time. By contrast, ConfDiagDetector uses a dynamic approach to inject real configuration errors, executes the program, and observes its test outcomes to precisely analyze diagnostic messages and detect inadequate ones.

Why not apply fault injection and text analysis to detecting diagnostic messages caused by other types of software errors? It would be interesting for future work to apply our technique to detecting inadequate diagnostic messages caused by other software errors, such as program input errors. We started with the more tractable problem of detecting inadequate diagnostic messages related to software misconfigurations. Software configuration input formats tend

to be simpler (e.g., key–value pairs) and better-documented than other parts of program input, making it easier to generate plausible misconfigurations. For example, to detect inadequate diagnostic messages caused by illegal program input, ConfDiagDetector would need to know a program's input format (e.g., the structure of an image file for an image processing program along with an English description of each part in the user manual) and the proper ways to modify a given input (e.g., which bits of the input can be modified and how). An equally important issue is the need for an accurate behavioral specification. Configuration options usually have small, localized descriptions in a single part of the user manual. By contrast, general program specifications (needed to detect errors or incorrect behavior) are usually absent, and inferring them for realistic software systems is still beyond the state of the art [14, 32, 70]. When such specification inference techniques or tools improve, we could integrate them into ConfDiagDetector, enabling it to detect inadequate diagnostic messages caused by other types of errors.

Limitations. Our technique is neither complete nor sound. It is incomplete (it does not detect every inadequate diagnostic message produced by a software system) because the injected configuration errors do not exhaust all possible errors a user may encounter; furthermore, some misconfigurations it produces may be unlikely in practice. Another source of incompleteness is that ConfDiagDetector ignores non-failing test executions, even though there might have been invisible errors or an incorrect mutated configuration value might have been silently changed to a default value.

It is unsound (not every message flagged by ConfDiagDetector is necessarily inadequate) because the employed natural language processing technique uses heuristics to decide the adequacy of a diagnostic message. However, each misconfiguration detected by our technique is real and fully reproducible. ConfDiagDetector is useful in practice: it detected 43 distinct inadequate diagnostic messages in 4 real-world, mature configurable systems, of which 42 messages have been confirmed as inadequate by their developers or in our user study (Section 4).

Our technique examines a limited number of executions, both because it relies on existing system tests to run the target program and because it does not generate all possible misconfigurations. Thus, ConfDiagDetector does not cover all possible program behaviors nor detect all inadequate diagnostic messages.

It is possible that the unintended software behavior caused by one misconfigured option can be fixed by changing the value of another option, or one configuration option's behavior may depend on another option. (For example, consider Figure 9. The value of `derby.stream.error.rollingFile.count` matters only if `derby.stream.error.style=rollingFile`.) ConfDiagDetector injects one mis-configured option at a time, so it has no special treatment for such cases. A possible way to address this limitation is incorporating a feature model [24] that encodes the relationship between configuration options. Doing so might enable ConfDiagDetector to detect even more inadequate diagnostic messages.

3. IMPLEMENTATION

ConfDiagDetector supports configurations specified as a set of key–value pairs, where the keys are strings and the values have arbitrary type. ConfDiagDetector uses the `java.util.Properties` API to parse and set such configurations.

ConfDiagDetector uses the Stanford Parser [48] to perform phrase and clause parsing and the WordNet library [57] to measure similarity between two words. We enhanced WordNet by adding computer science glossaries [11]. This enhancement is a one-time effort and we did not tune ConfDiagDetector to specific programs. Before checking the semantic similarity between two sentences, ConfDiagDetector discards stop words (such as *a* and *the*).

Given a Java application, ConfDiagDetector provides two modes to launch it: reflectively executing its main method, or executing a provided script. In either case, ConfDiagDetector redirects Java's standard error stream to a file, and then analyzes the produced diagnostic message in the file. If an application writes diagnostic messages to standard output or log files rather than the error stream, ConfDiagDetector requires users to provide a regular expression to extract the diagnostic message. To increase robustness, ConfDiagDetector spawns a thread for each test execution, and monitors its execution. If it hangs (e.g., an infinite loop caused by a configuration error) after the user-specified time (default: 25 seconds), ConfDiagDetector spawns a new thread to execute the next test.

