On the use of statistical machine translation for suggesting variable names for decompiled code: The Pharo case

Juan Pablo Sandoval Alcocer [a],[*], Harold Camacho-Jaimes [c], Geraldine Galindo-Gutierrez [c], Andrés Neyem [a],[b], Alexandre Bergel [d], Stéphane Ducasse [e]

[a] *Department of Computer Science, School of Engineering, Pontificia Universidad Católica de Chile, Chile*
[b] *National Center for Artificial Intelligence CENIA, Santiago 7820436, Chile*
[c] *Exact Sciences and Engineering Research Center (CICEI), Universidad Católica Boliviana, Bolivia*
[d] *RelationalAI, Switzerland*
[e] *Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, France*

ARTICLE INFO

ABSTRACT

Adequately selecting variable names is a difficult activity for practitioners. In 2018, Jaffe et al. proposed the use of statistical machine translation (SMT) to suggest descriptive variable names for decompiled code. A large corpus of decompiled C code was used to train the SMT model. Our paper presents the results of a partial replication of Jaffe's experiment. We apply the same technique and methodology to a dataset made of code written in the Pharo programming language. We selected Pharo since its syntax is simple – it fits on half of a postcard – and because the optimizations performed by the compiler are limited to method scope. Our results indicate that SMT may recover between 8.9% and 69.88% of the variable names depending on the training set. Our replication concludes that: (i) the accuracy depends on the code similarity between the training and testing sets; (ii) the simplicity of the Pharo syntax and the satisfactory decompiled code alignment have a positive impact on predicting variable names; and (iii) a relatively small code corpus is sufficient to train the SMT model, which shows the applicability of the approach to less popular programming languages. Additionally, to assess SMT's potential in improving original variable names, ten Pharo developers reviewed 400 SMT name suggestions, with four reviews per variable. Only 15 suggestions (3.75%) were unanimously viewed as improvements, while 45 (11.25%) were perceived as improvements by at least two reviewers, highlighting SMT's limitations in providing suitable alternatives.

1. Introduction

Variable names play an important role in many activities in software engineering, in particular, source code understanding, one of the main activities in software engineering [1–3]. It has been shown that descriptive variable names reduce the effort of practitioners in reading and comprehending source code [4,5]. For this reason, many software companies introduce some code standards to improve code quality, including variable names [6–9]. However, there are some situations where variable names are not available. For example, when an executable program is provided by a third party without the program's original source code.

Decompilers are tools designed to help practitioners recover source code from an executable or compiled program [10,11]. These tools are used by reverse engineers who need to read and understand the behavior of a program from an executable, for example, to discover software vulnerabilities, analyze malware, or maintain a legacy program where the source code is not available [12–17]. However, for many programming languages, decompilers are unable to recover variable names mainly because these names are lost during the compilation process [18–20].

Statistical Machine Translation Approach. Statistical machine translation (SMT) was originally designed to automatically map sentences from one human language to another. It uses a model trained with a parallel corpus, which is a set of previously translated source–target sentence pairs. In 2018, Jaffe et al. [21] proposed the use of the statistical machine translation (SMT) technique to suggest descriptive variable names for decompiled code, where an SMT model uses the original source code as the training set and its decompiled version as the translation. Jaffe et al.'s study presents an evaluation using a large corpus of decompiled C code. The technique was able to

---

* Corresponding author.
  E-mail address: juanpablo.sandoval@uc.cl (J.P. Sandoval Alcocer).

recover 2.9% to 37.1% of the original variable names, depending on the alignment algorithm and the strategy used to build the parallel corpus. In addition, the accuracy of this approach depends on different factors, including: (i) the syntax and grammar characteristics of the C programming language, (ii) the aggressive optimizations performed by the C compiler, which make the original source code and its decompiled version considerably different, and (iii) the size of the parallel corpora used for training. The study has also proposed some variable renaming strategies that, when applied to the decompiled code version, improve the accuracy of the technique, reaching 37.1% as the upper bound. Most of these renaming strategies make use of type information.

