Detecting Anomalies in the Order of Equally-typed Method Arguments *

Michael Pradel
Department of Computer Science
ETH Zurich

Thomas R. Gross
Department of Computer Science
ETH Zurich

ABSTRACT

In statically-typed programming languages, the compiler ensures that method arguments are passed in the expected order by checking the type of each argument. However, calls to methods with multiple equally-typed parameters slip through this check. The uncertainty about the correct argument order of equally-typed arguments can cause various problems, for example, if a programmer accidentally reverses two arguments. We present an automated, static program analysis that detects such problems without any input except for the source code of a program. The analysis leverages the observation that programmer-given identifier names convey information about the semantics of arguments, which can be used to assign equally-typed arguments to their expected position. We evaluate the approach with a large corpus of Java programs and show that our analysis finds relevant anomalies with a precision of 76%.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and debugging; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Experimentation, Languages, Reliability, Verification

Keywords

Anomaly detection, static analysis, method arguments, automated program analysis, maintenance

1. INTRODUCTION

In statically-typed programming languages, each parameter of a method has a type to ensure that only objects of the expected type are passed as arguments to the method.[1] Unfortunately, type specifications are futile when a method has multiple parameters of the same type. For example, a method `setEndPoints(int high, int low)` requires two `int` arguments. How can a programmer using this method be sure that the arguments passed to `setEndPoints()` are ordered correctly?

There are three kinds of problems related to equally-typed method arguments, which we illustrate with examples from real-world Java programs. First, a programmer can accidentally reverse arguments and pass them in the wrong order (Figure 1a). Such a mistake leads to unexpected program behavior and affects the program's *correctness*. Second, equally-typed method parameters with badly chosen names make using a method unnecessary difficult (Figure 1b). Identifier names play an important role for program understanding [22] and code quality [7]. As this is particularly true for equally-typed method parameters, inadequate names affect the program's *understandability*. Third, arguments that are unusually ordered can confuse a reader of the source code. An unusual argument order can be necessary, for example, because the program's semantics require to do the inverse of the expected (Figure 1c). Naturally, an unusual argument order raises the question whether a method call is correct. Unless a comment explains the reason for such an anomaly, it will negatively affect the program's *maintainability*.

Problems related to equally-typed arguments are hard to find. The main reason is that these problems involve the semantics of the program, which are not explicit in the source code but only exist in the mind of the programmer. Traditional compilers are oblivious to the order and names of equally-typed arguments; a program compiles without any warning as long as the types of arguments and parameters match. The problem is compounded by the fact that bugs caused by incorrectly ordered arguments may not raise an exception, and therefore remain unnoticed during testing. For instance, reversing the arguments of a call to `setEndPoints(int high, int low)` introduces a subtle semantic error, which can remain unnoticed until late in the development process.

Calls to methods with equally-typed arguments account for a significant part of all method calls. Within a corpus of programs comprising over 1.5 million lines of Java code, 11% of all method calls (77,610 out of 683,504) have two or more equally-typed arguments. That is, for 77,610 method calls

*The work presented in this paper was partially supported by the Swiss National Science Foundation under grant number 200021-134453.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '11, July 17–21, 2011, Toronto, ON, Canada
Copyright 2011 ACM 978-1-4503-0562-4/11/05 ...$10.00.

[1] We refer to formal parameters in a method declaration as *parameters* and to objects passed to methods at a call site as *arguments*.

	(a)	(b)	(c)
Program	Eclipse 3.5.1	Jython 2.5.1	Eclipse 3.5.1
Method call	`createAlignment(name, mode, Alignment.R_INNERMOST, count, sourceRestart, adjust);`	`_pow(coerce(left), value, null)`	`generateOptimizedBoolean( currentScope, codeStream, falseLabel, trueLabel, valueRequired)`
Called method	`Alignment createAlignment( String name, int mode, int count, int sourceRestart, int continuationIndent, boolean adjust)`	`PyFloat _pow( double value, double iw, PyObject modulo)`	`void generateOptimizedBoolean( BlockScope currentScope, CodeStream codeStream, Label trueLabel, Label falseLabel, boolean valueRequired)`
Comment	Bug caused by incorrect argument ordering: The highlighted arguments are not at the expected position. Triggered by our bug report, the problem has been fixed for Eclipse 3.7.	Badly chosen parameter names: The method performs exponentiation of two `double` parameters. Renaming the first two parameters to `base` and `exponent` would clarify their semantics.	Noteworthy anomaly: `trueLabel` and `falseLabel` are passed in the inverse order of the method declaration. A comment explaining this anomaly has been added in Eclipse 3.6.

Figure 1: Examples of problems related to equally-typed method arguments.

the type system cannot ensure that the arguments passed by the programmer are ordered correctly. As evidenced by various entries in public issue tracking systems and source code repositories (for example, see [1, 2, 3, 4]), programmers are susceptible to problems related to equally-typed arguments.

In this paper, we present an automated, static program analysis to detect anomalies in the order of equally-typed method arguments. The key observation that enables our approach is that one can extract implicit semantic knowledge from programmer-given names of identifiers. Our analysis leverages this knowledge by searching for inconsistencies in the names given to method arguments and method parameters. The analysis extracts identifier names from the source code of a program and compares the names used at different call sites of a method with each other using string similarity metrics. If reordering equally-typed arguments at a particular call site fits the names used at other call sites of this method significantly better, our system reports an anomaly and proposes to reorder the arguments.

The anomalies detected by our analysis correspond to the kinds of problems mentioned earlier. The analysis finds bugs caused by accidentally reversed arguments, such as Figure 1a, because the names of these arguments often deviate from normal naming practices. The analysis also reveals badly chosen parameter names, such as Figure 1b, as these names often do not allow to infer the correct argument order. Finally, the analysis detects noteworthy anomalies, such as Figure 1c, where reordering the arguments seems more in line with other calls to the method than the current argument order. Our analysis find all examples given in Figure 1.