Program	Version	Lines of Code	Options	System Tests
Weka	3.6.11	274,448	125	16
JMeter	2.9	91,979	212	5
Jetty	9.2.1	123,028	23	7
Derby	10.10.1.1	645,017	56	7

Figure 8: Subject programs used in the evaluation. Column "Lines of Code" shows the number of non-blank, non-comment lines of code as counted by CLOC [9]. Column "Options" shows the number of configuration options supported by the evaluated program version. Column "System Tests" shows the number of system tests converted from the usage examples in each program's user manual.

4. EVALUATION

We evaluated the effectiveness of our inadequate diagnositic message detection technique by answering these research questions:

1. How accurate is ConfDiagDetector in detecting inadequate diagnostic messages (Section 4.3.1)?
2. How long does it take for ConfDiagDetector to detect inadequate diagnostic messages (Section 4.3.2)?
3. How does ConfDiagDetector's effectiveness compare to an approach without using natural language processing techniques (Section 4.3.3)?
4. How does ConfDiagDetector's effectiveness compare to an approach based on Google search (Section 4.3.4)?

4.1 Subject Programs

We evaluated ConfDiagDetector on 4 mature Java programs listed in Figure 8. Weka [54] is a toolkit that implements machine learning algorithms. JMeter [28] is a tool for measuring performance. Jetty [27] is an HTTP server and servlet container for serving static and dynamic content. Derby [12] is a relational database system. These programs are configurable, have user manuals, are actively maintained, and have been developed for a long time (5–18 years).

4.2 Evaluation Procedure

If a program's diagnostic messages are mixed with other output, ConfDiagDetector uses a user-specified regular expression to extract issued diagnostic messages from the program's logs. For our evaluation, we spent less than 10 minutes in total to write one regular expression for each subject program. Specifically, a diagnostic message in Weka always starts with "Weka exception"; a diagnostic message in Derby always starts with "IJ ERROR" or "JAVA ERROR"; and a diagnostic message in Jetty or JMeter is always issued along with a stack trace.

ConfDiagDetector takes as input the user manual description for each configuration option. Due to our unfamiliarity with the subject

```
...
derby.stream.error.method
   The derby.stream.error.method property specifies a static method
   that returns a stream to which the Derby error log is written.

derby.stream.error.rollingFile.count
   The derby.stream.error.rollingFile.count property specifies the
   number of rolling log files to permit before deleting the oldest
   file when rolling to the next file, if derby.stream.error.style
   is set to rollingFile.

derby.stream.error.style
   The derby.stream.error.style property specifies that the Derby
   log file should be rolled over when it reaches a certain size.
...
```

Figure 9: A partial list of configuration options and their descriptions extracted from Derby's user manual.

```
1.  public void testOutputFormat() {
2.    String inputFile = "../threadgroup.jmx";
3.    String configFile = ... ;
4.    File f = ReflectionExecutor.execute(
                  org.apache.jmeter.NewDriver.class,
                  inputFile, configFile);
5.    assertTrue(isXMLFile(f));
6.  }
```

Figure 10: An example system test converted from the JMeter usage example shown in Figure 2. The `configFile` variable at line 3 points to the path of JMeter's configuration file part of which is shown in Figure 4, `ReflectionExecutor.execute` is a helper method that reflectively runs JMeter's main method with the given input and configuration, and the `isXMLFile` method at line 5 checks whether the content of a given file is in XML format.

programs and their manuals, we spent 1.5 hours in total to extract the configuration option descriptions from the user manuals. Figure 9 shows an example. Future work could automate parsing the user manual.

A user of ConfDiagDetector would already have a suite of system tests available. However, each of our subject programs only has a unit test suite. Therefore, we generated a system test suite by converting each usage example in the user manuals (35 in all, see Figure 8) into a system test consisting of configuration settings, input data, and expected results. This manual process took about 2 hours in total. Figure 10 shows an example system test converted from the JMeter usage example in Figure 2.

In our experiments, the manual effort was less than 1 hour on average per subject program, or 5 minutes per detected inadequate diagnostic message. We believe this manual cost is reasonable, especially considering our unfamiliarity with these large, complex subject programs.

