2016 IEEE 27th International Symposium on Software Reliability Engineering

# ORPLocator: Identifying Read Points of Configuration Options via Static Analysis

Zhen Dong*, Artur Andrzejak*, David Lo†, Diego Costa*
*Institute of Computer Science, Heidelberg University
†School of Information Systems, Singapore Management University
{zhen.dong, artur.andrzejak, diego.costa}@informatik.uni-heidelberg.de, davidlo@smu.edu.sg

***Abstract*—Configuration options are widely used for customizing the behavior and initial settings of software applications, server processes, and operating systems. Their distinctive property is that each option is processed, defined, and described in different parts of a software project - namely in code, in configuration file, and in documentation. This creates a challenge for maintaining project consistency as it evolves. It also promotes inconsistencies leading to misconfiguration issues in production scenarios.**

**We propose an approach for detection of inconsistencies between source code and documentation based on static analysis. Our approach automatically identifies source code locations where options are read, and for each such location retrieves the name of the option. Inconsistencies are then detected by comparing the results against the option names listed in documentation.**

**We evaluated our approach on multiple components of Apache Hadoop, a complex framework with more than 800 options. Our tool ORPLocator was able to successfully locate at least one read point for 93% to 96% of documented options within four Hadoop components. A comparison with a previous state-of-the-art technique shows that our tool produces more accurate results. Moreover, our evaluation has uncovered 4 previously unknown, real-world inconsistencies between documented options and source code.**

***Index Terms*—Configuration options, Static analysis, Inconsistency detection, Empirical study**

## I. Introduction

### A. Motivation

Almost all modern non-trivial software systems provide users with configuration options - essentially variables with constant values, specified in user-editable text files. Such options allow customization of behavior of individual program instance and adaptation to the operating environment without need to recompile the source code. The number of configuration options in highly-configurable systems can reach thousands. For instances, the current version of Mozilla Firefox 43.0 has more than 2000 configuration options. Apache Hadoop 2.7.1, a software framework studied in this work, has more than 800 options available to users.

Maintaining large quantities of options is difficult, primarily since each option must be correctly named and used in three independent entities: the source code, the configuration file(s), and the documentation. Especially in large and fast-evolving projects this can lead to many inconsistencies. First, documentation is updated separately from source code, and often by other project participants. Second, developers can fail to update the option names, values or options present in the configuration files during program evolution. The consequences can be erroneous names or descriptions of configuration options in the documentation, incorrect usage of options spanning multiple modules, or mistyped, missing or obsolete options in configuration files.

Debugging associated failures can be tedious and mostly based on trial-and-error. This can cause frustrating experiences for users, e.g., if they cannot achieve a desired effect due to an outdated documentation. Issue HDFS-8274 (see bug repository of Hadoop-HDFS[1]) has been detected with our technique and is a real case of this type. According to documentation, users can specify the dump directory of NFS (*Network File System*) files by setting the value of the option "nfs.dump.dir". However, the value of this option has no effect on the system behavior since the option name used in the source code is "nfs.**file**.dump.dir". This inconsistency can cost hours of debugging until the source code is inspected.

Moreover, issues of this type can be very costly as they are likely to manifest in a production setting, e.g., after software updates. This can have significant economic consequences: Yin *et al*. states that 31% of root causes of high-severity issues in a commercial storage company are caused by configuration errors [24]. Another work by Rabkin and Katz confirm these findings [18].

A straightforward approach for detecting inconsistencies between documented options and the corresponding version of source code is to check whether an option is used in the source code. An indication of this fact is that at least one code statement accesses the option value in the source code. We call such a code statement an *option read point* (*ORP*). To simplify, we do not consider whether the statement is dead code. Given a list of ORPs together with corresponding option names, we can compare it against the documentation in order to detect the above-mentioned inconsistencies. We also can check configuration files and detect incorrect option names there in the same way.

In addition to inconsistency detection, locating option read points is also of value for another two application cases: automated misconfiguration diagnosis, and the extraction of configuration option constraints.

[1]https://issues.apache.org/jira/browse/HDFS-8274

2332-6549/16 $31.00 © 2016 European Union
DOI 10.1109/ISSRE.2016.37

A large body of research [16], [23], [26], [3], [25], [8], [7] has attempted to automatically diagnose software configuration errors. The approaches here include tracing the data flow of option values using program analysis techniques. A prerequisite for almost all of these works is a list of (typically manually specified) option read points. Thus, our approach can further automate these approaches and save considerable manual efforts.

Another branch of work [23], [25], [14], [15] attempts to prevent configuration errors by telling users whether given option values violate pre-defined rules or a set of constraints. Also in this case the identification of option read points is a requirement for using such techniques.

### B. Locating ORPs and Inferring Option Names

A common way for developers to locate option read points is to search the calls of the methods for reading options and infer option names directly from the string parameters used in a method call. This approach is not sufficient for complex applications. In the application we studied, the calls of option-reading methods usually take a variable as a parameter. Detecting the name of an option requires to investigate many methods or classes.

A real example is shown in Figure 1. Line 116 in the class file *CompositeGroupsMapping.java* is an option read point in Apache Hadoop, which takes the class variable *D* as a parameter. The variable *D* is initialized by an expression of combining a string constant and another class variable *C*. Similarly, variable *C* is initialized by an expression of combining variable *B* and a string constant. However, variable *B* is declared in the super class *GroupMappingServiceProvider* of class *CompositeGroupsMapping*. Again, variable *B* is initialized by the variable *A* in a dedicated class for storing option names or their prefixes. In this dedicated class, variable *A* is initialized via a string. Finally, the value of variable D, the option name, is obtained as "hadoop.security.service.user.name.key.providers.combined". This analysis spans 3 classes and 2 packages. Such coding patterns are quite common in the applications which we studied.

The most recent existing work, Confalyzer [17], extracts configuration options by directly reading parameter values of call sites of configuration APIs in the call graph. However, Confalyzer is not sufficient to solve this problem because the dynamic construction of option names like the example above would lead to a low precision of the analysis.