The main appeal of the proposed technique is that it can be applied with very little effort. The analysis requires no input except for the source code of the program to analyze. Instead of relying on additional information, such as formal specifications, our technique infers knowledge about equally-typed arguments from the source code. The output of the analysis is precise in the sense that most of the reported anomalies are indeed relevant problems. Experiments with well-tested programs show a true positive rate of 72% for seeded anomalies and of 76% for real anomalies.

To our knowledge, there is no other technique to automatically find anomalies related to equally-typed arguments. However, there exist two kinds of approaches to prevent argument ordering problems. The first approach are conventions accepted by most programmers. For example, arguments of methods moving data from a source to a sink are often ordered so that the source argument is passed before the sink argument. Conventions can prevent argument ordering bugs, but require careful and disciplined programming. Also, there are cases where no obvious ordering of arguments exists, and hence, conventions are of no use. The second approach to prevent argument ordering problems is better support by the programming language. Some languages, such as Scala, allow for named arguments, which allow callers of a method to explicitly assign arguments to method parameters. For example, one can call `setEndPoints(high=myHigh, low=myLow)`. However, named arguments are not available in all languages, and also introduce additional boilerplate code, which may not be accepted by programmers.

We envision multiple usage scenarios for our approach. The analysis can be used during the development of a program as an inexpensive, automated technique to find problems related to equally-typed arguments in an early stage of development. For example, if a programmer accidentally reverses two arguments, our analysis can spot this anomaly and report a warning even before testing the source code. Another usage scenario is maintenance of mature and well-tested programs. While in this scenario, we expect few bugs to be found, anomalies are nevertheless of interest, for example, to add a comment explaining why an unusual order of arguments is correct in a particular context.

We evaluate this work with twelve real-world Java programs from the DaCapo benchmark suite [6]. This corpus of programs comprises over 1.5 million lines of source code. We experiment both with real anomalies and with seeded anomalies. The evaluation with seeded anomalies shows that argument ordering anomalies can be found with high *precision* (that is, a low false positive rate) and with acceptable *recall* (that is, an acceptable false negative rate). The sensitivity of the analysis to anomalies can be controlled by parameters. Our default configuration leads to a precision of 72% and a recall of 38%. To evaluate the analysis' ability to find real anomalies, we analyze the unmodified DaCapo pro-

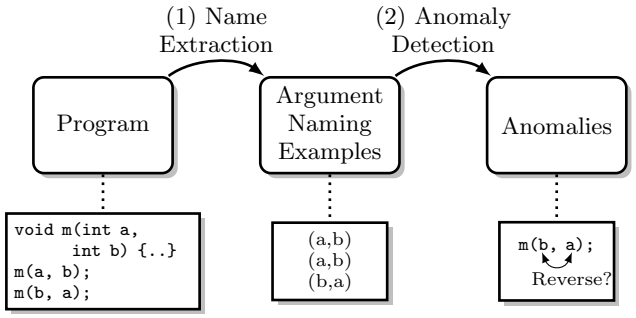

Figure 2: Overview and simple example.

grams. The analysis finds 29 anomalies, of which 22 (76%) are relevant problems.

In summary, this work makes the following contributions:

- We introduce the concept of *anomalies of equally-typed arguments*, that is, potential programming problems that relate to the order of equally-typed method arguments and that affect program correctness, program understandability, and program maintainability.
- We present an automated analysis to detect anomalies of equally-typed arguments based on programmer-given names of identifiers. The analysis can be easily applied to arbitrary programs, as it requires no input except for source code.
- We present the results of applying the analysis to over 1.5 million lines of Java source code. Our results show that relevant problems can be detected with high precision and acceptable recall, and that relevant problems exist even in mature programs.

2. APPROACH

This section presents a static program analysis to detect anomalies in the order of equally-typed method arguments. Such anomalies are often caused by problems in the source code that affect the program's correctness, understandability, or maintainability. The presented analysis is fully automated and requires no input except for source code.

Our approach consists of two steps (Figure 2). The first step, *name extraction*, gathers identifier names that programmers have given to method arguments and method parameters. The output of this step is a list of *argument naming examples* for each method with equally-typed arguments. These naming examples provide insights into the semantics of arguments and can be used to determine the order in which arguments should be passed. The second step of the analysis is *anomaly detection*. It searches anomalies in the naming examples by computing and comparing the similarities between names used at different positions. An *anomaly* occurs if the names of arguments deviate from typically used names and if a different argument order than the order in the source code seems appropriate. The output of the second step is a list of anomalies, each coming with a proposal how to reorder arguments to avoid the anomaly.

The two steps of the analysis can be viewed as a front end and a back end. While the front end, which extracts naming examples from source code, is language-dependent, the back end, which searches for anomalies, is language-independent. A benefit of this separation is that one can easily adapt our approach to other programming languages than Java. The front end analyses two language elements, methods and identifiers of method call arguments, that can be easily mapped to other commonly used languages.

2.1 Name Extraction

The goal of the name extraction step of our analysis is to gather as many examples as possible showing how programmers name the arguments passed to a method. We extract these examples from source code by analyzing method calls and method declarations. As this work focuses on problems related to equally-typed arguments, only methods with multiple parameters of the same type are considered.

The analysis traverses the abstract syntax tree and extracts from each method call two kinds of information: the signature of the called method and the names of the arguments passed to the method. In Java, different kinds of expressions can be passed as arguments. Our analysis extracts names from the following expressions:

- Local variables: The name of a local variable is simply its name.
- Field accesses: The name of a field access is the name of the accessed field, ignoring the underlying expression on which the field is accessed.
- Method calls: The name of a method call is the name of the called method, ignoring the underlying expression that yields the method receiver. In Java, getter methods are a common naming practice. As the `get` prefix does not convey any semantics relevant for our approach, we remove this prefix from all method names starting with `get`.
- Array accesses: The name of an array access is the name of the array expression, that is, ignoring the index expression.

For instance, the following method calls provide two naming examples for the arguments of `setEndPoints()`:

```
setEndPoints(highEP[i], lowEP);
setEndPoints(obj.h, getLow());

// ==> naming examples: (highEP, lowEP), (h, Low)
```

Our analysis cannot extract names from all expressions. Some argument expressions, such as literals and the addition of two numbers, are not analyzed because we cannot extract unambiguous names from them. If one or more argument of a method call has no unambiguous identifier name, the analysis ignores the method call.

