Name-based Analysis of Equally Typed Method Arguments

Michael Pradel, *Member, IEEE,* Thomas R. Gross, *Member, IEEE,*

Abstract—When calling a method that requires multiple arguments, programmers must pass the arguments in the expected order. For statically typed languages, the compiler helps programmers by checking that the type of each argument matches the type of the formal parameter. Unfortunately, types are futile for methods with multiple parameters of the same type. How can a programmer check that equally typed arguments are passed in the correct order? This paper presents two simple yet effective, static program analyses that detect problems related to the order of equally typed arguments. The key idea is to leverage identifier names to infer the semantics of arguments and their intended positions. The analyses reveal problems that affect the correctness, understandability, and maintainability of a program, such as accidentally reversed arguments and misleading parameter names. Most parts of the analyses are language-agnostic. We evaluate the approach with 24 real-world programs written in Java and C. Our results show the analyses to be effective and efficient. One analysis reveals anomalies in the order of equally typed arguments; it finds 54 relevant problems with a precision of 82%. The other analysis warns about misleading parameter names and finds 31 naming bugs with a precision of 39%.

Index Terms—Testing and Debugging, Maintenance, Documentation, Static Program Analysis, Anomaly Detection, Method Arguments

1 Introduction

In statically typed programming languages, method parameters have types to ensure that each argument passed to a method has the expected type.[1] Unfortunately, type specifications are useless when a method has multiple parameters of the same type. For example, a method `setEndPoints(int high, int low)` requires two `int` arguments. If a programmer accidentally calls this method with incorrectly ordered arguments, the compiler has no means to warn her. Can we support programmers in ordering equally typed arguments correctly?

There are three kinds of problems related to equally typed method arguments, which we illustrate with examples from real-world Java and C programs.

1) 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*.
2) 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 1b). Naturally, an unusual argument order raises the question whether a method call site is correct. Unless a comment explains the reason for such an anomaly, it will negatively affect the program's *maintainability*.
3) Equally typed method parameters with badly chosen names make using a method unnecessary difficult (Figure 1c). Identifier names play an important role for program understanding [24] and code quality [7]. Since this is particularly true for equally typed method parameters, inadequate names affect the program's *understandability*.

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

Call sites of methods with equally typed arguments account for a significant part of all method call sites. Within a corpus of programs comprising 1.6 million lines of Java code (the DaCapo benchmarks [6]), 11% of all method call sites (77,610 out of 683,504) have two or more equally typed arguments. That is, for 77,610 method call sites the type system cannot ensure that the arguments passed by the programmer are ordered correctly. The problem is even more severe for C programs. In a collection of 620 thousand lines of C code (the SPEC CPU benchmarks [20]), 26% of all call sites (25,219 out of

1. When saying method, we mean both functions (as in C) and methods (as in Java). 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	gcc 3.2

Call site	`createAlignment(name, mode,`
	`Alignment.R_INNERMOST,`
	`count, sourceRestart,`
	`adjust);`
		`do_compare_and_jump(exp,`
		`rcmp, rcmp,`
		`if_true_label,`
		`if_false_label)`
			`_pow(coerce(left), value, null)`

Called method	`Alignment createAlignment(`
	`String name, int mode,`
	`int count, int sourceRestart,`
	`int continuationIndent,`
	`boolean adjust)`
		`void do_compare_and_jump(`
		`tree exp,`
		`enum rtx_code signed_code,`
		`enum rtx_code unsigned_code,`
		`rtx if_false_label,`
		`rtx if_true_label)`
			`PyFloat _pow(`
			`double value, double iw,`
			`PyObject modulo)`

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.	Noteworthy anomaly: `if_true_label` and `if_false_label` are passed in the inverse order of the method declaration. A comment explaining this anomaly makes maintaining the code easier.	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.

Fig. 1: Examples of problems related to equally typed method arguments found in Java and C programs.

96,633) have equally typed arguments. On average, these programs contain a call site with unchecked argument order every 24 lines of code. As evidenced by various entries in public issue tracking systems and source code repositories (for example, see [1]–[4]), programmers are susceptible to making errors related to equally typed arguments.

In this paper, we present two automatic, mostly language-agnostic, static program analyses to detect problems related to equally typed method parameters. First, we present an *anomaly detection* analysis that searches for anomalies in the order of equally typed method arguments. Second, we present a *naming bug detection* analysis that searches for parameter names that fail to guide programmers in assigning arguments to the right position. The key observation that enables our approach is that programmer-given names of identifiers convey implicit semantic knowledge about arguments. Our analyses leverage this knowledge by searching for inconsistencies in the names given to method arguments and method parameters. The analyses extract identifier names from the source code of a program and compare the names used at different call sites of a method with each other using string similarity metrics. The anomaly detection reports a warning if reordering equally typed arguments at a particular call site fits the names used at other call sites of this method significantly better. The naming bug detection warns about methods where many call sites agree on a naming scheme for arguments but where the parameter names do not reflect the naming scheme.

The problems detected by our analyses correspond to the kinds of problems mentioned earlier. The anomaly detection finds correctness 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 detects noteworthy anomalies, such as Figure 1b, where reordering the arguments seems more in line with other call sites of the method than the current argument order. Finally, both analyses reveal badly chosen parameter names, such as Figure 1c. We have found all examples in Figure 1 with the automatic analyses.

The main appeal of the proposed techniques is that they can be applied with very little effort. The analyses require no input except for the source code of the program to analyze. Instead of relying on additional information, such as formal specifications, our techniques infer knowledge about equally typed arguments from the source code. The output of the analyses are precise in the sense that most of the reported anomalies are relevant for developers. We consider a warning as relevant if it corresponds to one of the three kinds of problems illustrated in Figure 1: correctness bugs, noteworthy anomalies, and badly chosen parameter names.

To our knowledge, there is no other technique to automatically find problems related to equally typed arguments. However, there exist two approaches to prevent argument ordering problems. The first approach are conventions. For example, the arguments of a method moving data from a source to a sink are typically ordered so that the source argument is passed before the sink argument. Conventions can prevent argument ordering bugs but require careful and disciplined programming. Unfortunately, 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 Smalltalk and Scala, allow for named arguments, where callers of a method 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 evaluate this work with 12 real-world Java programs from the DaCapo benchmarks [6] and 12 real-world C programs from the SPEC CPU benchmarks [20]. The anomaly detection reveals 54 relevant problems, including eleven correctness bugs; 82% of the reported warnings are true positives. To measure how many anomalies the analysis misses, we seed anomalies. The analysis finds 74% of them. The naming bug detection identifies 31 naming bugs and has a true positive rate of 39%. Most results are comparable for Java and C, which shows that the approach works well for multiple languages.

This work builds upon a previously presented analysis [32], which we extend, refine, and complement in the following ways:

- We show that the approach is language agnostic by devising and evaluating a variant of the analysis for C, in addition to the previously presented Java variant. The core analysis techniques are the same for both languages and turn out to be useful for both.
- We complement the anomaly detection with an analysis focused on naming bugs. The existing anomaly detection also finds some naming bugs because badly chosen names often do not allow for inferring the correct argument order. The additional naming bug detection finds further naming bugs that the anomaly detection misses.
- We refine the anomaly detection in various ways to improve both its precision, that is, how many of the reported warnings are relevant, and its recall, that is, how many anomalies the analysis detects. As a result, recall improves from 38% to 74% and precision improves from 76% to 84%. That is, the refined approach finds problems missed by the analysis presented in [32] and reports fewer false positives.