To evaluate ConfDiagDetector we need a human's judgment about whether each issued diagnostic message is adequate or not. To obtain the ground truth, we conducted a user study in Section 4.3.1, asking three participants to manually label each non-empty diagnostic message as "adequate" or "ambiguous".

4.3 Results

Figures 11, 13, 14, and 15 show the experimental results.

4.3.1 Detected Inadequate Diagnostic Messages

As shown in Figure 11, ConfDiagDetector detected previously unknown inadequate diagnostic messages in every subject program. ConfDiagDetector triggered 25 empty diagnostic messages for different configuration options and 25 distinct non-empty diagnostic messages, and classified 43 messages as inadequate (25 missing and 18 ambiguous) and 7 messages as adequate. ConfDiagDetector is very accurate — our user study confirmed that 42 (out of the 43 messages flagged as inadequate by ConfDiagDetector) messages are actually inadequate, for a false positive rate of 2%. All 7 messages flagged as adequate by ConfDiagDetector are actually adequate, for a false negative rate of 0%.

	Diagnostic Messages					
Program	Total	Missing	Ambiguous	Adequate	FP	FN
Weka	14	0	6	7	1	0
JMeter	12	7	5	0	0	0
Jetty	10	9	1	0	0	0
Derby	14	9	5	0	0	0
Total	50	25	17	7	1	0

Figure 11: ConfDiagDetector caused the subject programs to issue 25 empty diagnostic messages for distinct configuration options (column "Missing") and 25 distinct non-empty diagnostic messages. ConfDiagDetector classified 43 of the messages as inadequate. ConfDiagDetector's classification had only one false positive and no false negatives.

In Figure 11, ConfDiagDetector detected missing diagnostic messages for 25 configuration options. The root cause of these missing messages is that the program silently ignores invalid configuration option values. In three programs, when an invalid value is found, the program silently replaces the invalid value by a default value, as in Figure 1. ConfDiagDetector also detected 17 inadequate messages. These messages arise because of improper handling of invalid configuration option values. All 4 subject programs fail to check the validity of some configuration option values when the program is launched; instead, the invalid value propagates in the program and finally causes a crash or erroneous output. Often, a diagnostic message issued at the crash site only reflects the current program state, rather than the root cause.

The only incorrectly-classified diagnostic message is from Weka. When ConfDiagDetector used an invalid value (such as `67`) for the configuration option `split-percentage`, Weka issued a message "Percentage split cannot be used in conjunction with cross-validation ('-x')." The issued diagnostic message has a quite different semantic meaning from the root-cause option's manual description, but the manual suggests a valid way to correct the exhibited configuration error by changing the value of a *different* option (i.e., the option `x`). ConfDiagDetector does not consider relations between different options and thus incorrectly flagged this message as inadequate. One way to address this limitation is to incorporate a feature model [24] that encodes such a relation across different options.

User Study We conducted a user study to determine the ground truth of whether a diagnostic message is adequate or not. The participants were 3 graduate students majoring in computer science. On average, they had 10 years of programming experience, but none of them was familiar with the subject programs.

We gave each participant the diagnostic messages and the user manuals. For each diagnostic message, we asked each participant to list all configuration options they might change to fix the exhibited error, based on the message content and the user manual.

Each diagnostic message was examined by three participants. We treated a message as adequate if at least two participants correctly identified the root-cause configuration option. In the user study, for a diagnostic message, each user reported 1.5 configuration options on average. As shown in Figure 11, our user study participants concurred with ConfDiagDetector on 49 (out of 50) messages, showing ConfDiagDetector's high accuracy (2% false positive rate and no false negatives).

An **ambiguous** diagnostic message issued by JMeter, caused by `jmeter.save.saveservice.timestamp_format=XYZ`:

```
An error occurred: null
errorlevel=1
Press any key to continue . . .
```

The manual's description:

```
jmeter.save.saveservice.timestamp_format
   Timestamp format - this only affects CSV output files.
```

An **adequate** diagnostic message issued by Weka, caused by `x=-1`:

```
Number of folds must be greater than 1
```

The manual's description:

```
x
   Sets number of folds for cross-validation.
```

Figure 12: Two example diagnostic messages. The top message is an ambiguous one, and the bottom message is an adequate one. ConfDiagDetector correctly classifies both messages. An approach that does not use natural language processing techniques (Section 4.3.3) incorrectly classifies the bottom one as an inadequate message.