### C. Our Technique and ORPLocator

Addressing this issue, this work proposes a technique capable of automatically locating the option read points from source code, and a prototype implementation called Option Read Point Locator (ORPLocator).

ORPLocator is an automated static analysis technique for locating option read points from source code. Given the name of a class $C$ for handling configuration options and the source code of a program, ORPLocator generates a map between options and their read points.

CommonConfigurationKeysPublic.java

```
    ...
249: public static final String A =
250: "hadoop.security.service.user.name.key";
    ...
```

GroupMappingServiceProvider.java

```
    ...
34: public static final String B =
          CommonConfigurationKeysPublic.A;
    ...
```

CompositeGroupMapping.java

```
    ...
47: public static final String C =
    B + ".providers";
48: public static final String D =
    C + ".combined";
    ...
    public synchronized void
          setConf(Configuration conf) {
    ...
116: this.combined = conf.getBoolean(D,true);
    ...
    }
    ...
```

Fig. 1. A real case of how an option is used in Hadoop 2.7.1. Variable names are replaced by capitalized letters to improve readability.

The first step identifies all classes which extend the specified class $C$. The second step selects methods reading option values in such classes. Then all call sites of these methods are identified. Finally, ORPLocator infers names of the options read by the program at each call site and builds a map between option names and the corresponding read points.

### D. Contributions

Our work makes the following contributions:

- **Technique**. We present an accurate and automated static analysis technique for locating option read points in evolving, highly-configurable, modern software systems. The analysis computes dependence information as needed.
- **Implementation**. We implement our technique in a prototype, called ORPLocator, for Java software programs.
- **Empirical Study**. We conduct an empirical study on the latest version (2.7.1) of Apache Hadoop, a widely popular framework for distributed data processing with more than 1.3 million lines of source code and 800+ configuration options. The result shows that our technique is effective in identifying option read points in source code. Besides, our experimental study discovers 4 previously unknown inconsistencies between documented options and source code.

The rest of this paper is organized as follows. In Section II we describe the assumptions and definitions and present our technique. Section III discusses the implementation of OR-PLocator. We present our empirical evaluation in Section IV. Section V discusses related work. Finally, we conclude and discuss future work in Section VI.

## II. Technique

In this section we first introduce the key-value configuration model targeted by our technique. Then we present the overview and the details of our technique.

### A. Key-value Configuration Model

Most modern applications provide a mechanism allowing users to change the behavior or features. The key-value configuration is a common and widespread approach for users to configure applications [17]. It is supported by the POSIX system environment, the Java Properties API, and the Window Registry.

The key-value configuration model can be illustrated by the example shown in Figure 2. Configuration options are designed as a set of key-value pairs and stored in a configuration file. The keys are strings and the values have arbitrary type. Each pair corresponds to an application attribute. Users are able to control features of applications by setting attribute values in the configuration file.

Meanwhile, application programs have a dedicated class for managing these configuration options, called a *configuration class*. The class takes responsibility of loading key-value pairs in the configuration file to a map, and offers a set of methods like *getInt* and *getString*, each of which takes one option name as a parameter and returns the value of the option. We call methods of reading option values in a configuration class as *get-methods*.

Programs read option values by calling these get-methods. In the example, *conf.getString(keyName)* returns the value of the option with name *key_1*. This statement is called as an option read point of the option named *key_1*.

### B. Overview of the Technique

Our technique requires the source code of a program and specifying its configuration class name. Its workflow is illustrated in Figure 3. The first step identifies subclasses of the given configuration class. The second step selects get-methods in the configuration classes. Then, all call sites of these get-methods, i.e. option read points, are located in the source code. Finally, names of the options read by each call site are inferred and a map between these option names and their read points is reported to users.

Our technique targets applications written in object oriented languages such as Java, C++, and C#. We abstract the source code as a set $\psi$ of entities of classes, interfaces, and enums, which are distributed in different class files. Each entity $e$ has a simple name and a fully-qualified name. The fully-qualified name consists of the package name and the simple name. An entity is retrieved by its fully-qualified name from $\psi$. Statements in classes are denoted by a tuple <$f$, $l$> where $f$ represents the name of its class file; $l$ represents the line number of the statement in the class file.

### C. Identifying Subclasses of the Configuration Class

Modern, non-trivial applications typically have a base configuration class $C$ dedicated to deal with configuration options. Furthermore, different components or subprograms of the application have its own configuration classes obtained by extending or inheriting from $C$. In order to obtain all call sites of get-methods in the program, we need to find all such subclasses and store them in a set $S$.

### D. Identifying the Get-Methods

Obviously, not all methods in a configuration class are get-methods, and distinguishing get-methods from other methods is necessary. Rabkin and Katz observe that methods for accessing option values usually have a common characteristic: their names obey a naming convention, starting with the same prefix like *get* [17]. For particular types of option values, method names which reveal the returning types are given such as *getBoolean*, *getInt*, and so forth. This naming convention for configuration APIs holds in many programs. In our prototype, we adopt this convention and consider the methods in the configuration class whose names start with prefix *get* as get-methods.

The naming convention of methods accessing option values may not hold in some programs. In these cases, we need to manually check each method in the configuration class and select get-methods.

### E. Locating Call Sites of Get-Methods

Intuitively, one can obtain call sites of a method from a specified class by directly searching the method name in the source code. These search results are inaccurate and would contain call sites of methods which have the same name from different classes. In order to accurately locate call sites of get-methods, we first identify instances of configuration classes (Section II-E1) and then locate call sites of get-methods of these instances (Section II-E2).

*1) Identifying Instances of a Configuration Class:* We identify instances of a configuration class by variables declared with the type of the configuration class as well as scopes of these variables. An instance is represented by a tuple <$v$, $s$>, where $v$ is the name of a variable and $s$ is the scope of the variable. All instance variables of configuration classes are stored into a set $V$.