We envision two usage scenarios for our approach. During the development of a program, the analyses provides 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, the anomaly detection 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, problems related to equally typed arguments are nevertheless of interest, for example, to add a comment explaining why an unusual order of arguments is correct in a particular context or to improve badly chosen parameter names.

In summary, this work makes the following contributions:

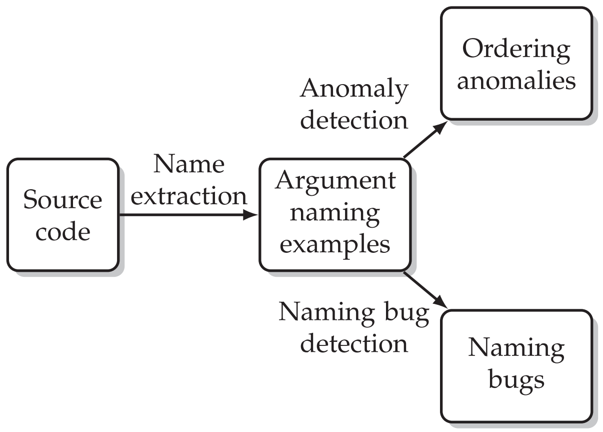

Fig. 2: Overview.

- We introduce the concept of *anomalies of equally typed arguments*, that is, potential programming problems related to the order of equally typed method arguments and that affect program correctness, program maintainability, and program understandability.
- We present two automatic analyses to detect anomalies of equally typed arguments based on programmer-given names of identifiers. The analyses can be easily applied to arbitrary programs because they require no input except for source code.
- We show that the approach is mostly language-agnostic by applying it to Java and C programs.
- We improve recall and precision of a previously presented analysis [32].
- We present the results of evaluating the approach with 2.2 million lines of Java and C source code. Our results show that relevant problems can be detected with high precision and recall, and that relevant problems exist even in mature programs.

2 Overview of the Approach

This paper presents two static analyses to detect anomalies involving equally typed method arguments. Such anomalies often correspond to problems in the source code that affect the program's correctness, maintainability, or understandability. The presented analyses are fully automatic and require no input except for source code.

Figure 2 gives an overview of our approach. The first step, *name extraction*, gathers identifier names that programmers have given to method arguments and method parameters (Section 3). The output of this step is a list of *argument naming examples* for each method with equally typed parameters.

The *anomaly detection* analysis searches for anomalies in the order of arguments by leveraging the observation that naming examples provide insights into the semantics of arguments (Section 4). The analysis searches for anomalies in the naming examples by computing the similarities between names used at different positions.

An *anomaly* occurs if the names of arguments deviate from typically used names in such a way that a different argument order than the order in the source code seems appropriate. The output of the anomaly detection are warnings about suspicious call sites, each coming with a proposal how to reorder arguments to avoid the anomaly.

The *naming bug detection* analyzes naming examples to find *naming bugs*, that is, parameter names that fail to help developers in ordering equally typed method arguments correctly (Section 5). The key idea of this analysis is to search for methods where many call sites agree on a particular naming scheme but where the parameter names do not reflect this naming scheme. The output of the naming bug detection is a list of methods that are likely to have inappropriate parameter names.

The approach shown in Figure 2 can be viewed as a front end and two back ends. While the front end, which extracts naming examples from source code, is language-dependent, the back ends, which search for anomalies and naming bugs, are language-independent. A benefit of this separation is that one can easily adapt our approach to different programming languages. In this paper, we present front ends for Java and C.

3 Name Extraction

The goal of the name extraction step of our approach is to gather as many examples as possible that show how programmers name the arguments passed to a method. We extract these examples from source code by analyzing method call sites 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 site two kinds of information: the signature of the called method and the names of the arguments passed to the method. The part of the analysis that extracts names from arguments is language-specific. In each language, different kinds of expressions can be passed as arguments. In the following, we present name extraction techniques for Java and C. We begin with kinds of argument expressions supported by both Java and C, and afterwards discuss language-specific argument expressions.

3.1 Extracting Names from Source Code

3.1.1 Java and C Language

The analysis extracts names from the following expressions:

- Identifiers (for example, local variables): The name of an identifier is simply the identifier itself.
- Array accesses: The name of an array access is the name of the array expression, that is, ignoring the index expression.
- Casts: The name of a cast expression is the name of the casted expression, ignoring the type to which it is cast.

3.1.2 Java Language

Our analysis extracts names from the following Java-specific argument expressions:

- 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. This includes fields of `this` and super fields.
- Method call sites (with return value passed as an argument): The name of a method call site 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`.

For instance, the following Java method call sites provide three naming examples:

```
setEndPoints(x.highEP[i], lowEP);       // (highEP,lowEP)
setEndPoints(obj.h, getLow());          // (h,Low)
setEndPoints(getHighs()[5], (int) low) // (Highs,low)
```

The analysis currently does not consider equally typed arguments passed as Java varargs.

3.1.3 C Language

For C programs, the analysis extracts the following names in the addition to those described in Section 3.1.1:

- Access to members of structs and unions: The name of a member access is the name of the accessed member, ignoring the underlying expression on which the member is accessed.
- Address-of operators: The name of an expression of the form `&x` is the name of `x`.
- Dereference operators: The name of an expression of the form `*x` is the name of `x`.
- Method call sites: The name of a method call site is the name of the called method, ignoring the underlying expression that yields the receiver.

For example, consider the following C method call sites and the extracted naming examples:

```
setEndPoints(h, low[3]);            // (h,low)
setEndPoints(foo->high, &low);      // (high,low)
setEndPoints(*highPtr, f().low()); // (highPtr,low)
```

3.1.4 Unnamed Arguments

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 site:

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

3.2 Creating Naming Examples

Besides call sites of 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) {..} // (high, low)
```

The analysis groups 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 because the analysis cannot 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: Argument naming examples
The *argument naming examples* of a method `m()` and a type `T` consist of the set $\{N_{c_1}, .., N_{c_k}, N_{decl}\}$, where

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

4 Anomaly Detection

The anomaly detection 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 site 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 argument 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 significantly 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.

4.1 Algorithm

Algorithm 1 outlines our approach for detecting anomalies. The algorithm takes a list of argument naming examples as input and outputs a set of warnings that each consist of a proposed permutation and a confidence value. The algorithm iterates over all examples, and for each example, goes through all possible permutations of the example's names. The core of the algorithm are lines 6 to 18. Here, it computes a score, $permScore_{norm}$, 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 score that is significantly higher than the score $currentScore$ of the current argument order, then the analysis reports an anomaly and proposes to reorder the arguments according to the permutation.

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