	Time Cost (seconds)			
Program	Mutation	Execution	Checking	**Total**
Weka	1	10	3	14
JMeter	1	2803	8	2812
Jetty	1	1945	1	1947
Derby	1	2880	9	2890

Figure 13: ConfDiagDetector's performance. Column "Mutation" shows the time to perform configuration mutation (Section 2.1), column "Execution" shows the time to execute the tests (Section 2.2) using all mutated configurations, and column "Checking" shows the time to check the adequacy of all triggered diagnostic messages (Section 2.3). Our experiments were run on a 2.67GHz Intel Core PC with 4GB physical memory, running Windows 7.

Examples Figure 12 shows two example diagnostic messages triggered by ConfDiagDetector. The first diagnostic message, produced by JMeter, is correctly classified as ambiguous by ConfDiagDetector, since its content is so different from the user manual description and is unlikely to be unhelpful in error diagnosis. The second diagnostic message is correctly classified as adequate by ConfDiagDetector, since the content has a very similar semantic meaning as the user manual description, even though the message itself does not pinpoint the root-cause option. For both messages, our user study confirmed ConfDiagDetector's judgment.

Feedback from Developers We sent the detected inadequate messages to the developers of the subject programs. As of Aug. 2014, they had confirmed 30 of the 43 messages as inadequate.

We also received several pieces of positive feedback from the developers. For example, Derby developer Rick Hillegas noted the pervasiveness of the problems: "Connection attributes have similar problems [to those reported by ConfDiagDetector] involving silent swallowing of validation errors. This can be particularly problematic if the attributes/properties are attempting to configure security mechanisms."

4.3.2 Time Cost

Figure 13 shows the run-time performance of ConfDiagDetector. On average, ConfDiagDetector took 182 seconds to detect each inadequate diagnostic message.

For each subject program, ConfDiagDetector creates 208–1908 mutated configurations based on the number of available configuration options and their types. For the 416 configuration options listed in Figure 8, ConfDiagDetector infers correct data types for 311 (75%) options, incorrect types for 11 (3%) options, and imprecise types for 94 (22%) options. Incorrect types are inferred because these options have enum types, which are not supported in ConfDiagDetector. Imprecise types are inferred primarily because these options have empty values in the example configuration.

The "Mutation" and "Checking" steps are cheap, while the "Execution" step largely depends on the given test. For instance, one test in Derby launches a database server, connects to it, and issues a database query. Running this test took 3 seconds on average, and ConfDiagDetector created over 500 different mutated configurations (after modifying an existing, well-formed configuration) for this test.

	ConfDiagDetector				No Text Analysis			
Program	Ambiguous	Adequate	FP	FN	Ambiguous	Adequate	FP	FN
Weka	6	7	1	0	6	**0**	**8**	0
JMeter	5	0	0	0	5	0	0	0
Jetty	1	0	0	0	1	0	0	0
Derby	5	0	0	0	5	0	0	0
Total	17	7	1	0	17	**0**	**8**	0

Figure 14: Comparison of ConfDiagDetector with a variant that does not use text analysis. Columns are as in Figure 11. Both techniques analyzed the same number of diagnostic messages, and both detected the same 25 missing diagnostic messages as in Figure 11, but the No Text Analysis variant assessed every non-empty diagnostic message as ambiguous, which is 7 additional false positives.

4.3.3 Comparison with Approach that Does No Text Analysis

A straightforward approach to determine the adequacy of a diagnostic message is checking whether the message contains the misconfigured option's name or value. This approach handles the first case of the "Diagnostic Message Analysis" step in ConfDiagDetector (Section 2.3), and has been used in two testing techniques, ConfErr [29] and Spex-INJ [61].

To demonstrate the need for text analysis, we evaluated a variant of ConfDiagDetector by removing the text analysis phase. Like ConfErr and Spex-INJ, this variant simply treats a diagnostic message as inadequate if the message does not contain the misconfigured option's name or value. We used our own implementation of the technique. We were unable to use ConfErr and Spex-INJ because they do not detect inadequate diagnostic messages. Further, ConfErr assumes the availability of the configuration feature model for the tested software, and Spex-INJ assumes the availability of source code and only works for C programs.