For any entity $e \in \psi$, we check each statement in $e$. If a statement is a declaration statement and the declared type is one of configuration classes in the set $S$, the declared variable $v$ is considered a variable of configuration classes.

The scope of a variable is determined based on three cases. First, the scope of an instance variable or class variable is identified as the largest block of the class. Second, if the variable is a formal parameter of a method, its scope is the corresponding method. Last, the variable is declared locally.

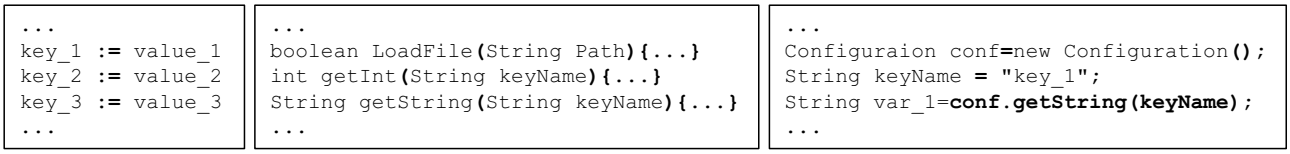


Fig. 2. An example scenario of the key-value configuration schema

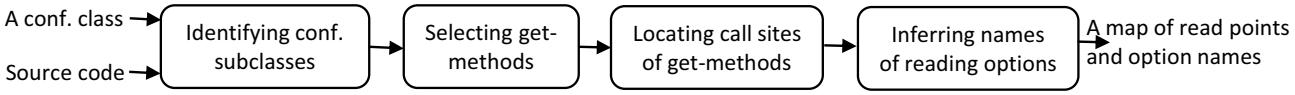


Fig. 3. The workflow of our technique

We consider the smallest block which contains the declaration statement of the variable as its scope.

*2) Searching Call Sites of Get-Methods:* Once the configuration class instances and the corresponding scopes are identified (Section II-E1), we locate call sites of get-methods referenced by these variables in their scopes.

<*methodCall*> → <*methodName*>(<*argumentList*>)|
<*reference*><*selectionOperator*>
<*methodName*>(<*argumentList*>)
<*selectionOperator*> → <*Operator*>
<*reference*> → <*expression*>
<*methodName*> → <*identifier*>
...

Fig. 4. A segment of Backus-Naur Form (BNF) grammar specification for a method call

The grammar of a method call is shown in Figure 4. Based on this grammar, we classify method calls of get-methods into three categories.

First, the <*reference*> in the grammar refers to an instance variable of a configuration class. For any variable <$v$, s> $\epsilon$ $V$, we search all statements in the scope $s$ of the variable and identify method calls which have the pattern <*v*><*selectionOperator*><*methodName*>, where the method name can be any one of the get-method names. These call sites are stored in a set $\Omega$.

Second, the <*reference*> refers to an instance returned by another method call. We adopt a conservative solution to deal with this case. All methods which return configuration class types are identified. By these names, we search the whole program and obtain call sites of these methods. If these call sites are followed by <*selectionOperator*><*methodName*>, where the method name can be any one of the get-method names, we consider them as call sites of the get-methods and add them to $\Omega$.

Last, inside configuration class, the <*reference*> can be implicit. For instance, the keyword *this* is used to reference to the object typed as the current class in Java. Even some get-methods are called without the reference. In order not to miss such call sites, we identify all call sites of get-methods in a configuration class and append them to the set $\Omega$.

### F. Inferring Option Names

This section describes how we infer the name of an option read at a specific option read point. Based on this knowledge, a map between option names and their read points can be created.

In key-value configuration model, a specific option name is passed to a get-method through its call site and this site returns the value of this option. As stated in Section I-B, call sites of get-methods usually take a variable storing an option name as a parameter instead of a string constant. We have to track down the value of the actual variable in a call site and obtain the option name.

Our investigation shows that variables storing the option names have characteristics which distinguish them from variables carrying other values. Variables with option names are typically initialized when they are declared and not reassigned by new values before being read. Values of such variables can be obtained by searching their declaration statements and scanning their initial values. This heuristic was used in several past papers [17], [4]. Besides, their initial values are often not string constants yet expressions combining a variable and a string constant or other variables. In the example in Figure 1, variable $D$ is initialized by variable $C$ and a string constant ".combined" and variable $C$ is initialized by variable $B$ and a string constant ".providers". This usage creates convenience for managing options for different components of a program.

For this complex usage of configuration options, we implement two distinct approaches to track down which option (identified by name) is read at a call site: identifying declaration statements of variables (Section II-F1) and computing variable values (Section II-F2).

*1) Identifying Declaration Statements of Variables:* The usage of variables is represented by the grammar in Figure 5. As the grammar shows, variables can be accessed in two ways.

**Direct variable names.** A variable which can be accessed directly by its name could be a local variable, an instance

<*variableUse*> → <*variableName*> | <*reference*> <*selectionOperator*><*variableName*>
<*selectionOperator*> → <*Operator*>
<*reference*> → <*expression*>
<*variableName*> → <*identifier*>
...

Fig. 5. A segment of Backus-Naur Form (BNF) grammar specifying the use of a variable

**Algorithm 1** Finding the declaration statement of a variable without a reference

**Auxiliary functions:**
*searchDeclAsClassFields(*$var$*, o)* : search the declaration statement of variable $var$ among the declaration statements in the field of class $o$ and superclasses of class $o$

Input: the statement of accessing a variable $var$ and the current class $o \in \psi$
Output: the declaration statement of the variable
locateDeclNoReference($var$, $o$)
1. **if**(*searchDeclInLocal* ($var$))
2. **return** the matched statement
3. **if**(*searchDeclAsParameters*($var$))
4. **return** $null$
5. **if**(*searchDeclAsClassFields*($var$))
6. **return** the matched statement
7. **if**(*searchDeclAsImportedVars*($var$)){
8. locate the class $o'$ where the variable is declared
9. **if**($o'$ not in $\psi$ )
10. **return** *null*
11. **if**(*searchDeclAsClassFields*($var$, $o'$))
12. **return** the matched statement
13.}

variable, a class variable, or a parameter variable. We locate variable's declaration statements based on this type of the declaration statement in the class.