```
Input: Argument naming examples $\mathcal{N}$
Output: Warnings $\mathcal{W}$, each being a pair of a permutation
    that resolves an anomaly and a confidence value
1: for all $N \in \mathcal{N}$ do
2:   $currentScore \leftarrow 0.0$
3:   $bestScore \leftarrow 0.0$
4:   $P_{best} \leftarrow$ current permutation
5:   for all $P \in permutations(N)$ do
6:     $permScore \leftarrow 0$
7:     for all $n \in N$ do
8:       $posScore \leftarrow 0$
9:       for all $i \in \{1, \ldots, |N|\}$ do
10:        if $(n, i) \in P$ then
11:          $posScore \leftarrow posScore + score_{assign}(n, i)$
12:        else
13:          $posScore \leftarrow posScore - score_{assign}(n, i)$
14:        end if
15:      end for
16:      $posScore_{norm} \leftarrow (posScore + |N| - 1)/|N|$
17:      $permScore \leftarrow permScore + posScore_{norm}$
18:    end for
19:    $permScore_{norm} \leftarrow permScore/|N|$
20:    if $isCurrent(P)$ then
21:      $currentScore \leftarrow permScore_{norm}$
22:    end if
23:    if $permScore_{norm} > bestScore$ then
24:      $bestScore \leftarrow permScore_{norm}$
25:      $P_{best} = P$
26:    end if
27:  end for
28:  $conf \leftarrow bestScore - currentScore$
29:  if $conf > t$ then
30:    $\mathcal{W} \leftarrow \mathcal{W} \cup \{(P_{best}, conf)\}$
31:  end if
32: end for
```

Permutations

We represent a permutation as a set of assignments of argument names to a position:

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

For example, the naming example N_5 as shown in Figure 3a is represented as:

$$\{(low, 1), (high, 2)\}$$

Inverting the two arguments is represented as:

$$\{(low, 2), (high, 1)\}$$

Permutation score

We compute a score $permScore$ for each permutation. This score is the sum of scores for each argument position. For example, the score for a permutation of two arguments is the sum of a score for the first position and a score for the second position. Since the permutation score depends on the number of equally typed arguments, we normalize it into the range $[0, 1]$ (line 19).

Position score

The score $posScore$ for a position depends on a score $score_{assign}(n, i)$ that indicates how well a name n fits position i. The $posScore$ for a position i is the score for assigning the name to the position proposed by the permutation minus the sum of scores for all assignments of this name to other positions. That is, the assignments of a permutation influence its score positively (line 11), while all other possible assignments influence its score negatively (line 13).

Including positive and negative scores for assignments into the overall score of a position makes the algorithm more robust to cases where an argument seems to fit multiple positions. In this case, our algorithm cannot choose a single position as the most suitable, and computing a high score for any position 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.

Similar to the permutation score, the position score depends on the number of equally typed arguments and therefore is normalized into the range $[0, 1]$ (line 16).

Assignment score

The score $score_{assign}(n, i)$ for assigning an argument name n to a position 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 $simil_i^n$ of n to the arguments used elsewhere at position i:

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

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

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

Finally, we combine both intermediate values into the result:

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

Subtracting $simil_{others}^n$ from $simil_i^n$ is important to adjust the result of $simil_i^n$ 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. To

Ex.	Pos. 1	Pos. 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	0.71	0	0	0	0
highEP	0.71	1	0	0	0.60	0
h	0	0	1	0	0	0
low	0	0	0	1	0.53	1
lowEP	0	0.60	0	0.53	1	0.53
Low	0	0	0	1	0.53	1

(b) String similarities.

$$\begin{aligned}
score_{assign}(low, 1) &= max(0, 0 - 0.85) = & 0 \\
score_{assign}(low, 2) &= max(0, 0.85 - 0) = & 0.85 \\
score_{assign}(high, 1) &= max(0, 0.62 - 0) = & 0.62 \\
score_{assign}(high, 2) &= max(0, 0 - 0.62) = & 0
\end{aligned}$$

$$\begin{aligned}
&permScore_{norm} \text{ of } \{high \mapsto 1, low \mapsto 2\} \\
=&\frac{\frac{0.85-0+2-1}{2} + \frac{0.62-0+2-1}{2}}{2} = 0.87 \\
&permScore_{norm} \text{ of } \{low \mapsto 1, high \mapsto 2\} \\
=&\frac{\frac{0-0.85+2-1}{2} + \frac{0-0.62+2-1}{2}}{2} = 0.13 \\
&conf \text{ of } \{high \mapsto 1, low \mapsto 2\} = 0.87 - 0.13 = 0.74
\end{aligned}$$

(c) Score computation for example N_5.

Fig. 3: Examples of anomaly detection.

deal with such cases, we subtract $simil^n_{others}$, which can be thought of as a measure for noise, from $simil^n_i$. 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.

Best versus current permutation

The last step of Algorithm 1 is to select permutations for which the analysis is confident that they make the order of arguments more "normal" than the current order. While computing scores for permutations of a naming example, the algorithm stores the score of the current permutation into $currentScore$ and the maximum score over all permutation into $bestScore$. If the best score is at least t larger than the current score, then the algorithm adds a warning to the set $\mathcal{W}$ of reported warning. The warning proposes the permutation P_{best}, which has the best score among all permutations. We discuss how to set the threshold t in Section 7.1.4.

The output of the algorithm a set of warnings. Each warning consists of a permutations that avoids an anomaly and a confidence value that indicates how confident the analysis is that the warning should be reported.

4.2 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 call sites of 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 *TFIDF* metric. We discuss and compare several metrics in Section 7.1.4.

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 scores that indicate how "normal" this permutation and the current permutation are. The computation combines scores for each assignment of the permutation. For example, assigning `low` to position 2 has a score of $score_{assign}(low, 2) = 0.85$, because $simil^{low}_2 = 0.85$ and $simil^{low}_{others} = 0$. The overall score for the permutation is 0.87, whereas the score for the current permutation is 0.13. That is, the confidence for inverting the arguments of N_5 is 0.74. Because the confidence is greater than our default threshold $t = 0.4$, the analysis reports a warning about N_5 and suggests to invert the two arguments.

4.3 Refinements

The anomaly detection presented in Algorithm 1 can be used as described so far and we show in previous work that it can effectively detect anomalies that point to real problems [32]. In the following, we describe several refinements of the approach that allow for detecting more anomalies while reporting less false positives. Some of the refinements depend on configurable parameters; we evaluate the sensitivity of the analysis to these parameters in Section 7.1.4.

Example families

The anomaly detection described so far analyses all naming examples of a method together. Often, there are multiple common naming schemes for arguments of a method and analyzing them separately exposes more about the semantics of arguments than analyzing all together. For example, Figure 4a lists naming examples for `String.substring(int,int)`. Obviously (for a human), the last example is an anomaly that the analysis should report. However, some naming examples use a "start"-"end" naming scheme, whereas others use a "first"-"last" naming scheme. If the anomaly detection analyzes all naming examples together, it may miss the anomaly because permuting `end` and `start` does not have enough confidence.

We refine the approach presented so far by applying the anomaly detection to subsets of all naming examples.

The subsets are chosen in such a way that they contain examples with similar names, that is, examples that are likely to use the same naming scheme. We call such subsets of naming examples *example families*.

Our algorithm for creating example families depends on a similarity measure for naming examples. Naming examples are similar to each other if their sets of argument names are similar. To compute the similarity of naming examples, we extend the string similarity measure to sets of strings as follows. Given two sets of strings $\mathcal{S}_1$ and $\mathcal{S}_2$, we find the pairwise mapping between strings from $\mathcal{S}_1$ and $\mathcal{S}_2$ where the average similarity between pairs is maximal and then report this similarity. For example, given $\mathcal{S}_1 = \{start, end\}$ and $\mathcal{S}_2 = \{startPos, endPos\}$, we find that $start \mapsto startPos$ and $end \mapsto endPos$ gives a similarity of 48%, whereas $start \mapsto endPos$ and $end \mapsto startPos$ gives a similarity of 0%. Thus, the similarity of $\mathcal{S}_1$ and $\mathcal{S}_2$ is 48%.

Based on the similarity measure for naming examples, we compute example families for each naming example observed for a method. For a particular naming example N, we perform two steps. At first, we compute the similarity of N to the other naming examples of the method. Then, we create example families that contain N and that fulfill two conditions. First, all naming examples in a family must have a similarity to N below some *coherence* value. Our implementation considers ten coherence levels (10%, 20%, ..., 100%), that is, we try to create families with 10% coherence, 20% coherence, etc. The rationale for trying different coherence levels is that there is no level that works best for all programs. Second, the number of naming examples in the family must be above a configurable threshold, which we discuss in Section 7.1.4.