The experimental results are shown in the "No Text Analysis" column of Figure 14. The variant without text analysis incorrectly flagged 7 more adequate diagnostic messages in Weka as inadequate than ConfDiagDetector, and increased the overall false positive rate from 2% to 16%. Figure 12 shows an example. These messages do not directly pinpoint the root-cause option, but their content is close enough to the user manual description. Our user study confirmed that a user can identify the root-cause option by reading the message and the user manual. By contrast, ConfDiagDetector uses a natural language processing technique to "understand" such messages, reducing the overall false positive rate to 2%.

ConfDiagDetector's text analysis was extremely useful for the Weka subject program, but it did not affect the results for the other 3 subject programs. Text analysis was not necessary in the 71% of cases when the diagnostic message named the root-cause configuration option. Ideally, each diagnostic message would explicitly list

Program	ConfDiagDetector				Internet Search Technique			
	Ambiguous	Adequate	FP	FN	Ambiguous	Adequate	FP	FN
Weka	6	7	1	0	6	**2**	**6**	0
JMeter	5	0	0	0	5	0	0	0
Jetty	1	0	0	0	1	0	0	0
Derby	5	0	0	0	5	0	0	0
Total	17	7	1	0	17	**2**	**6**	0

Figure 15: Comparison of ConfDiagDetector with an Internet-search-based technique (Section 4.3.4). Columns are as in Figure 11. Both techniques analyzed the same number of diagnostic messages, and both detected the same 25 missing diagnostic messages as in Figure 11, but the search-based technique assessed all but 2 diagnostic messages as inadequate, which is 5 additional false positives.

every possible contributing configuration option, because that makes diagnosis easiest for a user. However, that may not always be possible, and thus it remains valuable to use text analysis to filter out false positive reports that might discourage a user of ConfDiagDetector.

4.3.4 Comparison with Internet Search

A common way for users to diagnose an error is performing a search on the Internet to find the solution, using the program name plus the diagnostic message content as search terms.

We evaluated a search-based approach to determine the adequacy of a diagnostic message. This approach treats a message as adequate if the root-cause configuration option appears in the top-N entries returned by Google. Our evaluation used N=10, which is the default number of entries Google displays on the first page and is also the maximum number of entries a typical user would inspect [56].

Figure 15 shows the experimental results. Compared to ConfDiagDetector, this Internet-search-based technique incorrectly flagged 5 more adequate diagnostic messages in Weka as inadequate, and yielded an overall false positive rate 12% (ConfDiagDetector's false positive rate is 2%). This is primarily because the words used in a diagnostic message are also used elsewhere on the Internet, making a search engine return information that is not related to the root-cause configuration option. This experiment suggests that focusing on the configuration option description in a user manual rather than searching the Internet is a better way to evaluate the adequacy of diagnostic messages.

We also evaluated another variant that employs Google to search the just the user manual, using the message content as a search keyword. This variant achieved the same results as shown in Figure 15.

4.4 Discussion

Implications for Designing Good Configuration Error Handling Mechanisms. Many configuration errors in our evaluation stem from improper configuration error handling. Our findings suggest some good practices in designing good configuration error handling mechanisms. **(1)** Check configuration option values early, ideally immediately after the program is launched. Otherwise, an exhibited configuration error manifests far from the an erroneous program point and becomes difficult to diagnose. **(2)** Never ignore an invalid configuration option value or override it with a default. Rather, the program should notify its users or at least log the invalid value. **(3)** Keep diagnostic messages consistent with the user manual description.

Threats to Validity. There are several major threats to the validity of our evaluation. **(1)** The subject programs, though large and mature, may not be representative. However, these are the first 4 subject programs we tried, and the fact that ConfDiagDetector found inadequate diagnostic messages in all of them is suggestive. **(2)** ConfDiagDetector assumes the user manual correctly describes the usage of each configuration option. It may produce different results if the user manual is out of date or erroneous. **(3)** The generality of our user study is limited: this was a small task, a small sample of people, and unfamiliar code. **(4)** We only considered configuration options used in the example configuration file. There might be other configuration options, even hidden ones not described in the manual. However, such configuration options are often experimental or used only for debugging or in other specific situations.