First, the variable is considered as a local variable. We search the declaration statement of this variable in the block where the variable is used. If this is not successful, the search is extended to the outer block until the largest block of the method is reached.

If the declaration statement of the variable is not found in the method, the variable is considered as an instance variable or class variable. We search its declaration statement in the class where the variable is used but outside of any methods in the class. A variable in a class might come from its super classes. If we fail to obtain the declaration statement in the current class, we repeat the search on field members of its super classes (if they are defined in the program, i.e. not from the library or third-party packages).

If the declaration statement still is not located, the variable is considered imported from other classes. The imported variable can be used without specifying the class in which the variable is defined. For instances, the keywords *import static* is used to import a class variable in Java. Our technique also considers this usage of a variable by matching the imported class variables. If the variable is imported, the fully-qualified name of the class where the variable is defined is extracted from the full name of the imported variable. The entity of the class can be selected by retrieving its name from $\psi$ if the class is not from the library or the third-party packages. Then we search the declaration statement of the variable in this class.

The algorithm for extracting the declaration statement of a variable without a reference is shown in Algorithm 1.

**Variable names with references.** An instance or class variable can be accessed with a reference. The syntax of accessing those variables is like <*reference*><*selectionOperator*> <*variableName*>. Our static analysis considers three cases of this usage. First, the reference is keyword *this* referencing the current instance or class. We search the declaration statement of the variable in the field of the current class. Second, the reference is a class name and the variable is a class variable. We locate the class this reference represents, in which the declaration statement of the variable is searched. Last, the reference is a variable name and the variable is a class member. For this case, the declaration statement of the reference variable is first located and the data type of the reference variable is obtained. If the data type is defined in the application, we select the entity of this data type and search the declaration statement of the class member in this entity.

We designed Algorithm 2 for both usage cases: direct variable names and variable names with references. If a variable is accessed directly by its name, we invoke Algorithm 1. For the usage of a variable with a reference, the algorithm considers the reference of the variable as an instance or class variable. If the reference is other expression like the call site of a method, *null* is returned. In the case where the type of the object of the reference is not defined in the application, *null* is returned too. Similarly, Algorithm 1 searches super classes of a class for locating the declaration statement of an instance or class variable.

*2) Computing Values of Actual Parameters:* As stated above, in many cases parameters of an option read point are initialized by an expression combining a variable and a string constant or a variable expression. These expressions can be modeled by the grammar shown in Figure 6.

<*expression*> → <*S*> | <*S*> + <*S*>
<*S*> → <*variable*> |<*stringLiteral*>
...

Fig. 6. A segment of Backus-Naur Form (BNF) grammar specification for expressions of generating an option name

In order to infer which option name is used by an option read point, we propose Algorithm 3. First, the algorithm obtains operands and operators of an expression $expr$. There are two cases for an operand in this model. If the operand is a string literal, its value is stored into $str$. If the operand is a variable, we call Algorithm 2 to locate the declaration statement of the variable and obtain its initial expression $expr'$.

**Algorithm 2** Locating the declaration statement of a variable

Input: the statement with access to a variable $var$ and the class $o \in \psi$
Output: the declaration statement of the variable
locateDecl($var$, $o$)

```
1.if(var without reference)
2.   return locateDeclNoReference(var, o)
3. else{
4.        reference ← getReference(var)
5.        if(reference is a key word this)
6.              return searchDeclAsClassFields(var, o)
7.       o' ←locateClassEntity(reference)
8.        if(o' is not null)
9.              return searchDeclAsClassFields(var, o')
10.      declStat ← locateDeclNoReference(reference, o)
11.      if(declStat is not null){
12.            type ← getType(declStat)
13.            o' = locateClassEntity(type)
14.             if(o' is in ψ)
15.             return searchDeclAsClassFields(var, o')
16.      }
16.      return null
17. }
```

Then we recursively call Algorithm 3 to compute the value of $expr'$ until the initial expression of a variable is a string literal. If the values of all operands are successfully inferred, the combination of values of all operands is returned. Note that our technique does not consider expressions which do not follow the model in Figure 6. Our evaluation shows that this algorithm can infer values of most variables except when the value of a variable is generated by another method.

According to our experience, most of the time, option names are concatenated by using the operator "+" instead of calling APIs for concatenating strings. Consequently, in Figure 6, we only consider the operator "+".

In the end, a map is built between option names and the corresponding option read points. By searching for option names in the documentation we can obtain the map between documented options and their read points in the program.

### G. An Example

We use the example in Figure 1 to illustrate how our technique infers the option names used by option read points. Here a call site *conf.getBoolean(D, true)* is located at line 116 in the class file *CompositeGroupMapping.java.* The call site takes variable $D$ storing an option name as a parameter. The goal of our technique is to infer the value of variable $D$.

Algorithm 3 takes variable $D$ as input and considers it as a variable instead of a string constant. Then Algorithm 2 is called to identify the declaration statement of variable *D.* Since variable *D* is accessed directly by variable name, Algorithm 1 is called to identify its statement at line 48 in the class file *CompositeGroupMapping.java* and returns the initial expression of the statement *C + ".combined*". Algorithm

**Algorithm 3** Inferring values of actual parameters at a call site

Input: an expression $expr$
Output: a string literal $str$
inferValue($expr$)

```
1. optionName ← null
2. elements ←getElements(expr)
3. for (element element in elements){
4.     if(element is an operand){
5.         if(element is a string literal)
6.             optionName← element
7.         if(element is a variable){
8.             currentClass ←getCurrentClass(expr)
9.             declStat ←locateDecl(element, currentClass)
10.            expr' ← getInitializedExpression(declStat)
11.        optionName ← optionName + inferValue(expr')
12.        }
13.        else
14.            return null
15.    }
16.    else if(element is an operator "+")
17.        continue:
18.    else
19.        return null
20. }
21. return optionName
```

3 parses the expression. The string constant ".combined" is stored in a variable *optionName*. Then Algorithm 3 starts to infer the value of variable $C$ in the expression. Similarly, Algorithm 3 obtains *B + ".providers*", the initial expression of variable $C$. Then the string ".providers" and ".combined" is combined to ".providers.combined" and stored in variable *optionName.* Inference of the value of variable $B$ is started.