For the naming examples in Table 4a, the algorithm creates four families for the last call site in the table. These families are listed in Figure 4b. The first family contains only naming examples with exactly the same argument names as the last call site. The other families gradually add more and more call sites and the fourth family contains all naming examples.

The refined anomaly detection checks whether a naming example is an anomaly by analyzing each example family separately. If the analysis finds an anomaly for any of the families, then the anomaly is reported. This refined approach increases the recall of our approach compared to the approach described in [32] because the anomaly detection now detects anomalies that are obvious when analyzing a subset of all naming examples but that previously have been hidden among other naming examples.

Subsets of all parameters

The approach described so far considers all parameters of equal type at once. For methods with many equally typed parameters, this approach leads to two problems. First, an anomaly that involves only a subset of all parameters may be hidden within the other parameters. For example, consider a method with seven equally typed parameters and a call site of the method from which we extract the naming example `a,b,c,d,f,e,g`, where `f` and `e` should be inverted. If the anomaly detection considers all seven parameters, the anomaly may remain unnoticed because the difference between the scores for `a,b,c,d,f,e,g` and `a,b,c,d,e,f,g` is below the threshold. In contrast, analyzing only `f,e` and `e,f` may reveal the anomaly. Second, the number of permutations of naming examples of methods with many parameters is a scalability problem: Each naming example of a method with k parameters has $k!$ permutations.

To address these two problems, we refine our approach by considering subsets of all equally typed parameters of a method. Instead of analyzing all parameters at once, the refined analysis considers all subsets with at least two parameters and at most four parameters.[2] For each such subset, we run the anomaly detection separately. If for any of the subsets the analysis detects an anomaly for a call site, then this anomaly is reported to the user. If the analysis finds multiple anomalies for the same call site and argument type, then only the anomaly with the highest confidence is reported.

Considering subsets of all parameters addresses the problem described above. First, the refined analysis discovers anomalies that otherwise would be hidden among other parameters. For the example above, the refined analysis may report that `f` and `e` should be inverted because it analyzes these two arguments without considering the other parameters of the method. Second, the refined analysis improves the scalability of the approach for methods with many equally typed parameters. Instead of considering $k!$ permutations, it considers only $2! \cdot \binom{k}{2} + 3! \cdot \binom{k}{3} + 4! \cdot \binom{k}{4}$ permutations. For example, for $k = 10$, the refinement reduces the number of permutations to consider from 3,628,800 to 5,850.

Higher weight for parameter names

Parameter names should guide a developer that calls a method in assigning arguments to the correct position. To achieve this goal, parameter names often have descriptive names that convey valuable information about the semantics of the expected arguments. We found that parameter names often have more meaningful names than argument names and therefore give naming examples extracted from parameters a higher weight than naming examples extracted from arguments. Our default configuration (discussed in Section 7.1.4) considers parameters to be five times as important as arguments.

Method filtering

The approach presented so far considers all methods with equally typed arguments. However, some methods implement commutative operations, that is, any order of

2. We choose the upper bound four as a pragmatic compromise between ensuring scalability and detecting anomalies that involve more than two arguments.

All examples

Position 1	Position 2
start	end
startIndex	endIndex
first	last
start	end
startPos	lastPos
start	end
first	last
first	last
end	*start*

(a) Naming examples.

Coherence: 100%

Position 1	Position 2
start	end
start	end
start	end
end	*start*

Coherence: 40%

Position 1	Position 2
start	end
startIndex	endIndex
start	end
start	end
end	*start*

Coherence: 20%

Position 1	Position 2
start	end
startIndex	endIndex
start	end
startPos	lastPos
start	end
end	*start*

Coherence: 0%

Position 1	Position 2
start	end
startIndex	endIndex
first	last
start	end
startPos	lastPos
start	end
first	last
first	last
end	*start*

(b) Example families for the naming example that is printed in italics in Figure 4a.

Fig. 4: Example families.

arguments gives the same results. Reporting an anomaly for such a method is certainly a false positive and the analysis should not consider such methods. We observed that many methods that implement commutative operations have generic or single character parameters names, such as `op0,op1,op2` or `a,b`. The refined analysis heuristically checks for methods with such parameter names and automatically excludes them from the analysis. Being a heuristic, this refinement may accidentally remove methods that do not implement commutative operations. However, we find this risk to be outweighed by the increase in precision that the analysis achieves by filtering methods.

Filtering contradicting warnings

Some of the warnings found by Algorithm 1 may contradict each other. For example, consider two call sites `m(a,b)` and `m(b,a)` that refer to the same method `m`. When inspecting `m(a,b)` the analysis finds that changing the arguments to `m(b,a)` makes the call site more similar to all other call sites (only one in this example), and the other way around for `m(b,a)`. In general, if the analysis reports a warning for two call sites, where each warning proposes to invert the arguments, then at least one of the warnings is a false positive. We call such warnings *contradicting warnings* and remove them from the list of reported warnings. The revised analysis removes two warnings if there is a naming example N_1 with a warning that proposes P_1 and a naming example N_2 with a warning that proposes P_2, where P_1 gives N_2 and P_2 gives N_1.

This refinement is motivated by false positives reported when the analysis runs with a low threshold for anomalies (threshold t in Algorithm 1). In its default configuration (discussed in Section 7.1.4), the analysis does not report any contradicting warnings during our experiments. That is, filtering contradicting warnings increases the precision when the analysis is configured to report many anomalies, but it does not affect its effectiveness in the default configuration.

5 Naming Bug Detection

Meaningful parameter names for equally typed parameters are crucial for two reasons. First, they help to avoid correctness problems caused by arguments passed in the wrong order. Second, they help a programmer that calls a method to quickly find the order in which to pass arguments, that is, they help to reduce development time. Unfortunately, not all methods with equally typed arguments have meaningful parameter names that allow programmers to assign arguments to their position. How can we find parameter names that fail to help programmers?

In the following, we present an analysis that takes argument naming examples and reports warnings about badly chosen parameter names. The key idea is to leverage the names of arguments passed to a method at different call sites and to find methods where many arguments agree on a particular naming scheme, but where the parameter names do not reflect it.

Algorithm 2 describes our approach to detect badly chosen parameter names. The input are a list of parameter names for parameters of equal type and a set of naming examples from call sites of the method. The algorithm produces a set of positions of parameters that appear to have badly chosen names. This set is either empty, that is, no badly chosen names are reported, or it contains at least two positions. The reason for not reporting a set of size one is that a single badly named parameter is not a problem because a programmer should be able to figure out the right order of arguments from the other names.

At first, the algorithm checks whether there are at least $minCallSites$ naming examples to analyze. If so, then the algorithm performs two main steps. First, it searches for positions $\mathcal{P}_{coherent}$ where the argument names are coherent with each other, that is, where programmers use similar argument names at different call sites. To find coherent arguments, $coherence()$ computes the average pairwise similarity of all arguments passed at a particular position. If the average coherence value is below a threshold $minCoherence$, then the algorithm discards this position. If there are at least two positions with

Algorithm 2 Naming bug detection.