Experimental Conclusions. We have two chief findings. **(1)** The technique of mutating configurations is an effective and efficient way to proactively detect inadequate diagnostic messages. Usage examples found in practice work well even if no test suite targets configuration options. **(2)** Text analysis based on natural language processing reduces the number of false positives in the detected inadequate diagnostic messages. It is more accurate in determining the adequacy of a diagnostic message than using an Internet search. Text analysis is not necessary if the diagnostic message is empty or if it explicitly names the root-cause configuration option.

5. RELATED WORK

The most closely related work falls into four main categories: (1) configuration error detection and diagnosis techniques; (2) configuration analysis and testing techniques; (3) software error reporting mechanisms; and (4) text analysis for software engineering tasks.

5.1 Configuration Error Detection & Diagnosis

Designing techniques to detect and diagnose software configuration errors has gained increasing attention in both the software engineering and systems communities. Generally speaking, prior techniques falls into two categories: white-box approaches [2–4, 13, 39, 61, 66, 67] and black-box approaches [49, 51, 53, 55, 63, 65].

The white-box approaches use program analyses to reason about a configurable system's behavior and identify potential configuration problems and their root causes. For example, ConfAid [4] and X-Ray [2] use dynamic information flow analysis to detect configuration errors by monitoring causality within the program as it executes. ConfAnalyzer [39] uses static information flow analysis to precompute possible configuration error diagnoses for every possible crashing point in a program. Our previous work, ConfDiagnoser [66] and ConfSuggester [67], analyze the dynamic execution traces in a program to find the root-cause configuration options.

Chronus [55], AutoBash [49], Strider [53], PeerPressure [51], and EnCore [65] are five representative black-box approaches for configuration error detection. Chronus [55] relies on a user-provided testing oracle to check the behavior of the system, and uses binary search to find the point in time where the program started to misbehave. Strider uses manually-labeled working/failing configuration cases as prior knowledge to filter the suspicious entries [53]. AutoBash [49] detects and fixes a misconfiguration by using OS-level speculative execution to try possible configurations, examines their effects, and rolls them back when necessary. PeerPressure [51] 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. EnCore learns configuration rules from a given set of sample configurations to identify a set of configuration anomalies that deviate from the vast majority [65].

Significantly different from existing techniques above, ConfDiagDetector is not a configuration error detection and diagnosis tool. It

does not debug an exhibited configuration error nor help developers find the best location for a fix; rather, it uses configuration mutation to proactively create potential configuration errors before they actually occur in the field. ConfDiagDetector's text analysis to detect inadequate diagnostic messages is also novel. ConfDiagDetector is a black-box approach that requires no source code. Compared to some related techniques, ConfDiagDetector does not require OS-level support as Chronus [55], Strider [53], and AutoBash [49] do. Further, unlike EnCore [65] and PeerPressure [51], ConfDiagDetector requires no prior information (such as training data) nor domain knowledge about how the software behaved in the past.

5.2 Configuration Analysis and Testing

A number of analysis and testing techniques and tools have been developed to improve quality of a configurable software system. For example, Rabkin et al. [40] proposed a method to statically extract and document software system configurations, but their approach requires source code and cannot detect inadequate configuration-error-related messages. The testing community has demonstrated the need for configuration-aware testing techniques [38] and methods for sampling and prioritizing the configuration space [10]. Recent work uses configurability as a way to avoid failures through self-adaption [18] based on a known configuration model. Unlike ConfDiagDetector, these efforts focus on finding configuration errors earlier rather than analyzing the diagnostic messages.

Configuration testing tools, such as SPEX [61] and ConfErr [29], can be used to generate realistic, high-coverage test cases to defend a system against misconfigurations. Those tools have a different focus than ConfDiagDetector: they aim to improve configuration testing coverage rather than evaluating diagnostic messages. Even when a configuration error has been triggered by a generated test input, those tools do not provide support to check its validity and diagnosability. Thus, the error checking phase still remains manual. By contrast, ConfDiagDetector automates this manual phase by using NLP techniques for text analysis and determines whether an output diagnostic message is adequate for error comprehension.