Besides calls to methods, there is another source of information about the names of method arguments. Formal parameter names given in the declaration of a method are often similar to the names used at call sites. Therefore, we analyze all method declarations in a program and use formal parameter names as an additional example of how arguments are named. For example, the following method declaration gives a naming example:

```
void setEndPoints(int high, int low) { .. }

// ==> naming example: (high, low)
```

We group naming examples so that all examples for the same method signature and for the same argument type are in one group. Grouping by method signature is useful because the argument names of one method are independent of the argument names of other methods. Overloaded methods are treated as different methods, because one cannot easily map their parameters to each other. For instance, the following two variants of `m()` are treated as two methods, as we do not know how to map `a` and `b` to `x`, `y`, and `z`:

```
void m(int a, int b) {..}
void m(int x, int y, int z) {..}
```

Grouping by argument type is required because some methods expect equally-typed parameters of multiple types. For instance, the following method expects two `int` parameters and two `String` parameters:

```
void m(int length, int offset,
       String name, String msg) {..}
```

In this case, we analyze naming examples for `m()`'s `int` arguments separately from naming examples for `m()`'s `String` arguments.

In summary, the naming examples extracted by the first step of our analysis are defined as follows:

Definition 1. The *argument naming examples* of a method `m()` and a type `T` are a list of examples $(N_{c_1}, .., N_{c_k}, N_{decl})$, where

- $N_{c_1}, .., N_{c_k}$ are the lists of names given to the arguments of type `T` at call sites c_1 to c_k of `m()`, and
- N_{decl} is the list of names given to the formal parameters of type `T` in `m()`'s declaration.

2.2 Anomaly Detection

The second step of our analysis leverages the extracted argument naming examples to search for anomalies in the order in which arguments are passed to a method. An anomaly is a call of a method where arguments of the same type are named in a way that suggests a different order than the order in the source code. For instance, Figure 3a shows a list of naming examples for `setEndPoints()`'s `int` arguments. We refer to naming examples with N_1, N_2 etc. Example N_5 is an anomaly, because the first argument name, `low`, is similar to names used at the second position, while the second name, `high`, is similar to names used at the first position. Our analysis detects such anomalies and proposes a way to avoid them (here, by reversing the arguments of example N_5).

To avoid overwhelming a user of our analysis with irrelevant reports, it is important to not report every unusual argument name as an anomaly. Our analysis reports an anomaly only if changing the order of arguments makes the arguments significantly more similar to other arguments used in their respective position then using the current order. For instance, example N_2 is not an anomaly, although the name of the first argument is dissimilar to the other names of arguments used at the first position. The reason is that the second argument name of example N_2 is similar to other names at the second position; therefore, changing the argument order would not increase the overall fit of N_2 to the other naming examples.

The key idea of our analysis is that argument names used at different call sites of a method are often similar to each other. We exploit this observation to detect anomalies by comparing argument names using a string similarity metric. Such a metric returns for each pair of strings a value in the range between zero (dissimilar) and one (very similar or equal). For each argument naming example, we compute the similarity of a name used at a particular position with other names used at this position and with other names used at other positions. If a permutation of the current argument order makes the names of an example more similar to the other examples than the current order, then the analysis reports an anomaly.

An alternative to using string similarity is to check whether names are equal. However, slight variations of an argument name, such as `high` and `highEP`, would make two arguments seem different although they clearly mean the same. A string similarity metric allows for quantifying the similarity of names, and thus, to also consider variations of names.

Algorithm 1 Anomaly detection based on string distance between argument names.

Input: Argument naming examples $\mathcal{N}$
Output: Permutations $\mathcal{P}$ of argument names that resolve an anomaly

```
 1: for all N ∈ 𝒩 do
 2:   for all P ∈ permutations(N) do
 3:     score ← 0
 4:     for all n ∈ N do
 5:       for all i ∈ {1, ..., |N|} do
 6:         if (n, i) ∈ P then
 7:           score ← score + score_assign(n, i)
 8:         else
 9:           score ← score − score_assign(n, i)
10:         end if
11:       end for
12:     end for
13:     score_norm ← score/|N|
14:     if score_norm ≥ t then
15:       𝒫 ← 𝒫 ∪ P
16:     end if
17:   end for
18: end for
```

Algorithm 1 outlines our approach for detecting anomalies. The algorithm takes a list of argument naming examples as input and iterates over all examples. For each example, it goes through all possible permutations of the example's names. The core of the algorithm are lines 3 to 12. Here, we compute a score that indicates how "normal" the argument names are with a permutation P. That is, the score expresses how similar the reordered names are to other names found at their respective positions. If a permutation of the current argument order has a high score, the analysis reports an anomaly and proposes to reorder the arguments according to the permutation.

We represent a permutation as a set of assignments of arguments to a position:

$$\begin{aligned} P &\subseteq N \times \{1, \ldots, |N|\} \\ &= \{(n, i) \mid P \text{ assigns name } n \text{ to position } i\} \end{aligned}$$

The score of a permutation is computed based on a score for assignments, $score_{assign}(n, i)$, which indicates how well a name n fits position i. The overall score of a permutation

Ex.	Arg. 1	Arg. 2
N_1	high	low
N_2	h	Low
N_3	high	low
N_4	highEP	lowEP
N_5	**low**	**high**

(a) Arguments.

	high	highEP	h	low	lowEP	Low
high	1	0.93	0	0	0	0
highEP	0.93	1	0	0	0	0
h	0	0	1	0	0	0
low	0	0	0	1	0.91	1
lowEP	0	0	0	0.91	1	0.91
Low	0	0	0	1	0.91	1

(b) String similarities.

$$score_{assign}(low, 1) = max(0, 0 - 0.98) = 0$$
$$score_{assign}(low, 2) = max(0, 0.98 - 0) = 0.98$$
$$score_{assign}(high, 1) = max(0, 0.73 - 0) = 0.73$$
$$score_{assign}(high, 2) = max(0, 0 - 0.73) = 0$$

$$score_{norm}(\{high \to 1, low \to 2\}) = \frac{0.98 + 0.73 - 0 - 0}{2} = 0.86$$

(c) Score computation for example N_5.