```
Input: Argument naming examples $\mathcal{N}_{arg}$ from call sites
    and a naming example $N_{param}$ from parameters
Output: Positions $\mathcal{P}$ of parameter names that appear to
    be badly chosen
1:  $\mathcal{P} \leftarrow \emptyset$
2:  if $|\mathcal{N}_{arg}| \geq minCallSites$ then
3:      $\mathcal{P}_{coherent} \leftarrow \emptyset$
4:      for all $p \in \{1, \ldots, |N_{param}|\}$ do
5:          if $coherence(\mathcal{N}_{arg}, p) \geq minCoherence$ then
6:              $\mathcal{P}_{coherent} \leftarrow \mathcal{P}_{coherent} \cup \{p\}$
7:          end if
8:      end for
9:      if $|\mathcal{P}_{coherent}| \geq 2$ then
10:         $\mathcal{P}_{dissimil} \leftarrow \emptyset$
11:         for all $p \in \mathcal{P}_{coherent}$ do
12:             if $avgSimil(\mathcal{N}_{arg}, p) \leq maxArgParamSimil$
                then
13:                 $\mathcal{P}_{dissimil} \leftarrow \mathcal{P}_{dissimil} \cup \{p\}$
14:             end if
15:         end for
16:         if $|\mathcal{P}_{dissimil}| \geq 2$ then
17:             $\mathcal{P} \leftarrow \mathcal{P}_{dissimil}$
18:         end if
19:     end if
20: end if
```

coherent arguments, the algorithm continues with the second step. In the second step, the algorithm computes for each coherent position how similar the argument names observed at this position are to the position's parameter name. If the average similarity, computed by $avgSimil()$, is below a threshold $maxArgParamSimil$, then the position is added to the set $\mathcal{P}_{dissimil}$ of positions with parameter names that are dissimilar from the arguments names. Finally, the algorithm checks whether there are at least two positions in $\mathcal{P}_{dissimil}$ and if so, reports the parameters of these positions as being badly named.

The key idea of the algorithm is to find methods where many arguments agree on a particular naming scheme (first step), but where the parameter names do not reflect it (second step). Without the first step, the algorithm finds many methods where the observed argument names differ from the parameter names. For example, this situation often occurs for methods of general-purpose classes, such as `Map.put(Object,Object)`, because the arguments given to such methods typically have domain-specific names. By checking whether the argument names for a position are coherent in the first step, the algorithm avoids reporting false positives caused by methods of general-purpose classes.

6 Implementation

We implement the approach described in Sections 2 to 5 for Java and C. The language-dependent front ends are built upon Eclipse Java Development Tools (JDT) and Eclipse C/C++ Development Tooling (CDT). The Eclipse framework provides us with an AST and statically resolves the methods at call sites. The two front ends are independent of the language-agnostic implementation of the analyses for finding anomalies and naming bugs. Therefore, extending our implementation to other programming languages is straightforward.

We optimize the implementation of Algorithm 1 by leveraging the fact that different call sites often give the same naming examples. For example, consider two call sites `m(x,y)` and `m(x,y)` that both pass local variables with the same names. Instead of running the anomaly detection twice, we run the anomaly detection once for each set of equal examples.

To compute the similarity of strings, we build upon existing implementations of string distance metrics [10]. These metrics are not designed for identifier names in programs and do not take into account the idiosyncrasies of these names. Some of the metrics, in particular the metric that we found to be most successful, rely on a tokenizer that splits a given string into tokens. The out of the box tokenizer does not consider camel case, which is a very common way to concatenate words in identifier names, in particular, in Java programs. We adapt the tokenizer to consider camel case.

Our implementation is available for download at:

http://mp.binaervarianz.de/argument_analyzer

7 Evaluation

The following section reports the results of evaluating our analyses with real-world Java and C programs. We address the following main questions:

- *How effective is the anomaly detection?* The analysis finds 31 anomalies in the Java programs, out of which 26 (84%) are relevant problems. For the C programs, the analysis find 35 anomalies, out of which 28 (80%) are relevant. To measure recall, we automatically seed bugs and our analysis finds 74% of them.
- *How sensitive are the results to parameters of our analysis, such as the threshold for anomalies?* We perform a sensitivity analysis of five parameters and discuss our default configuration.
- *How effective is the naming bug detection?* The analysis finds 31 naming bugs with a precision of 41% for Java programs and 37% for C programs.
- *Do the analyses scale to large programs?* Analyzing 2.2 MSLOC of Java and C code takes about seven minutes. The time required for a program correlates strongly with the number of method call sites in the program.

We use the Java and C programs provided by the DaCapo benchmark suite (version 9.12) [6] and by the SPEC CPU2006 benchmark suite [20]. There are twelve

TABLE 1: Programs used for the evaluation and results from the anomaly detection and the naming bug detection (ETA=equally typed arguments; NETA=named, equally typed arguments; W.=warnings; Corr.=correctness bugs; Nw.=noteworthy anomalies; Nam.=Naming bugs).

	Program	SLOC	Call sites			Anomaly Detection						Naming Bug Detection		
			Total	ETA	NETA	W.	Corr.	Nw.	Nam.	Prec. (%)	Recall (%)	W.	Nam.	Prec. (%)
Java	Avrora	69,393	20,276	3,179	878	0	0	0	0	-	73.4	2	1	50.0
	Batik	186,460	47,655	6,127	2,694	7	0	5	2	100.0	73.4	8	5	62.5
	DayTrader	12,325	4,613	311	103	0	0	0	0	-	80.6	0	0	-
	Eclipse	289,641	280,289	26,097	13,595	12	1	9	1	91.7	74.2	6	0	0
	FOP	102,909	32,806	2,796	1,266	0	0	0	0	-	75.8	4	2	50.0
	H2	120,821	53,221	5,210	1,607	0	0	0	0	-	71.5	4	0	0.0
	Jython	245,016	85,729	15,785	2,480	11	1	3	4	72.7	70.4	2	1	50.0
	Lucene	124,105	41,092	5,667	1,422	1	0	0	0	0.0	60.8	8	6	75.0
	PMD	60,062	21,394	2,601	507	0	0	0	0	-	53.7	1	0	0.0
	Sunflow	21,970	8,139	1,200	537	0	0	0	0	-	69.3	2	1	50.0
	Tomcat	161,131	54,462	4,974	1,482	0	0	0	0	-	81.8	1	0	0.0
	Xalan	172,300	33,828	3,663	1,650	0	0	0	0	-	75.5	1	1	33.3
	All	1,566,133	683,504	77,610	28,221	31	2	17	7	83.9	71.7	41	17	41.5
C	bzip2	5,731	762	110	52	0	0	0	0	-	86.3	1	0	0.0
	gcc	235,884	54,343	14,027	9,988	16	0	10	0	62.5	64.1	20	6	30.0
	gobmk	157,649	10,146	4,551	1,505	17	8	5	4	100.0	80.3	6	3	50.0
	h264ref	36,098	3,815	791	319	0	0	0	0	-	81.0	1	1	100.0
	hmmer	20,658	4,154	758	299	0	0	0	0	-	70.1	0	0	-
	lbm	904	78	9	2	0	0	0	0	-	100.0	0	0	-
	libquantum	2,606	557	176	78	0	0	0	0	-	62.7	2	0	0.0