When locating the declaration statement of variable $B$, Algorithm 2 finds out that variable $B$ is not defined in the class *CompositeGroupMapping.* Then the algorithm identifies the super class of class *CompositeGroupMapping,* i.e. *GroupMappingServiceProvider,* where the declaration statement of variable $B$ is found and its initial expression *CommonConfigurationKeysPublic.A* is returned to Algorithm 3. Algorithm *3* continues to infer the value of variable $A$. While locating the declaration statement of variable $A$, Algorithm 2 finds variable $A$ is accessed with a reference *CommonConfigurationKeysPublic* which is a class name. Algorithm 2 locates class *CommonConfigurationKeysPublic,* where the declaration statement of variable $A$ is found and its initial expression "hadoop.security.service.user.name.key" is returned to Algorithm 3. Algorithm 3 finally outputs the value of variable $D$, i.e., "hadoop.security.service.user.name.key. providers.combined".

## III. Implementation

We implemented our technique as a prototype, called Option Read Points Locator (ORPLocator), which is restricted to

TABLE I
SUBJECT PROGRAMS. COLUMN "#JAVA FILES" IS THE NUMBER OF JAVA FILES. COLUMN "#LOC" IS THE NUMBER OF LINES OF CODE. THEY ARE COUNTED BY CLOC [5]. COLUMN "#OPTIONS" IS THE NUMBER OF DOCUMENTED OPTIONS FOR EACH MODULE.

| Modules | #Java Files | #LOC | #Options |
|---|---|---|---|
| Common (2.7.1) | 1,495 | 294,898 | 127 |
| HDFS (2.7.1) | 1,380 | 400,353 | 216 |
| MapReduce (2.7.1) | 1,275 | 255,670 | 172 |
| YARN (2.7.1) | 1,612 | 354,901 | 197 |
| Summary | 5,762 | 1,305,822 | 712 |

applications in Java. The tool relies on the srcML library [20]. The library computes an XML representation for source code, where the markup tags identify elements of the abstract syntax for the language. Currently srcML supports mainstream programming languages such as Java, C++, C#, and C. Our analysis targets applications in written multiple languages. Consequently we choose srcML instead of Java analysis tools such as JDT [1] and Spoon [2].

## IV. EVALUATION

We conducted an empirical study to evaluate the effectiveness and usefulness of ORPLocator and attempted to answer the following research questions.

- RQ1: How effective is ORPLocator in locating option read points and identifying option names?
- RQ2: How does ORPLocator's effectiveness compare to existing techniques?
- RQ3: What is the time cost of locating option read points by ORPLocator?

### A. Experimental Setup

*1) Subject Programs:* We evaluated ORPLocator on the Apache Hadoop framework [9] for scalable and distributed computing, which mainly consists of four modules (see Table I). Aside from these 4 modules, Hadoop consists of more than 10 tools. In the evaluation, we do not consider these tools because they have few configuration options.

There are considerations for choosing the latest version of Hadoop (version 2.7.1) as the subject program. First, Hadoop is currently widely used to store, analyze and access large amount of data and has evolved into a complex ecosystem with more than 100 related systems. The configuration mechanism in Hadoop has matured through the evolution across a number of versions from 0.14.1 to 2.7.1. Choosing Hadoop as the subject program is representative. Second, Hadoop has an abundance of configuration options.

*2) Evaluation Procedure:* The four modules are independently implemented and each of them has a configuration file which contains its own options. For each module its source code (excluding the test code) is converted into the srcML format by using the tool srcML [20]. Then we provide the resulting srcML file as well as the name of the configuration class as input to our tool ORPLocator. The ORPLocator outputs a map between option names and their read points for each module. For our evaluation, we also parse (automatically) the XML-formatted documentation of Hadoop and retrieve all option names listed there (we call there *documented options*).

### B. RQ1: Effectiveness

We evaluate the effectiveness of ORPLocator by computing the number of identified options among the documented options. We also check if a located option read point does indeed reads an option value via manual inspection of the source code. To reduce manual errors, two people performed the manual check and resolved discrepancies.

*1) Results:* The overall results obtained by ORPLocator and the numbers of documented options are shown in Table II. Column "Modules" lists modules of Hadoop we studied. Column "get-Method Callsites" shows the number of call sites of get-methods located by ORPLocator. In the part labeled "Found by ORPLocator" column "#Options" shows the numbers of detected options and column "#ORPs" the numbers of detected read points, respectively. Table part "Documented Options" reports options in documentation. Column "Total" shows the number of options in documentation. Column "#Read" represents the number of documented options which are read in the corresponding module. Note that not all documented option are read by the modules, we further discuss unread options in Section IV-B2. Finally, part "Documented and Found" shows how many of the documented options are detected ("#Found") and their percentage among all documented options ("%Found").

As shown in Table II, ORPLocator successfully locates 1861 read points yielding 1300 options (i.e., distinct option names) in the source code of the 4 modules. Only a part of these options (namely 658) are documented. Manual checking shows that the remaining, not documented options are indeed used to control the behavior of the application. As we see, ConfLocator is also able to detect active options which are not available to users. Such a comprehensive list of options can support developers in removing, changing, and adding options in the documentation.

Not every call site of a get-method is considered as an option read point by ORPLocator. First, in a benign case, such a get-method indeed does not retrieve a value of a configuration option. For instance, the class *JobConf* in MapReduce extends the configuration class, but has many get-methods which are not used for fetching option values (such as *getKeepTaskFilesPattern()*, *getNumReduceTasks(),* and so on.)