5.3 Software Error Reporting

Software errors degrade software reliability and usability, causing high costs for system administration and maintenance. The cost of maintaining a machine is surpassing the cost of the hardware for modern computing systems [20]. To alleviate the impact of software errors, many advanced error-reporting techniques have been developed to reduce software development cost and improve end-user satisfaction. We next discuss a few representative ones and compare them with ConfDiagDetector.

Statistical debugging [31] correlates low-level application behavior with application behavior and builds a model to predict likely erroneous code fragments. An invariant-based approach [21] relies on dynamic program invariants to detect program behavioral anomalies at run time. Source-code-based techniques, such as LogEnhancer, improve software error reporting by automatically enhancing existing logging code to aid in future post-failure debugging [64]. It analyzes the source code to reason about the information that programmers should have captured when writing log messages. The additional log information provided by LogEnhancer can help narrow down the number of possible code paths and execution states for developers to examine. Stack backtraces are widely used by many remote diagnostic systems, such as Microsoft's online crash analysis [36], the DebugAdvisor system [1], GNOME's bug-buddy [8], and the Clarify diagnosis system [20]. Such systems aim to reduce the human effort needed to identify a reported problem from different program executions.

By contrast, ConfDiagDetector is not a software error reporting or debugging system. Instead, ConfDiagDetector focuses on finding inadequate diagnostic messages in a configurable software system. Error reporting tools are complementary to and can benefit from ConfDiagDetector, since ConfDiagDetector offers a new way to analyze a software system's behavior and find scenarios in which the error reporting mechanism should be improved.

5.4 Text Analysis for Software Engineering

Natural language processing (NLP) techniques are increasingly applied to software engineering tasks. NLP techniques have been shown to be useful in requirements engineering [19, 43, 44], usability of API documents [69, 70], generation of program comments [23,34,47], code completion [22,37,41], and other tasks [50]. For example, Zhong et al. [70] employed NLP and machine learning (ML) techniques to infer resource specifications from API documents. Xiao et al. [58] used shallow parsing techniques to infer Access Control Policy (ACP) rules from natural language text in use cases. Tan et al. [50] applied an NLP- and ML-based approach to test Javadoc comments against implementations. However, to the best of our knowledge, none of the previous work analyzes software diagnostic messages caused by configuration errors nor evaluates their adequacy. Our ConfDiagDetector technique applies NLP techniques to a different problem domain by checking the semantic similarity between two sentences (Section 2.3.1), rather than extracting useful properties from natural language properties [19,43,44,58].

6. CONCLUSION AND FUTURE WORK

This paper presented an approach to detecting inadequate diagnostic messages for software configuration errors. The approach utilizes two key ideas: configuration mutation (creation of configurations that cause a diagnostic message) and NLP text analysis of the diagnostic messages. We demonstrated the accuracy and efficiency of our ConfDiagDetector implementation via an evaluation on 4 real-world, mature configurable software systems.

Our future work will focus on the following two directions:

- **Improving configuration mutation.** ConfDiagDetector employs heuristics to inject configuration errors for inadequate diagnostic message detection and may fail to detect diagnostic messages that need to be triggered by a combination of multiple modified options. We plan to investigate alternative strategies in generating misconfigurations. One possible direction is to employ advanced test generation techniques [26,59] to guide the creation of misconfigurations.
- **Localizing relevant code fragments.** ConfDiagDetector detects inadequate diagnostic messages but does not identify the relevant code fragments. We plan to develop techniques to precisely localize the code fragments that produce the identified inadequate diagnostic messages, so that developers have better guidance when improving them. One possible way is to leverage recent advances in tainting analysis [52] and feature localization techniques [68] to reason about the responsible code snippet.

7. ACKNOWLEDGMENTS

We thank René Just, Xusheng Xiao, and Cheng Zhang for providing insightful comments on a draft. This work was supported in part by NSF grants CCF-1016701 and CCF-0963757.

8. REFERENCES

[1] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala. DebugAdvisor: A recommender system for debugging. In *ESEC/FSE*, 2009.

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

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

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

[5] R. Barrett, E. Kandogan, P. P. Maglio, E. M. Haber, L. A. Takayama, and M. Prabaker. Field studies of computer system administrators: Analysis of system management tools and practices. In *CSCW*, 2004.