	mcf	1,574	80	20	11	0	0	0	0	-	84.6	0	0	-
	milc	9,575	1,589	524	271	0	0	0	0	-	77.4	4	2	50.0
	perlbench	126,266	16,791	2,672	1,027	1	0	0	0	0.0	59.2	2	1	50.0
	sjeng	10,544	1,366	374	77	0	0	0	0	-	76.6	0	0	-
	sphinx3	13,128	2,952	1,207	169	1	1	0	0	100.0	78.6	2	1	50.0
	All	620,617	96,633	25,219	13,794	35	9	15	4	80.0	76.7	38	14	36.8

Java programs and twelve C programs.[3] Table 1 lists these programs along with their number of source lines of code (SLOC). In total, the programs sum up to 2.2 million SLOC. Table 1 shows the total number of call sites, the number of call sites with equally typed arguments (ETA), and the number of call sites with named, equally typed arguments (NETA). In total, 102,829 call sites have equally typed arguments. Our analysis can extract names from 42,015 of these call sites.

7.1 Anomaly Detection

We evaluate the anomaly detection in three ways. First, we apply the analysis to the programs from Table 1 as they are shipped with the benchmark suites (Section 7.1.1). This experiment allows us to measure the precision of the analysis and shows what kinds of anomalies the analysis detects in mature and well-tested programs. Second, we automatically seed anomalies into the programs by changing the order of equally typed arguments (Section 7.1.2). This experiment allows us to measure the recall of the analysis (Section 7.1.3). Finally, we measure the sensitivity of the analysis to configuration parameters and discuss our default configuration (Section 7.1.4).

3. There are twelve Java programs even though there are 14 DaCapo benchmarks: *DayTrader* is part of the *tradebeans* and the *tradesoap* benchmarks; *Lucene* is part of the *luindex* and *lusearch* benchmarks. We omit the SPEC CPU *specrand* programs because they have only 49 SLOC each.

7.1.1 Anomalies in Mature Programs

We apply the anomaly detection to the programs listed in Table 1. 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. Table 1 summarizes the results.

For the Java programs, our analysis reports 31 anomalies. We manually inspect these anomalies and classify them as follows:

- Contrary to our expectations, two anomalies are bugs affecting the program's correctness. Figure 1a shows the relevant source code fragments of one bug. 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). The other Java bug found by the analysis is in test code of *Jython*. The programmer calls `assertEquals()`, which takes the actual and the expected result of some computation, and reports a warning if they are not equal. The programmer accidentally passes the expected result first, which will lead to an incorrect output if the test fails.

- 17 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. For example, *Eclipse* has a method `resetTo(int begin, int end)` with a call site that passes arguments called `end` and `length`, which raises the question whether the arguments are ordered correctly. A closer look at the code reveals that the argument order is correct at this particular call site because the variable `end` is the index to start with. 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, or by renaming the argument.
- Seven anomalies can be classified as *naming bugs* [21]. 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 1c 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.
- Finally, five anomalies are false positives. They provide no insight to a developer and, ideally, would not be reported. Most of the false positives are due to names that are similar to each other, such as two arguments called `firstName, name` where the parameter names are `name, outerFullName`. Since the analysis is based on heuristics and programmer-given identifier names, we cannot avoid false positives entirely.

For the C programs, the analysis reports 35 anomalies, which we classify as follows:

- Contrary to our expectations, nine of the anomalies are correctness bugs. Several of them involve a method `gnugo_estimate_score(float *upper, float *lower)` from *gobmk*. For example, one call site for which the analysis reports a warning is `gnugo_estimate_score(&lower_bound, &upper_bound)`. This call site is incorrect and has been fixed in a later version of the analyzed program. Another correctness bug is for method `lm_read_ctl` in *sphinx3*. The method takes nine parameters and three of them have type `float64`. Two of the `float64` parameters are called `wip` ("word insertion penalty") and `uw` ("unigram weight"). The method has exactly one call site, where the arguments for these two parameters are called `uw` and `inspen`, that is, they are clearly passed in the incorrect order. This bug has been fixed in a more recent version of *sphinx3*.
- 15 anomalies are noteworthy. For example, Figure 1b is a noteworthy anomaly from *gcc*, where two local variables `if_false_label` and `if_true_label` are passed as arguments. The arguments are ordered in such a way that `if_false_label` is bound to the formal parameter `if_true_label`, while `if_true_label` is bound to the formal parameter `if_false_label`. Documenting this anomaly would improve the maintainability of the code, because the natural question whether the arguments are ordered correctly does not arise.
- Four anomalies are naming bugs. For example, the `add_attack_move()` method of *gobmk* expects two `int` parameters that both represent positions. Unfortunately, the parameters are called `ww` and `pos`, making it difficult for a programmer that calls this method to distinguish the two kinds of positions.
- Seven warnings turn out to be false positives. Similar to the false positives found in the Java programs, most of them are caused by argument names that are similar to each other.

In summary, 54 of 66 reported anomalies (82%) point to problems that affect the program's correctness, understandability, or maintainability. 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% [37], 37.5% [29], 38% [35], and 70% [21]. For a fair comparison, we use the same procedure to obtain these numbers for each work: at first, accumulate results from all programs analyzed in the respective work, and then, compute the overall true positive rate.

7.1.2 Automated Evaluation Technique

Measuring the recall of the anomaly detection is challenging because the set of all relevant anomalies in real-world programs is unknown. To estimate the recall, we seed 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 site with equally typed arguments and change the order of these arguments. We then assess whether the analysis detects the seeded anomaly. We seed one anomaly after the other 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 recall for a single seeded anomaly is:

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