Figure 3: Examples of anomaly detection.

is the sum of all $score_{assign}$ of assignments that are part of the permutation, minus all $score_{assign}$ of assignments that are *not* part of the permutation. That is, the assignments of a permutation influence its score positively (line 7), while all other possible assignments influence its score negatively (line 9).

Including positive and negative scores for assignments into the overall score of a permutation makes the algorithm more robust to cases where an argument seems to fit multiple positions. In this case, our algorithm cannot choose a single permutation as the most suitable, and computing a high score for any permutation would be misleading. If a permutation includes highly ranked assignments but also rejects other highly ranked assignments, the overall score includes high positive and high negative assignment scores that compensate for each other. Thus, the overall score expresses the uncertainty resulting from multiple apparently suitable permutations.

The score for assigning an argument name n to a position i, $score_{assign}(n, i)$, indicates how well a name n fits position i. To compute $score_{assign}$, we combine the string similarity between n and all other names in the naming examples of the method. At first, we compute the average similarity $simil_i$ of n to the arguments used elsewhere at position i:

$$simil_i = Avg(simil(n, n') \mid n' \text{ is argument at position } i \text{ in others examples})$$

Then, we compute the average similarity $simil_{others}$ of n to arguments used in other examples at positions other than i:

$$simil_{others} = Avg(simil(n, n') \mid n' \text{ is argument at position } j \neq i \text{ in other examples})$$

Finally, we combine both intermediate values into the result:

$$score_{assign}(n, i) = max(0, simil_i - simil_{others})$$

Subtracting $simil_{others}$ from $simil_i$ is important to adjust the result of $simil_i$ to the degree to which all arguments passed to the method resemble each other. The argument names of some methods vary a lot and one cannot infer any useful information from them. For example, the arguments passed to `Map.put(Object key, Object value)` typically have diverse names, from which our analysis cannot infer the order of arguments. To deal with such cases, we subtract $simil_{others}$, which can be thought of as a measure for noise, from $simil_i$. As a result, the score for assigning n to i is normalized to the amount of knowledge we can infer from the given names, and thus, is higher if we have more confidence in the result.

The last step of Algorithm 1 is to select permutations for which we know with a certain confidence that they make the order of arguments more "normal" than the current order. At first, the score is normalized with respect to the number of equally-typed arguments of the method (line 13). The normalized score, $score_{norm}$, ranges between zero (definitely no anomaly) and one (definitely an anomaly). Then, the algorithm selects all permutations with a score above a certain threshold t and outputs them (line 14). We discuss how to set the threshold t in Section 3.4.

The output of the described algorithm is finally transformed into anomalies to be presented to the user. The algorithm yields a set of argument permutations that each avoid an anomaly. For each such permutation, the method call from which the unusually ordered arguments have been extracted is an anomaly. We report these anomalies, along with the permutations.

2.2.1 Example

Figure 3 illustrates the anomaly detection technique with an example. Figure 3a shows five naming examples for the method `setEndPoints()`. Suppose that N_1 has been extracted from the declaration of `setEndPoints()` and that $N_2, \ldots, N_5$ are gathered from calls to the method. The algorithm traverses these naming examples and analyzes each permutation of the given argument names, that is, five permutations that each reverse the first and second argument of an example.

We compute the string similarities between all involved argument names (Figure 3b). Different string similarity metrics provide different results here. The shown numbers are computed with the *SoftTFIDF* metric. We discuss and compare several metrics in in Section 3.4.

The argument names of example N_5 deviate from the other naming examples. Their names suggest to reverse the arguments, that is, to order them according to the permutation $\{(high, 1), (low, 2)\}$. Figure 3c illustrates how our algorithm computes the score that indicates how "normal" this permutation would be. The computation combines scores for each assignment of the permutation. For example, assigning `low` to position 2 has a score of $score_{assign}(low, 2) = 0.98$, because $simil_i = 0.98$ and $simil_{others} = 0$. The overall score for the permutation is computed by adding the scores for included assignments (`low` assigned to position 2 and `high` assigned to position 1) and by subtracting the scores for all other assignments (`low` assigned to position 1 and `high` assigned to position 2). Finally, the score is normal-

Table 1: Programs used for the evaluation, their size, and how many method calls they contain (equally-typed arguments (ETA) and named, equally-typed arguments (NETA)).

Program	LOC	Method calls		
		Total	ETA	NETA
Avrora	69,393	20,276	3,179	878
Batik	186,460	47,655	6,127	2,694
DayTrader	12,325	4,613	311	103
Eclipse	289,641	280,289	26,097	13,595
FOP	102,909	32,806	2,796	1,266
H2	120,821	53,221	5,210	1,607
Jython	245,016	85,729	15,785	2,480
Lucene	124,105	41,092	5,667	1,422
PMD	60,062	21,394	2,601	507
Sunflow	21,970	8,139	1,200	537
Tomcat	161,131	54,462	4,974	1,482
Xalan	172,300	33,828	3,663	1,650
Sum	1,566,133	683,504	77,610	28,221

ized to the number of arguments in the example, giving the result of 0.86.

If 0.86 is greater than the threshold t, an anomaly is reported for the method call from which N_5 has been extracted. Since our default configuration is $t = 0.6$, the analysis reports this anomaly, together with the permutation:

```
Anomaly (confidence 86%):
    setEndPoints(low, high);
Permutation to avoid the anomaly:
    setEndPoints(high, low);
```

3. EVALUATION

The following section reports the results of evaluating our anomaly detection technique with real-world Java programs. We address the following main questions:

- *How effective is our technique in finding anomalies?* We address this question with an automated, large scale evaluation involving thousands of anomalies. Our results show a tradeoff between precision and recall. For example, one can find 14% of all anomalies with a precision of 92%, or 54% of all anomalies with a precision of 19%. Our default configuration gives 38% recall and 72% precision.

- *Which anomalies exist in mature and well-tested programs?* The default configuration of our technique finds 29 anomalies in the DaCapo benchmarks, out of which 22 (76%) are relevant problems.