Second, such a get-method is responsible for loading an option value but this was not recognized by our tool. This is the case when the names of options loaded by these call sites are generated by methods. Such names cannot be inferred by our technique. For instance, in the call site *conf.get("hadoop.rpc.socket. factory.class." + clazz.getSimpleName()),* a part of the option name is generated by another method. Our results show that luckily this type of option read points is quite rare (Table III, Category 5).

On the other hand, ConfLocator could mistakenly interpret some call sites of methods which are not configuration APIs as option read points. To evaluate this, we manually checked all read points of documented options in Hadoop Common but

TABLE II
THE OVERALL RESULTS OF ORPLOCATOR: THE NUMBERS OF DETECTED OPTIONS AND THEIR ORPS, AND THE NUMBER OF DETECTED AND DOCUMENTED OPTIONS. THE ORPS STANDS FOR OPTION READ POINTS.

| Modules | get-method Callsites | Found by ORPLocator | | Documented Options | | Documented and Found | |
|---|---|---|---|---|---|---|---|
| | | #Options | #ORPs | Total | #Read | #Found | %Found |
| Common | 352 | 210 | 261 | 127 | 115 | 109 | 95% |
| HDFS | 586 | 367 | 524 | 216 | 214 | 206 | 96% |
| MapReduce | 909 | 423 | 631 | 172 | 169 | 162 | 96% |
| YARN | 701 | 300 | 445 | 197 | 195 | 181 | 93% |
| Overall | 2548 | 1300 | 1861 | 712 | 693 | 658 | 95% |

TABLE III
CATEGORIZATION OF DOCUMENTED OPTIONS NOT DETECTED BY ORPLOCATOR.

| Categories | Description | Common | HDFS | MapReduce | Yarn | Sum |
|---|---|---|---|---|---|---|
| 1 | Their read points are located in other modules | 11 | 0 | 0 | 2 | 13 |
| 2 | Option names are deprecated in the version | 0 | 1 | 1 | 0 | 2 |
| 3 | Option values are loaded by other configuration classes | 4 | 0 | 0 | 0 | 4 |
| 4 | Option names are mistakenly documented | 1 | 1 | 2 | 0 | 4 |
| 5 | Option names are determined in runtime | 2 | 7 | 7 | 13 | 29 |
| 6 | Call sites reading values of these options are missed | 0 | 1 | 0 | 1 | 2 |
| The number of un-located documented options for each module | | 18 | 10 | 10 | 16 | 54 |

TABLE IV
THE REAL WORLD BUGS DETECTED BY ORPLOCATOR ARE REPORTED TO DEVELOPERS AND FIXED. ALL THE BUG REPORTS CAN BE FOUND IN APACHE HADOOP REPOSITORY.

| Issue | Type | Status | Documented option names | Option names read in the source code |
|---|---|---|---|---|
| HADOOP-12704[2] | Bug | Resolved | hadoop.**work.around**.non.threadsafe.getpwuid | hadoop.**workaround**.non.threadsafe.getpwuid |
| MAPREDUCE-6605[3] | Bug | Resolved | mapreduce.map.skip.**proc.count.autoincr**<br>mapreduce.reduce.skip.**proc.count.autoincr** | mapreduce.map.skip.**proc-count.auto-incr**<br>mapreduce.reduce.skip.**proc-count.auto-incr** |
| HDFS-8274 | Bug | Resolved | nfs.dump.dir | nfs.**file**.dump.dir |

could not find such false positives. Sure, our limited evaluation does not fully exclude errors of this type, but it indicates that they are unlikely.

*2) Documented Options Not Found by ORPLocator :* We analyse the cases where option read poins are not located by ORPLocator by breaking them down into 6 categories depending on the cause of the omission (see Table III).

Each module in Hadoop has its own configuration file. An option of a module might be inserted into configuration files of other modules. Also a read point of an option in a module may exist in other modules. Category 1 covers options of a module whose read points are located in other modules. For Hadoop Common, there are 11 documented options which are unread by Common but read by HDFS or Yarn. Similarly, 2 of options in Yarn are only read by MapReduce.

Category 2 represents options which are deprecated in the current version, but are still not removed from the list of documented options. There are two deprecated options: "nfs.allow. insecure.ports" for HDFS and "mapreduce.job.counters.limit" for MapReduce. The read points of their corresponding new options are successfully located by ORPLocator.

Category 3 indicates options which are loaded by other configuration classes instead of the main configuration class of Hadoop. There are 4 options for Common loaded by the Properties class in Java.

Category 4 covers options whose names are erroneously documented, i.e. true bugs of Hadoop. We have reported these bugs to developers as three issues (Table IV). In all cases, these bugs have been confirmed and fixed.

Category 5 represents options whose names or part of names are generated by another method. For instance, in the call site *conf.get(hadoop.rpc.socket.factory.class. + clazz.getSimpleName()),* the last part of the option name is generated by the method *getSimpleName()*. ORPLocator failed to infer names of these options based on their read points. This is one of the limitations of ORPLocator.

Category 6 displays options which are read in the program but whose read points were not located by ORPLocator. The reason is that ORPLocator failed to identify call sites of get-methods which read values of these options. We further discuss it in Section IV-E.

Overall, there are only 54 options whose read points are not located by ConfLocator, i.e., only 54 out of 712 options in the documentation would need manual checking. We conclude that our tool can significantly reduce the burden of maintaining options consistency between documentation and code.

**Summary of RQ1**. The evaluation shows that ORPLocator is effective in locating option read points. It successfully detects 1861 read points of 1300 distinct options for 4 modules we studied. For documented options, it locates at least one read point for 109 out of 115 (95%) options in Common, 206 out of 214 (96%) options in HDFS, 162 out of 169 (96%) options in MapReduce, 181 out of 195 (93%) options in Yarn.