The overall recall for the program is the mean value over all seeded anomalies. A similar evaluation technique has been used by others [28].

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 the known true positives described in Section 7.1.1. Since we know that these call sites expose relevant anomalies, we ignore them during the automated evaluation. Second, we ignore call sites of 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, call sites with many arguments impose a significant performance problem. However, only around 1% of all call sites with equally typed arguments have five or more arguments, so this restriction does not affect the generality of the evaluation. Third, we apply the automated evaluation only to call sites with named arguments. For other call sites, our technique does not apply and we know without experimenting that the analysis does not report any anomalies. With these refinements, we seed 48,543 anomalies in the Java programs and 24,989 anomalies in the C programs, and run the analysis for each seeded anomaly.

7.1.3 Recall

Table 1 shows the recall of the analysis for each program. For example, the analysis finds 74.2% of all anomalies that we seed in Eclipse. On average, the analysis reveals 72% and 77% of all anomalies for Java and C, respectively. Although automatically seeded anomalies may not be representative for real-world anomalies, these results give us some confidence that the analysis finds most real anomalies. The refinements described in Section 4.3 significantly improve recall compared to our previously presented approach, which has only 38% recall.

There are two main reasons why the analysis misses seeded anomalies. First, there are false negatives for methods that are called in many different contexts, such as `assertEquals()` from the JUnit framework. For such methods, the naming examples from one call site may have no similarity to naming examples from other call sites and therefore, the analysis cannot infer the correct order of arguments. Example families (Section 4.3) address this problem to some degree but cannot solve it entirely. Second, the analysis misses seeded anomalies for methods that implement commutative operations. Ideally, these missed anomalies should not count as false negatives because the analysis should not report any warnings about those methods. Unfortunately, we have not automatic way to exclude all commutative operations from the seeding process.

7.1.4 Parameter Calibration

The presented analysis has several parameters that have a strong influence on the overall results. We evaluate different configurations by measuring recall with the automated evaluation technique from Section 7.1.2 and by estimating precision. To precisely measure precision we would have to inspect for each configuration all warnings that the analysis reports. Instead, we compute a lower bound of the real precision by considering warnings to be false positives unless we know them to be true positives. In the following, we discuss the parameters and analyze the sensitivity of the analysis to each of them. We report results from varying each parameter individually while using default values for the others.

Threshold for Anomalies: The threshold for anomalies determines how deviant from other examples a call site 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 0.9 (large deviance from other examples).

Figures 5a and 5b show precision and recall with different thresholds for Java and C programs, respectively. 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. We choose a threshold of 0.4 as the default configuration, because it provides the best F-measure for both Java and C.

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 call sites of 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.

Figures 5c and 5d show 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 with a single call site because the call site and the formal parameter names give two naming examples.

Parameter Weight: The parameter weight determines how influential parameter names are compared to argument names. If the parameter weight is one, naming examples from parameters are considered as

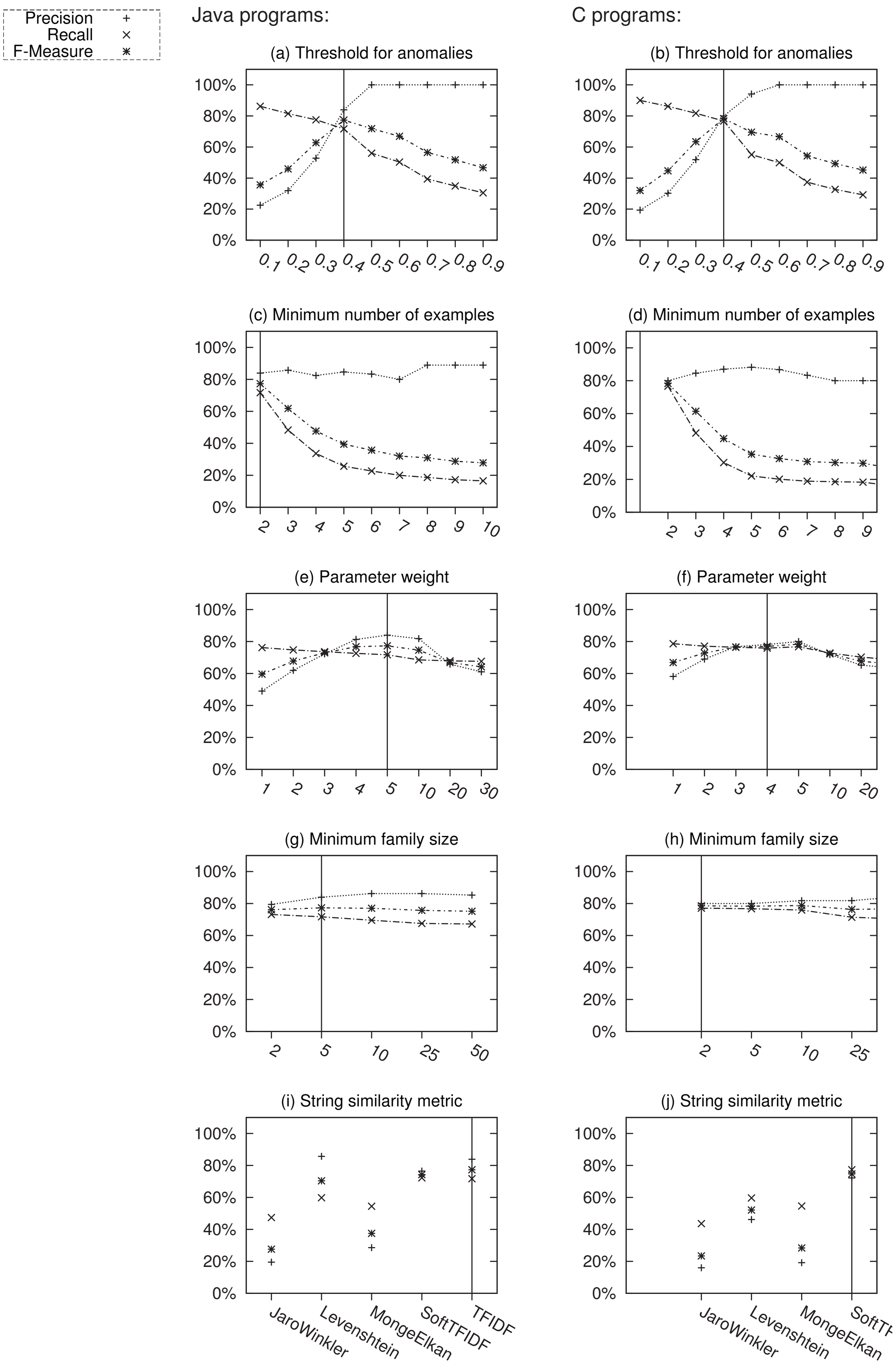

Fig. 5: Parameters of the anomaly detection and their influence on precision and recall. The vertical lines indicate the default configuration we use for the evaluation.

important as all other naming examples. We experiment with values in the range between 1 and 30.

Figures 5e and 5f show how the parameter weight influences precision and recall (note the non-linear x-axis). Giving parameters a higher weight than argument names improves precision, which is not surprising because parameter names should guide callers of method in assigning arguments to positions. A very large parameter weight decreases precision and recall, showing that considering only parameter names (and ignoring argument names altogether) does not work well. Our default configuration is a parameter weight of five, which gives the best F-measure for both Java and C.

Minimum Family Size: The minimum family size determines how many naming examples with similar names we need to analyze them for anomalies. We experiment with values in the range between 2 and 50.

Figures 5g and 5h show the results (again, note the non-linear x-axis). Compared to the other parameters, the minimum family size has a small influence on the overall results. Increasing the parameter slightly increases precision and decreases recall.

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 useful for comparing names [10].

Figures 5i and 5j compare 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.

7.2 Naming Bug Detection

We apply the naming bug detection to the Java and C programs listed in Table 1. The approach is configured with three parameters that determine thresholds for reporting parameters names as badly chosen. Based on initial experiments, we have chosen $minCallSites = 10$, $minCoherence = 0.3$, $maxArgParamSimil = 0.1$. Other configurations allow users to trade precision of warnings for the ability to find more naming bugs.

We inspect warnings reported by the analysis and classify each to be either a naming bug or a false positive. A warning is a naming bug if (i) the order in which arguments are passed to the method matters, and (ii) the names of the parameters are insufficient to order arguments correctly. Many naming bugs are due to cryptic names, such as `s1` and `s2`. However, not all short names are naming bugs. For example, a method taking two `float` parameters that describe a point in a two-dimensional space may be called `x` and `y`, and we do not classify them as naming bugs because these short names are sufficient to map arguments to their positions.