- *How sensitive are the results on parameters of our analysis, such as the threshold for anomalies?* We perform a sensitivity analysis of four parameters and discuss our default configuration.

We use all programs of the DaCapo benchmark suite (version 9.12) [6].[2] Table 1 lists these programs along with their number of non-comment, non-blank lines of code (LOC). In total, the programs sum up to over 1.5 million LOC. The last three columns of Table 1 show the total number of calls, the number of calls with equally-typed arguments (ETA), and the number of calls with named, equally-typed arguments (NETA). In total, 77,610 calls have equally-typed arguments. Our analysis can extract names from 28,221 calls.

We use two separate techniques in this evaluation. On the one hand, we perform an automated evaluation with seeded anomalies (Sections 3.1 and 3.2). On the other hand, we assess the effectiveness of our approach in finding real anomalies (Section 3.3).

3.1 Automated Evaluation Technique

Parts of our evaluation use an automated evaluation technique. It is based on seeding anomalies in programs that are assumed to be free of problems related to equally-typed arguments. By seeding anomalies, we know by construction where relevant anomalies reside, so that the evaluation is not biased by a human deciding whether a reported anomaly is relevant. This automated technique allows us to evaluate our analysis on a large scale and in an objective way.

To seed an anomaly, we take a method call with equally-typed arguments and change the order of these arguments. We then assess whether the analysis detects the seeded anomaly and how many other warnings it reports. To measure the effectiveness of the analysis, we compute precision and recall. Precision means the percentage of seeded anomalies among all reported anomalies. Recall means the percentage of reported anomalies that are true positives among the seeded anomalies.

To compute precision and recall for a program, we seed anomalies one by one and run the analysis each time on the entire program. That is, we analyze a program having a single relevant anomaly and assess whether our analysis finds it. The number of true positives is one if the analysis detects the seeded anomaly, and zero otherwise. Any other reported anomalies are considered false positives. That is, precision and recall for the seeded anomaly are:

$$\text{precision} = \frac{\#\text{true positives}}{\#\text{true positives} + \#\text{false positives}}$$

$$\text{recall} = \begin{cases} 1 & \text{if the seeded anomaly is found} \\ 0 & \text{otherwise} \end{cases}$$

The overall precision for the program is the mean value over all anomalies for which some anomaly was reported by the analysis. The overall recall for the program is the mean value over all seeded anomalies. A similar evaluation technique has been used by others [26].

To make the results of the automated evaluation technique more meaningful and to ensure the technique's feasibility, we refine the described approach. First, we adapt the assumption that all analyzed programs are free of relevant anomalies by taking into account known true positives. Concretely, we know about 22 calls that expose a relevant anomaly (details in Section 3.3), and therefore, ignore them during the automated evaluation. Second, we ignore calls to methods with five or more equally-typed arguments for performance reasons. For a method with n equally-typed arguments, we run the analysis $n! - 1$ times; thus, calls with many arguments impose a significant performance problem. However, only around 1% of all calls with equally-typed ar-

[2]There are twelve programs for 14 benchmarks: DayTrader is part of the tradebeans and the tradesoap benchmarks; Lucene is part of the luindex and lusearch benchmarks in [6].

Table 2: Precision and recall of detected anomalies.

Program	Prec. (%)	Recall (%)	F-measure (%)
Avrora	95	43	59
Batik	94	38	54
DayTrader	93	45	61
Eclipse	12	42	19
FOP	47	36	41
H2	94	36	52
Jython	47	28	35
Lucene	98	32	48
PMD	94	49	64
Sunflow	49	56	52
Tomcat	49	29	36
Xalan	97	26	41
Average	72	38	47

guments have five or more arguments, so this restriction does not affect the generality of the evaluation. Third, we apply the automated evaluation only to calls with named arguments. For other calls, our technique does not apply and we know without experimenting that the analysis does not report any anomalies. Including these refinements, we seed a total of 48,543 anomalies in the corpus of programs and run the analysis for each of them.

3.2 Precision and Recall

Using the automated evaluation technique described in Section 3.1, we measure precision and recall of our analysis. Table 2 shows the results for each program. On average, the analysis obtains a precision of 72% while having a recall of 38%. That is, almost three out of four reported anomalies point to a relevant problem in the source code, while the analysis finds an acceptable ratio of all seeded anomalies. The average F-measure, that is, the harmonic mean of precision and recall, is 47%.

While the precision for many programs is over 90%, one program, Eclipse, has a remarkably low precision of only 12%. The reason is that Eclipse has many more calls with named, equally-typed method arguments than the other programs, increasing the probability to have false positives. Since we apply the analysis to the entire program for each seeded anomaly, these false positive influence the precision value each time, giving a lower precision for Eclipse.

3.3 Anomalies in Mature Programs

In addition to the automated evaluation with seeded anomalies, we also evaluate what kinds of anomalies our analysis finds in the programs as they are shipped with the DaCapo benchmarks. As these programs are mature and well-tested, we do not expect to find any serious errors related to equally-typed arguments. Such errors are likely to change the behavior of a program, and therefore, are typically found at some point while using the program. Nevertheless, our analysis can detect relevant anomalies that are worth the attention of programmers or maintainers, for example, to add a comment explaining an unusual piece of source code.

In total, our analysis reports 29 anomalies. We manually inspect these anomalies and classify them as follows:

- Contrary to our expectations, one anomaly is a bug affecting the program's correctness. Figure 1a shows the relevant source code fragments. The buggy class contains a set of public, overloaded methods that call each other and that pass multiple `int` arguments. There is an anomaly because the programmer passes the arguments in the wrong order at one call site. We were surprised to find such a bug and reported it to the Eclipse developers, who fixed it immediately (see bug 333487 in the Eclipse bug tracking system). This bug could remain unnoticed because the public method containing the buggy call is not called in the analyzed code base.