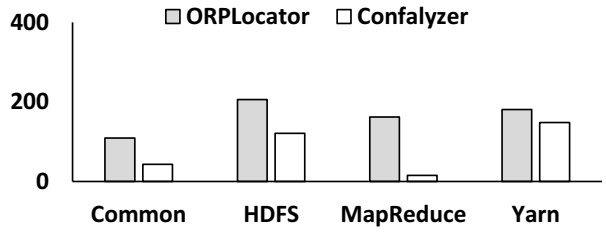


Fig. 7. The comparison on the number of documented options

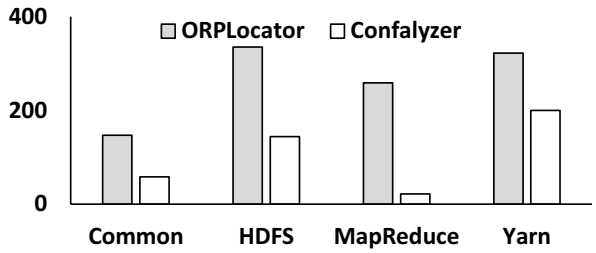


Fig. 8. The comparison on the number of read points of documented options

TABLE V
THE NUMBER OF ENTRY POINTS FOR EACH MODULE.

| Modules | Common | HDFS | MapReduce | Yarn |
|---|---|---|---|---|
| # Entry points | 22 | 25 | 8 | 13 |

TABLE VI
THE ANALYSIS TIME AND FILE SIZES OF SRCML OUTPUT.

| Modules | Common | HDFS | MapReduce | Yarn |
|---|---|---|---|---|
| File size (*mb*) | 65.6 | 86.9 | 57.8 | 90.7 |
| Time (*min*) | 69.9 | 135.7 | 90.0 | 172.2 |

### C. RQ2: Comparison with a Previous Technique

This section compares our technique with a previous technique, called Confalyzer [17], which is recently used by SCIC [4] to check software configuration inconsistencies. Within our best knowledge, Confalyzer is the most precise technique of locating option read points known in the literature.

Confalyzer, proposed by Rabkin and Katz [17], is a tool of extracting program configuration options assuming the key-value model. The core idea of Confalyzer is similar to ours and considers methods starting with *get* in the configuration class as APIs accessing option values. Then it identifies where these methods are called in the program by building a call graph and finds string parameters at these call sites, taking these parameters as options and call sites as option read points.

**Running Confalyzer**. The tool is published on GitHub[4]. For running it one needs to specify the entry points of analyzed programs. Missing entry points would decrease accuracy of detected results. To make an end-to-end comparison, we identified all possible entry points of each module by searching main methods in the source code (see Table V). Confalyzer also takes the Properties class in Java as a configuration class of Hadoop, which is not considered by ORPLocator. The options loaded by the Properties class are not considered.

**Results**. We collected all distinct options and their read points reported by Confalyzer and selected options which are documented (as well as corresponding read points). The results are shown in Figures 7 and 8. We can see that ORPLocator detects significantly more documented options and corresponding option read points than Confalyzer.

ORPLocator is more accurate than Confalyzer primarily for three reasons. First, ORPLocator detects option read points by scanning the *whole* source code to match call sites of configuration APIs. Contrary to this, Confalyzer identifies option read points by constructing a call graph using static analysis. Hadoop heavily uses reflection and this may cause incompleteness in the inferred call graph [13], [16]. Some methods containing option read points might not be included in the call graph. Second, ORPLocator considers subclasses of the base configuration class, which helps identify get-methods added in the configuration class for sub programs. Third, Confalyzer acquires option names by reading the value of actual parameters at call sites if the actual parameters are compile-time constants. Its accuracy depends on compiler optimization. ORPLocator implements a simple parser to infer values of actual parameters at call sites without this limitation.

Compared to Confalyzer, ORPLocator has another two significant advantages. First, Confalyzer requires a complete application program. Otherwise, the error due to missing classes propagates during building a call graph. We encountered this error many times when running experiments. Second, Confalyzer needs all entry points of a program. ORPLocator has none of these issues.

**Summary of RQ2**. ORPLocator produces more accurate results in locating option read points and does not require entry points of analyzed programs.

### D. RQ3: Time Cost

ORPLocator uses srcML output as the intermediate representation, which is a high-level and expensive representation compared to low-level representations such as SSA [6] or LLVM [12]. The performance is a crucial issue for ORPLocator. This section discusses the time overhead of locating option read points.

Our experiments were conducted on a laptop with Intel i7-2760QM CPU (2.40GHz) and 8 GB physical memory, running Windows 10. The analysis time and the size of srcML files for each module are shown in Table VI.

**Summary of RQ3**. As we can see, our analysis needs 1 to 3 hours for each module. This is substantial yet acceptable considering that the scale of each module is quite large, and an analysis is performed infrequently.

[2]https://issues.apache.org/jira/browse/HADOOP-12704
[3]https://issues.apache.org/jira/browse/MAPREDUCE-6605
[4]https://github.com/asrabkin/Confalyzer

### E. Discussion

*1) Limitations:* Our technique of locating option read points has some limitations. First, as we explained in Section II, ORPLocator focuses on configuration of the key-value style, with methods accessing option values have names starting with *get*, otherwise we have to manually select them. Second, the accuracy of the identification of option read points relies on the patterns how the get-methods are called in the program. The implicit invocation of get-methods will be missed by our technique, for instances, if get-methods are called by a complex call chain. Instances of configuration classes are stored in the complex data structures like a array list. Last, our technique assumes most of the option names are not generated by a method. Otherwise, the option names fail to be inferred.

*2) Threats to Validity:* The primary threat to external validity of this work concerns whether or not our results will generalize. The access techniques to configuration option values varies in different programs. This includes the construction of option names and the invocation patterns of configuration APIs. This variation may affect effectiveness of our technique. To mitigate this threat, we used Apache Hadoop, a Java program with one of the largest number of configuration options of the open-source projects. Moreover, it is a program which is developed by a variety of developers and widely used in both academy and industry. Although we need to conduct more evaluations on different programs, we believe that our results can generalize to other programs in Java which uses key-value style configuration. In the future work, we plan to instantiate our technique for other programs.