The Pharo Case. Differing from C programming language, Pharo offers a smaller grammar. The language essentially offers message-sending as a unique way to express computation. All computations are carried out by objects sending messages to each other. As such, the grammar of Pharo is extremely simple and consists of a few rules that express this notion of sending messages. The complete syntax of Pharo follows the one of Smalltalk and fits on half of a postcard [22,23]. In addition, the Pharo compiler performs fewer optimizations compared to C. The Pharo bytecode compiler is comparable to the one of Java or Python and, as such, is a representative example of a bytecode compiler [24]. Another important difference is that Pharo is a dynamically typed language, and therefore the code does not provide type information.

Replication Study. To gain a deeper understanding of how specific variations affect the accuracy of Statistical Machine Translation (SMT), we conducted a partial replication study based on the approach originally proposed by Jaffe et al. [21]. In this paper, we present our replication study of Jaffe et al.'s work, in which we applied statistical machine translation techniques to recover variable names from decompiled code written in the Pharo programming language. In their original work, Jaffe et al.'s study [21] focused on evaluating various strategies for constructing a training parallel corpus and aligning suggested variable names with the corresponding source code. In our study, we did not replicate all aspects of their original work. Instead, we narrowed our focus to adapting the strategies that demonstrated the most promising results in building the parallel corpus for the Pharo programming language. Furthermore, we implemented the alignment strategy that yielded the best results in their previous research. In our study, we address two research questions:

- *RQ1- How well does SMT recover original variable names for Pharo decompiled code?* We are interested in replicating the Jaffe et al. study, to evaluate how the particularities of Pharo affect the accuracy of variable name recovery from decompiled code.
- *RQ2- How useful is SMT in improving the original variable names?* This question is about whether the suggested variable names may help developers improve the original names.

Evaluation Setups. We evaluate the accuracy of the SMT approach with 10 Pharo projects using three different setups: (A) *all-but-one*, which represents a real-world scenario, where we train and tune an SMT model using 9 Pharo projects and recover the variable names of a decompiled version of one project not used in the training/tuning phase; (B) *all-as-one*, similar to Jaffe et al.'s study, we combine the code of 10 Pharo projects as one dataset and use 80% for training, 10% for tuning and the decompiled version of the remaining 10% for testing; finally, (C) *one-by-one*, where we train, tune, and test an SMT model with only one project at a time using 80%, 10%, and 10%, respectively. Additionally, in an effort to determine the suitability of suggested variable names compared to the original names, ten Pharo developers reviewed a sample of 400 name suggestions, with each variable name being evaluated by four reviewers.

Findings. Our replication study reveals several facts on the use of SMT to recover variable names from Pharo's decompiled code.

- The variable names suggested by an SMT model trained with different projects than the one under test (all-but-one) provide only between 8.9% and 20.89% of variable names similar to the original code.
- Using a setup similar to Jaffe et al. (all-as-one), an SMT model achieves an accuracy of 40.35%. This result is comparable to the previous work, except for the differences between Pharo and C, in particular, the lack of type information, the syntax, and the structure of the decompiled code.
- By training, tuning, and testing an SMT with only one project (one-by-one), SMT provides between 23. 97% and 69.88% variable names are similar to the original source code, depending on the project. Only one project has an accuracy of 23. 97%, the other nine projects have an accuracy greater than 60%.
- After reviewing a sample of 400 variable name suggestions, we found that 45 suggestions (11. 25%) were considered improvements by at least two reviewers, and only 15 (3.75%) by all four reviewers. However, for 44 (11%) of the variables, there was no clear preference, indicating that while there is room for improvement in the name suggested, the SMT approach did not provide a suitable alternative.

Outline. Section 2 highlights the syntax differences between Pharo and C. Section 3 details how we implement an SMT approach for Pharo. Section 4 describes the methodology we use to answer our two research questions. Section 5 summarizes our results. Section 7 provides an overview of related work. Section 8 concludes and sketches our future work.

2. Pharo overview

As we mentioned in the previous section, SMT was used to suggest variable names for C decompiled code [21]. This article performs a partial replication of Jaffe's work to generate variable names using the statistical machine translation (SMT) approach, but with Pharo as the programming language under study.

Message-send syntax. The Pharo programming language is a pure object-oriented programming language, and compared to C, C++, or JavaScript there are no operators or control structures: all computations are expressed using objects and messages [22]. In particular, the syntax considers three types of messages: unary, binary, and keyword messages. We show some examples in Fig. 1. Since everything in Pharo is an object, traditional operations like `sqrt` or `+` are methods, and primitive values are instances of a class. For instance, 2 and `false` are instances of the class `Integer` and `False` respectively. Keyword messages contain multiple keywords, each ending with a colon, and the arguments are between the keywords. For example, `dict at: index put: value` sends a message `at:put:` to the object receiver `dict`, where `index` and `value` are arguments.