- Ten anomalies can be classified as *naming bugs* [19]. In these cases, the programmers chose parameter names that do not clarify the expected order of arguments. As identifier names are crucial for equally-typed arguments, fixing these naming bugs would improve the understandability of the program. Figure 1b shows an example of a naming bug in a method computing exponentiation. The names of the two `double` parameters, `value` and `iw`, do not reveal which of the parameters refers to the base and which to the exponent. Of course, deciding about the quality of an identifier name is difficult and to some extent a matter of taste. We therefore classify only anomalies with obviously misleading names as naming bugs and count debatable cases as false positives.

- Eleven anomalies can be classified as noteworthy and should be considered by the developers to improve the program's maintainability. These anomalies show unusual argument orders that seem incorrect but are intended in their specific context. Figure 1c is an example, where two labels, `falseLabel` and `trueLabel`, are passed as arguments. The arguments are ordered in such a way that `falseLabel` is bound to the formal parameter `trueLabel`, while `trueLabel` is bound to the formal parameter `falseLabel`. One can improve the maintainability of such source code by adding a comment explaining why a seemingly incorrect argument order is required in a particular situation. For the example in Figure 1c, a comment has been added by the developers in a later version of Eclipse.

- Finally, seven anomalies are false positives. They provide no insight to a developer and, ideally, would not be reported. Three false positives are from naming examples where all names are very similar, such as `testLocation`, `location`, `location1` and `location2`. Two false positives are from methods where any argument ordering is legal, such as the computation of a vector cross product. Since the analysis is based on heuristics and programmer-given identifier names, we cannot avoid false positives entirely.

In summary, 22 of 29 reported anomalies (76%) point to problems in the source code that are relevant for the program's correctness, understandability, or maintainability. This true positive rate is in line with the findings obtained using the automated evaluation technique (Section 3.2). Given that the analysis requires no input except for source code, this rate is quite satisfactory. Existing anomaly detection techniques, which search for other kinds of anomalies, often obtain lower true positive rates, for example, 29% [33],

Table 3: Precision and recall with and without including formal parameter names.

Parameter names	Precision	Recall	F-measure
Included	72%	38%	50%
Not included	55%	35%	43%

37.5% [27], 38% [31], and 70% [19]. For a fair comparison, we use the same procedure to obtain these numbers for each paper: at first, accumulate results from all programs analyzed in the respective paper, and then, compute the overall true positive rate.

3.4 Parameter Calibration

The presented analysis involves certain choices and parameters that have a strong influence on the overall results. Instead of arbitrarily fixing a particular configuration, we use the automated evaluation technique to assess how these parameters influence precision and recall of the analysis. The following explains the parameters and the results of a sensitivity analysis of them. We report results from varying each parameter individually while using default values for the others. Furthermore, we report on varying all parameters, which helped us to find a suitable default configuration.

3.4.1 Threshold for Anomalies

The threshold for anomalies determines how deviant from other examples a call must be to be considered an anomaly. In Algorithm 1, we call this threshold t. We experiment with values in the range between 0.1 (little deviance from other examples) and 1.0 (maximal deviance from other examples).

Figure 4a shows precision and recall with different thresholds. The results illustrate the typical tradeoff between optimizing an analysis for precision and for recall. A higher threshold leads to less reported anomalies, and hence, increases precision while decreasing recall. In contrast, one can obtain a higher recall with a lower threshold for the price of losing precision. The default configuration is a threshold of 0.6.

3.4.2 Minimum Number of Examples

The minimum number of examples determines how many naming examples for a method we require to draw any conclusions about the method at all. If we have fewer examples than this minimum number, our analysis ignores all calls to the method. Note that the names of formal parameters serve as an additional naming example. We experiment with values in the range between 2 and 10.

Figure 4b shows the influence of this parameter. Similarly to the threshold for anomalies, one must choose it considering the tradeoff between precision and recall. The default configuration is to require at least two naming examples. This value allows for analyzing methods called a single time, if formal parameter names are considered as naming examples.

3.4.3 Inclusion of Formal Parameter Names

This parameter controls whether to consider the names of formal parameters given in the declaration of a method as an additional example of names for the method's arguments. We experiment with enabling and disabling this option.

Table 3 compares precision and recall with and without including formal parameters names. Including formal parameters leads to better precision and recall values, confirming our expectation that formal parameter names are good examples to learn from. The default configuration includes formal parameter names.

3.4.4 String Similarity Metric

There are various metrics to measure the similarity or distance of two strings. We experiment with five metrics, which have been found to be successful [9].

Figure 4c compares the results obtained with the five metrics. Interestingly, choosing the string similarity metric significantly influences the overall results. Two metrics, TFIDF and SoftTFIDF, that tokenize strings before comparing them give the best F-measure. The classical Levenshtein distance, which is the minimum number of edits needed to transform a string into another, leads to a higher precision but a lower recall. Our default is to use TFIDF.

3.4.5 Influence of Parameters on Each Other

The parameters of our analysis can influence each other. For instance, including formal parameter names as an additional example increments the number of examples available for each method. Thus, deciding whether to include formal parameter names influences the choice of the minimum number of examples. To find a suitable configuration to be used as our default, we experiment with combinations of the four parameters. Based on our results from varying a single parameter at a time, we combine the following values with each other:

- Threshold for anomalies: 0.3, 0.4, 0.5, 0.6, 0.7
- Minimum number of examples: 2, 3, 4, 5
- Inclusion of formal parameters: $true$, $false$
- String similarity metric: all five metrics

In total, we analyze all programs with $5 \cdot 4 \cdot 2 \cdot 5 = 200$ configurations. Table 4 shows the configurations that maximize either precision, recall, or F-measure. Our analysis obtains the best precision (92%) using the Levenshtein string similarity metric with a high number of minimum examples and a high threshold for anomalies. Unfortunately, this configuration also leads to a very low recall (14%). The analysis obtains the best recall using the TFIDF metric, a small minimum number of examples, and a low threshold for anomalies. Finally, the analysis maximizes the F-measure using the values shown in the last line of Table 4, which is why we select this configuration as the default configuration.

3.5 Performance

Our implementation analyzes all programs from Table 1 together in less than two minutes. The smallest program, Daytrader, takes 4.9 seconds, while the largest program, Eclipse, requires 51.8 seconds. On average, analyzing a program takes 9.8 seconds (median: 5.9 seconds). The running time strongly correlates with the number of calls in a program (Pearson correlation coefficient: 93%).