```
void noise2(final double[] noise,
            double vec0, double vec1) { .. }
```

Naming examples for `double` parameters:

Position 1	Position 2
pointX	pointY
pointX	pointY
pointX	pointY
pointX	pointY
pointX	pointY

Fig. 6: Example of a naming bug found in *Batik*.

Table 1 summarizes the results of the naming bug detection. In total, the analysis finds 31 naming bugs with a precision of 39%. The precision for C programs is 41%, whereas the analysis has 37% precision for Java programs. Inspecting a false warning takes very short time and we therefore consider the precision of the default configuration to be acceptable. Inspecting and classifying all naming bug warnings reported in Table 1 took us about one person-hour.

Figure 6 shows a naming bug that the analysis found in *Batik*. The two parameters of type `double` are called `vec0` and `vec1` and represent, according to the method's documentation "the X coordinate to generate noise for" and "the Y coordinate to generate noise for". Unfortunately, the parameter names do not convey this semantics of the parameters. Instead, they use generic names that do not help programmers in figuring out the expected argument order. The naming examples for this method, shown in the table in Figure 6, indicate which names callers of the method would prefer and thereby guide our analysis to reporting this naming bug.

The false positives reported by the analysis have various reasons. For some methods, the parameter names follow a different naming scheme than many argument names. For example, `drawLine()` takes four `float` parameters called `startx`, `starty`, `endx`, and `endy`, but many arguments are called `x1`, `y1`, `x2`, and `y2`. Another reason for false positives are parameter names that describe a class of values, such as `fruit`, combined with argument names that describe instances of this class, such as `apple` and `banana`. Enhancing our approach with sophisticated natural language processing techniques may help to address these two reasons for false positives. Finally, several false positives are due to methods that implement commutative operations. Since any order of arguments is correct for such methods, we reject all warnings for these methods as false positives.

7.2.1 Comparison to Anomaly Detection

Since the anomaly detection evaluated in Section 7.1 also finds naming bugs, one may ask how the sets of naming bugs detected by the two analyses compare to each other. For the Java programs, two of the seven naming bugs detected by the anomaly detection are also found by the naming bug detection. For the C programs, the naming bugs found by the two analyses do not overlap. This result is not surprising because the two analyses use different techniques to find anomalies. We conclude that developers interested in naming bugs should run both analyses.

7.3 Performance and Scalability

On a standard PC (3.16 GHz Intel Core 2 Duo with 2 GB RAM for the Java virtual machine), our prototype implementation requires seven minutes to run both the anomaly detection and the naming bug detection for all programs in Table 1. Most of the time is spent parsing source code; in contrast, the actual analyses, in particular the naming bug detection, terminate quickly. The largest Java program, *Eclipse*, requires 39 seconds. The largest C programs, *gcc*, requires 51 seconds. The running time strongly correlates with the number of call sites in a program (Pearson correlation coefficient: 93% for Java and 90% for C).

8 Related Work

Identifier names are the subject of several studies, which generally agree on the importance of well chosen names. A study involving 100 human participants shows that expressive names are important for program understanding [24]. 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 of FindBugs [22] warnings and other code quality metrics). 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 [21] 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.

Butler et al. [8] propose techniques for tokenizing identifier names in Java programs. These techniques can be combined with string similarity metrics that tokenize strings before comparing them, and therefore, can further improve the effectiveness of our analysis. Similar, integrating a synonym database into our approach, such as in [34], may further improve our analysis.

There are several approaches that address the inability of type systems to discern different usages of variables having the same type. Guo et al. [16] dynamically infer abstract types for variables of primitive types by analyzing how these variables interact, for example, through an assignment. Similarly, Hangal and Lam [18] propose a static analysis to infer dimensions that refine the type information of primitive type variables 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 [18], 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 problems within a single version of a program. One could combine the techniques in [16], [18] with ours by using inferred type refinements for finding problems related to equally typed arguments.

Lawall and Lo [23] propose 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 [15] propose adding type qualifiers to Java to express properties, such as that a variable is read-only. Such approaches require programmers to annotate variables with additional information, and hence, are orthogonal to automatic analyses like ours.

Erwig et al. [13] 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 [13] requires header names to match precisely.

Our work belongs to a class of techniques that search anomalies in a program based on the assumption that most parts of the program are correct. Engler et al. [11] statically search for violations of system-specific rules by inferring "beliefs" of programmers. Hangal and Lam [17] dynamically infer invariants and report violations of these invariants as potential bugs. A static analysis by Lu et al. [27] 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 [25] statically mines rules saying that calling a set of functions within some context implies calling another function. Chang et al. [9] detect missing condition checks by inferring graph-based rules from source code. Thummalapenta and Xie [36] target exception handling rules and how to find their violations automatically. Wasylkowski et al. [37], [38] present analyses to statically detect missing method calls. Similarly, Nguyen et al. [29] and Monperrus et al. [28] learn usage patterns to find code locations where a particular call seems to be missing. 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 may not be available for a particular program. Work on mining specifications follows a similar idea by inferring finite state machines describing method call sequences [5], [14], [26], [30], [31], [33], [39], algebraic specifications [19], or invariants [12] 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.

Zhang et al. propose an IDE-integrated system for recommending method arguments to developers [40]. Their approach is to build a database of usage examples of API methods from existing clients of this API and to use this database to recommend arguments at call sites similar to previous usages. In contrast to their work, our analysis addresses existing programs and specifically addresses problems related to equally typed arguments. A major motivation for their work is that state of the art code recommendation techniques (in the Eclipse JDT) fail to recommend the correct method argument for 53% of all arguments [40]. This result underlines the risk of passing wrong arguments, even in a sophisticated IDE.

9 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, maintainability, and understandability of a program. In this work, we present two automatic program analyses to detect problems related to equally typed arguments. The analyses leverage similarities between programmer-given identifier names. Experiments with a large corpus of Java programs show that the analyses find 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 analyses 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 site. 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.

Acknowledgments

The work presented in this paper was partially supported by the Swiss National Science Foundation under grant number 200021-134453. Thanks to Zoltan Majo, Christine Zeller, Peter Arrenbrecht, and the anonymous reviewers for their insightful comments and suggestions.

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. Exploring the influence of identifier names on code quality: An empirical study. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 156–165. IEEE, 2010.
[8] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Improving the tokenisation of identifier names. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 130–154. Springer, 2011.
[9] 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.
[10] 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.
[11] 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.
[12] 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.
[13] M. Erwig and M. M. Burnett. Adding apples and oranges. In *Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 173–191. Springer, 2002.
[14] 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.
[15] 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.
[16] 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.

[17] 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.

[18] 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.

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

[20] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] M. Pradel and T. R. Gross. Detecting anomalies in the order of equally-typed method arguments. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 232–242. ACM, 2011.

[33] 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.

[34] K. Taneja, D. Dig, and T. Xie. Automated detection of api refactorings in libraries. In *Conference on Automated Software Engineering (ASE)*, pages 377–380. ACM, 2007.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical API usage. In *International Conference on Software Engineering (ICSE)*, pages 826–836. IEEE, 2012.

Michael Pradel is a post-doctoral researcher at ETH Zurich, where he received his Ph.D. in 2012. Michael graduated in computer science at Technical University in Dresden, Germany. He also spent two years at Ecole Centrale Paris, where he became a graduate engineer, and visited EPFL to pursue his master thesis. His research interests are in the area of software engineering and programming languages. In particular, he is interested in automatic program analyses for finding programming errors.

Thomas R. Gross is a Professor of Computer Science at ETH Zurich. He is the head of the Computer Systems Institute and was, from 1999–2004, the deputy director of the NCCR on "Mobile Information and Communication Systems", a research center funded by the Swiss National Science Foundation. Thomas Gross joined Carnegie Mellon University in 1984 after receiving a Ph.D. in Electrical Engineering from Stanford University. In 2000, he became a Full Professor at ETH Zurich. He is interested in tools, techniques, and abstractions for software construction and has worked on many aspects of the design and implementation of software and computer systems. His current work concentrates on low-cost networks (in collaboration with Disney Research, Zurich), compilers, and programming parallel systems.