One internal validity threat regards the identification of option read points. In our technique, all call sites of get-methods whose parameters are successfully inferred are considered possible option read points. As stated in Section IV-B, a manual inspection for Hadoop Common showed no errors of this type.

Finally, the data of option read points produced by ORPLocator and Confalyzer is all available online[5].

## V. Related Work

The most closely related work falls into two categories: inconsistency detection in the source code and techniques for extracting configuration options.

### A. Inconsistency Detection

Most of the work on identifying inconsistencies in programs focuses on mismatches between documentation and source code like our work. Lin *et al.* developed iComment [21] which detects inconsistencies between source code and comments and identifies bugs in the source code and bad comments with these inconsistencies. Rubio-Gonzalez and Liblit [19] use static analysis to track down the error codes returned by system calls and to identify which of them are undocumented. Behrang et al. [4] present a technique to detect inconsistencies among multiple layers of accessing configuration options. Like ours, this work addresses detecting inconsistencies related to configuration options. But we focus on different styles of inconsistencies, namely manipulations of configuration options in different layers from the user interface down to source code. Our work targets detecting invalid configuration options in documents which are not updated in time of the evolution of source code or are mistakenly documented.

[5]https://goo.gl/7uVzYZ

### B. Extraction of Configuration Options

We are aware of several instances of prior work on extracting configuration options from source code. The closest work to ours is Confalyzer [17]. This work uses static analysis to extract a list of configuration option from source code and assisting options documentation. Although we address similar issues, our techniques are significantly different. As stated in Section IV-C, Confalyzer first marks configuration APIs in the configuration classes. Then it identifies calls to these APIs in the program by building a call graph and obtains option names by reading values of parameters of these calls. Contrary to this, our technique requires a smaller amount of dependency information. The dependency information is computed on the fly. The evaluation results show ORPLocator produces more accurate results.

PrefFinder [10] proposed by Jin *et al*., uses static analysis and dynamic analysis techniques to extract configuration options and stores them in database for query and use. The SCIC [4] exploits Confalyzer to implement the functionality of extracting configuration options in the key-value model and the tree-structured model. Differently from these, ORPLocator is a new and technique for locating option read points without relying on a call graph.

Besides, Zhou et al. [27] presents a prototype of extracting configuration knowledge from build files via symbolic analysis. Studies [22][11] analyze configuration variant and space in configurable system software and detect defects due to configuration inconsistency. All of them focus on compile-time or building time configuration issues. Different from them, our work focuses on configuration options at runtime.

## VI. Conclusion and Future Work

In this paper, we present a practical technique capable of building a map between option names in documentation and their read points in the source code. Compared to existing techniques, our analysis computes dependency information as needed without building a system dependency graph. The evaluation shows that our technique is effective in locating option read points and produces more accurate results than previous state-of-the-art. Besides, our empirical study has discovered multiple previously unknown inconsistencies between documented options and source code in Apache Hadoop.

In the future, we plan to explore other configuration models, making ORPLocator support majority of configuration models used in today's software systems and release it as a readily used tool.

## REFERENCES


[1] Jdt. http://www.eclipse.org/jdt/.

[2] Spoon. http://spoon.gforge.inria.fr/.

[3] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–11, Berkeley, CA, USA, 2010. USENIX Association.

[4] Farnaz Behrang, Myra B. Cohen, and Alessandro Orso. Users beware: Preference inconsistencies ahead. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 295–306, New York, NY, USA, 2015. ACM.

[5] CLOC. http://cloc.sourceforge.net/.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[7] Zhen Dong, Artur Andrzejak, and Kun Shao. Practical and accurate pinpointing of configuration errors using static analysis. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 171–180, Sept 2015.

[8] Zhen Dong, Mohammadreza Ghanavati, and Artur Andrzejak. Automated diagnosis of software misconfigurations based on static analysis. In *IWPD 2013 at ISSRE*, pages 162–168, 2013.

[9] Hadoop. http://hadoop.apache.org/.

[10] Dongpu Jin, Myra B. Cohen, Xiao Qu, and Brian Robinson. Preffinder: Getting the right preference in configurable software systems. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 151–162, New York, NY, USA, 2014. ACM.

[11] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824, New York, NY, USA, 2011. ACM.

[12] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 139–160, Berlin, Heidelberg, 2005. Springer-Verlag.

[14] S. Nadi, T. Berger, C. Kastner, and K. Czarnecki. Where do configuration constraints stem from? an extraction approach and an empirical study. *Software Engineering, IEEE Transactions on*, 41(8):820–841, Aug 2015.

[15] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 140–151, New York, NY, USA, 2014. ACM.

[16] Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 193–202, Washington, DC, USA, 2011. IEEE Computer Society.

[17] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 131–140, 2011.

[18] Ariel Rabkin and Randy Katz. How hadoop clusters break. *IEEE Softw.*, 30(4):88–94, July 2013.

[19] Cindy Rubio-González and Ben Liblit. Expect the unexpected: Error code mismatches between documentation and the real world. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 73–80, New York, NY, USA, 2010. ACM.

[20] srcML. http://www.srcml.org/index.html.

[21] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: Bugs or bad comments?*/. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 145–158, New York, NY, USA, 2007. ACM.

[22] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 47–60, New York, NY, USA, 2011. ACM.

[23] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 244–259, 2013.

[24] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, 2011.

[25] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 687–700, 2014.

[26] Sai Zhang and Michael D. Ernst. Automated diagnosis of software configuration errors. In *Proceedings of the 34th International Conference on Software Engineering*, San Francisco, CA, USA, May 22–24, 2013.

[27] Shurui Zhou, Jafar Al-Kofahi, Tien N. Nguyen, Christian Kästner, and Sarah Nadi. Extracting configuration knowledge from build files with symbolic analysis. In *Proceedings of the Third International Workshop on Release Engineering*, RELENG '15, pages 20–23, Piscataway, NJ, USA, 2015. IEEE Press.