[6] D. Bollegala, Y. Matsuo, and M. Ishizuka. Measuring semantic similarity between words using web search engines. In *WWW*, 2007.

[7] S. Bordag. A comparison of co-occurrence and similarity measures as simulations of context. In *CICLing*, 2008.

[8] GNOME Bug Buddy. `https://directory.fsf.org/wiki/Bug-buddy`.

[9] CLOC. `http://cloc.sourceforge.net/`.

[10] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA*, 2007.

[11] Computer Science Glossaries. `http://www.math.utah.edu/~wisnia/glossary.html`.

[12] Derby. `http://db.apache.org/derby/`.

[13] Z. Dong, M. Ghanavati, and A. Andrzejak. Automated diagnosis of software misconfigurations based on static analysis. In *ISSRE*, 2013.

[14] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

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

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

[17] P. Gamallo, C. Gasperin, A. Agustini, and J. G. P. Lopes. Syntactic-based methods for measuring word similarity. In *TSD*, 2001.

[18] 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.

[19] V. Gervasi and D. Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Trans. Softw. Eng. Methodol.*, 14(3):277–330, July 2005.

[20] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *PLDI*, 2007.

[21] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[22] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *ICSE*, 2012.

[23] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *MSR*, 2013.

[24] A. Hubaux, P. Heymans, P. Schobbens, and D. Deridder. Towards multi-view feature-based configuration. In *REFSQ*, 2010.

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

[26] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE*, 2008.

[27] Jetty. `http://www.eclipse.org/jetty/`.

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

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

[30] N. Kushman and D. Katabi. Enabling configuration-independent automation by non-expert users. In *OSDI*, 2010.

[31] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[32] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *FSE*, 2008.

[33] R. Mihalcea, C. Corley, and C. Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI*, 2006.

[34] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for Java classes. In *ICPC*, 2013.

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

[36] Microsofot's Online Crash Analysis. `http://support.microsoft.com/kb/923800`.

[37] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *ESEC/FSE*, 2013.

[38] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA*, 2008.

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

[40] A. Rabkin and R. Katz. Static extraction of program configuration options. In *ICSE*, 2011.

[41] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI*, 2013.

[42] Circleid, misconfiguration brings down entire .se domain in Sweden. `http://www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden`.

[43] A. Sinha, S. M. S. Jr., and A. M. Paradkar. Text2Test: Automated inspection of natural language use cases. In *ICST*, 2010.

[44] A. Sinha, A. M. Paradkar, P. Kumanan, and B. Boguraev. A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In *DSN*, 2009.

[45] Y. J. Song, F. Junqueira, and B. Reed. Bft for the skeptics. In *Proc. ACM SOSP Work in Progress Session*, 2009.

[46] K. Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. In P. Willett, editor, *Document Retrieval Systems*, pages 132–142. Taylor Graham Publishing, London, UK, UK, 1988.

[47] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *ASE*, 2010.

[48] Stanford NLP Parser. `http://nlp.stanford.edu/software/lex-parser.shtml`.

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

[50] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tcomment: Testing Javadoc comments to detect comment-code inconsistencies. In *ICST*, 2012.

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

[52] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *FSE*, 2012.

[53] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA*, 2003.

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

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

[56] R. W. White and D. Morris. Investigating the querying and browsing behavior of advanced search engine users. In *SIGIR*, 2007.

[57] WordNet. `http://wordnet.princeton.edu/`.

[58] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *FSE*, 2012.

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

[60] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki. Range fixes: Interactive error resolution for software configuration. *Software Engineering, IEEE Transactions on*, PP(99):1–1, 2014.

[61] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *SOSP*, 2013.

[62] 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.

[63] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar. Context-based online configuration-error detection. In *USENIX ATC*, 2011.

[64] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. *ACM Trans. Comput. Syst.*, 30(1):4:1–4:28, Feb. 2012.

[65] J. Zhang, L. Renganarayana, X. Z. N. Ge, V. Bala, T. Xu, and Y. Zhou. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *ASPLOS*, 2014.

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

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

[68] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. In *ICSE*, 2004.

[69] H. Zhong and Z. Su. Detecting API documentation errors. In *OOPSLA*, 2013.

[70] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language api documentation. In *ASE*, 2009.