4. DISCUSSION

Our approach depends on some properties of the analyzed program. First of all, there must be some code base to learn

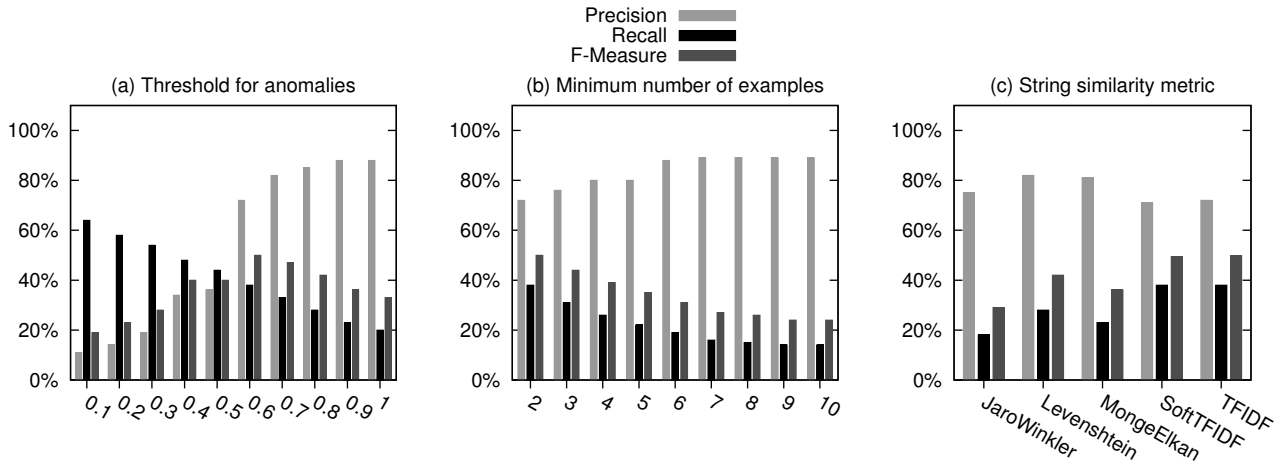

Figure 4: Parameters of the analysis and their influence on precision and recall.

Table 4: Results with different parameterizations.

Configuration				Results			Comment
Threshold	MinEx	FormParams	Metric	Prec.	Rec.	F-meas.	
0.6	5	TRUE	Levenshtein	92%	14%	24%	maximum precision
0.3	2	TRUE	TFIDF	19%	54%	28%	maximum recall
0.6	2	TRUE	TFIDF	72%	38%	47%	maximum F-measure, default configuration

from. That is, our approach cannot give useful suggestions on a newly created project. However, this limitation is mitigated by including formal parameter names as naming examples, which allow the analysis to detect an anomaly even for the first call to a method.

Another prerequisite is that arguments have extractable names. The analysis ignores arguments given via expressions that have no names, such as literals, or ambiguous names, such as mathematical expressions. For example, the analysis ignores the following call:

```
int total, current;
m(5, total - current);
```

Similarly, the analysis does not consider equally-typed arguments passed as Java varargs.

While our approach has up to 92% precision, the maximum recall is only 54%. We found two main reasons for this limitation. First, methods that are used in various contexts often have very diverse argument naming examples. For instance, arguments to `Map.put(Object key, Object value)` typically have domain-specific names, so that the analysis cannot reason about one call based on names from another. Second, the recall of our approach is bounded by the explanatory power of argument names. Short and meaningless names, such as `i` or `j`, not only can confuse programmers, but also prevent our analysis from inferring the semantics of an argument from its name.

5. RELATED WORK

Identifier names are the subject of several studies, which generally agree on the importance of well chosen names. A study [22] involving 100 human participants shows that expressive names are important for program understanding. In particular, the study shows that single letter names impede program understanding compared to appropriate full word names. Another study [7] shows that instances of bad naming practices correlate with poor code quality (measured in terms in FindBugs [20] warnings). Our analysis detects poor names of multiple equally-typed method parameters, that is, in a situation where meaningful names are crucial for programmers.

Høst and Østvold [19] propose an analysis to detect naming bugs. They combine two analyses to check whether the implementation of a method is consistent with its name. Their approach is based on implicit knowledge about method names that has been extracted from a large corpus of programs. Some of the anomalies detected by our approach are also caused by inappropriate identifier names. However, our analysis addresses argument names and the order in which arguments are passed, while Høst and Østvold analyze names of methods.

There are several approaches that address the inability of type systems to discern different usages of variables having the same type. Guo et al. [15] dynamically infer abstract types for variables of primitive types by analyzing how these variables interact, for example, through an assignment. Similarly, Hangal and Lam [17] propose a static analysis to infer dimensions that refine the type information of primitive type and string variables. Our analysis differs by analyzing programmer-given identifier names instead of the interactions of values or variables. Furthermore, we use the inferred knowledge for finding anomalies in a program. In [17], dimensions inferred from a program version that is assumed to contain no errors are used to report inconsistencies introduced by later revisions of the program. In contrast, our analysis can detect inconsistencies within a single version of a program. One could combine the techniques in [15, 17] with ours by using inferred type refinements for finding problems related to equally-typed arguments.

Lawall and Lo [21] present an analysis that infers type-like groups for `int` constants by analyzing the variables with which these constants are combined. Based on these groups, the analysis detects anomalies of variable-constant pairs, such as the incorrect use of a constant. Similar to us, Lawall and Lo address a weakness of type checkers by extracting implicit knowledge from source code. However, instead of analyzing similarities between identifier names, their approach leverages common programming idioms.

There are several approaches to explicitly refine standard types through additional information. For example, Greenfieldboyce and Foster [14] propose adding type qualifiers to Java to express atomic properties, such as that a variable is read-only. Their approach requires programmers to annotate variables with additional information, and hence, is orthogonal to an automated analysis like ours.

Erwig et al. [12] define a unit system for spreadsheet languages, which derives type-like information from headers of spreadsheet tables. Similarly to our approach, their work leverages user-provided natural language terms to search for errors caused by inconsistencies. While Erwig et al. deal with an otherwise untyped language, our approach addresses problems caused by a too coarse-grained existing type system. Another difference is that our analysis is robust against similar but different names, whereas the analysis in [12] requires header names to match precisely.

Our work belongs to a class of techniques that search interesting anomalies in a program based on the assumption that most parts of the program are correct. Engler et al. [10] statically search for violations of system-specific rules by inferring "beliefs" of programmers. Hangal and Lam [16] dynamically infer invariants and report violations of these invariants as potential bugs. A static analysis by Lu et al. [25] extracts correlations between variable accesses to find unusual pieces of code that can cause inconsistent updates and concurrency bugs. While these approaches share with our work the general idea of anomaly detection, the analysis presented here is unique in searching for problems related to equally-typed arguments.

Another group of anomaly detection techniques focuses on method calls and the order in which methods are called. PR-Miner [23] statically mines rules saying that calling a set of functions within some context implies calling another function. Chang et al. [8] detect missing condition checks by inferring graph-based rules from source code. Thummalapenta and Xie [32] target exception handling rules and how to find their violations automatically. Wasylkowski et al. [34, 33] present analyses to statically detect missing method calls. Similarly, Nguyen et al. [27] and Monperrus et al. [26] learn usage patterns to find code locations where a particular call seems to be missing. Again, all these anomaly detection approaches differ from our work in the kind of anomalies they search.

Our analysis extracts implicit knowledge from source code, instead of relying on formal specifications or other special input that is not available for many programs. Work on mining specifications follows a similar idea by inferring finite state machines describing method call sequences [5, 35, 24, 30, 13, 29, 28], algebraic specifications [18], or invariants [11] from source code or program executions. In contrast to these approaches, we do not attempt to formalize specifications in this work, but instead leverage the inferred knowledge to find anomalies.

6. CONCLUSIONS

Equally-typed method arguments slip through checks of the type system that ensure that arguments are ordered as expected by a method. Unfortunately, such arguments can be responsible for problems concerning the correctness, understandability, and maintainability of a program. In this work, we present an automated analysis to detect anomalies related to equally-typed arguments. The analysis is based on similarities between programmer-given identifier names. Experiments with a large corpus of Java programs show that the analysis finds relevant problems with high precision.

The presented approach can serve as a low-cost tool for programmers and maintainers. During development, programmers can use the analysis to find problems related to equally-typed arguments early. For example, one can think of an IDE extension that highlights unusually ordered arguments just as a programmer types a method call. During maintenance of programs, our approach can find noteworthy pieces of source code that should be enhanced, for example, by adding a comment explaining an anomaly.

Our work is part of a stream of research on extracting implicit knowledge from source code, program executions, or other software engineering artifacts. With few exceptions, identifier names have not yet been used in this context. Our work contributes by leveraging implicit knowledge from identifier names for detecting anomalies.

For our implementation see:

`http://mp.binaervarianz.de/issta2011`

Acknowledgments

Thanks to Zoltán Majó and the anonymous reviewers for their insightful comments and suggestions.

7. REFERENCES

[1] `https://issues.apache.org/jira/browse/HADOOP-4732`.

[2] `http://issues.liferay.com/browse/LPS-3890`.

[3] JBoss SVN repository. Revisions 58536, 58764, and 60357.

[4] JikesRVM SVN repository. Revisions 10263 and 13935.

[5] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages (POPL)*, pages 4–16. ACM, 2002.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190. ACM, 2006.

[7] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Relating identifier naming flaws and code quality: An empirical study. In *Working Conference on Reverse Engineering (WCRE)*, pages 31–35. IEEE, 2009.

[8] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *International*

Symposium on Software Testing and Analysis (ISSTA), pages 163–173. ACM, 2007.

[9] W. W. Cohen, P. D. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Workshop on Information Integration on the Web (IIWeb)*, pages 73–78, 2003.

[10] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles (SOSP)*, pages 57–72. ACM, 2001.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):213–224, 2001.

[12] M. Erwig and M. M. Burnett. Adding apples and oranges. In *Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 173–191. Springer, 2002.

[13] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Symposium on Foundations of Software Engineering (FSE)*, pages 339–349. ACM, 2008.

[14] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 321–336. ACM, 2007.

[15] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 255–265. ACM, 2006.

[16] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering (ICSE)*, pages 291–301. ACM, 2002.

[17] S. Hangal and M. S. Lam. Automatic dimension inference and checking for object-oriented programs. In *International Conference on Software Engineering (ICSE)*, pages 155–165. IEEE, 2009.

[18] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering*, 33(8):526–543, 2007.

[19] E. W. Høst and B. M. Østvold. Debugging method names. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 294–317. Springer, 2009.

[20] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 132–136. ACM, 2004.

[21] J. L. Lawall and D. Lo. An automated approach for finding variable-constant pairing bugs. In *International Conference on Automated Software Engineering (ASE)*, pages 103–112. ACM, 2010.

[22] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? A study of identifiers. In *International Conference on Program Comprehension (ICPC)*, pages 3–12. IEEE, 2006.

[23] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 306–315. ACM, 2005.

[24] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *Symposium on Foundations of Software Engineering (FSE)*, pages 265–275. ACM, 2006.

[25] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Symposium on Operating Systems Principles (SOSP)*, pages 103–116. ACM, 2007.

[26] M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–25. Springer, 2010.

[27] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 383–392. ACM, 2009.

[28] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *International Conference on Software Maintenance (ICSM)*, pages 1–10. IEEE, 2010.

[29] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *International Conference on Automated Software Engineering (ASE)*, pages 371–382. IEEE, 2009.

[30] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 174–184. ACM, 2007.

[31] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *International Conference on Automated Software Engineering (ASE)*, pages 283–294. IEEE, 2009.

[32] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *International Conference on Software Engineering (ICSE)*, pages 496–506. IEEE, 2009.

[33] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *International Conference on Automated Software Engineering (ASE)*, pages 295–306. IEEE, 2009.

[34] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 35–44. ACM, 2007.

[35] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Symposium on Software Testing and Analysis (ISSTA)*, pages 218–228. ACM, 2002.