Conditionals and loops. Another particularity of Pharo is that iterations are expressed in terms of sending messages. Consider the following iterator (many of the loops in Pharo are iterators acting as high-order functions) as an example:

```
numbers select: [ :each | each isEven ]
```

This expression is composed of a message send `select:`, and the argument is an instance of the `BlockClosure` class. This expression returns the list of even numbers. In the same way, Pharo conditional structures are also messages. Here, the `ifTrue:ifFalse:` message takes two closures as arguments. For example:
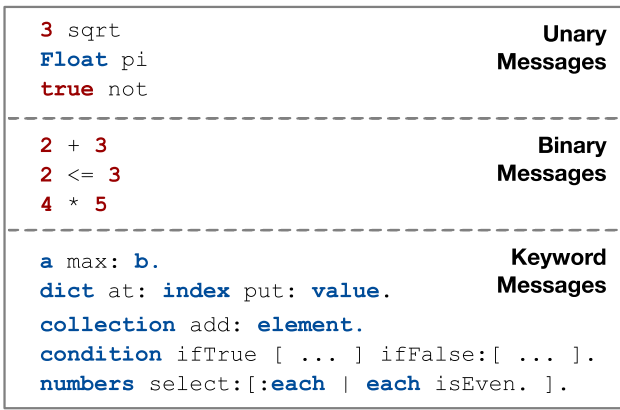
```
condition ifTrue: [ ... ] ifFalse:[ ... ]
```

```
3 sqrt                                    Unary
Float pi                                  Messages
true not
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
2 + 3                                     Binary
2 <= 3                                    Messages
4 * 5
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
a max: b.                                 Keyword
dict at: index put: value.               Messages
collection add: element.
condition ifTrue [ ... ] ifFalse:[ ... ].
numbers select:[:each | each isEven. ].
```

Fig. 1. Pharo message examples.

```
announce: anEvent              Original Source Code
  "Send an event"
  | theEventToSend |
  announcer ifNil: [ ^ self ].
  theEventToSend := anEvent value asAnnouncement.
  theEventToSend canvas: self.
  announcer announce: theEventToSend.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
announce: arg1                      Decompiled Code

  | tmp1 |
  announcer ifNil: [ ^ self ].
  tmp1 := arg1 value asAnnouncement.
  tmp1 canvas: self.
  announcer announce: tmp1.
```
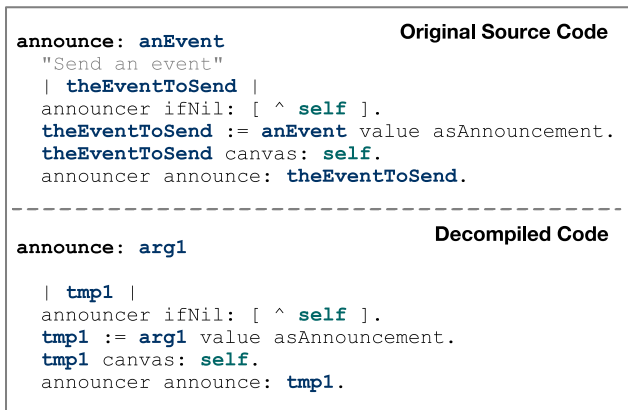
Fig. 2. Example of a decompiled method in Pharo.

Therefore, depending on the object receiver (the condition) only one block closure will be executed.

Decompiled code. Pharo has a live programming environment where each method is compiled individually. The names of the messages sent within the method and the names of the instance variables are preserved in the bytecode generated from the compilation. However, the names of the temporary variables are lost during this process, as is the case with decompilation in most other programming languages. Fig. 2 shows the original source code of a method and its decompiled version.

3. Statistical machine translation approach

Statistical Machine Translation (SMT) is a technique normally used to translate text from one language to another. SMT operates by building a statistical model from a bilingual text corpora. In our case, we apply SMT to translate decompiled code to code with variable names. Similarly to previous work, we use MOSES,[1] an open-source statistical machine translation system [25].

To use MOSES, we first need to train a model with a dataset of already translated texts. This data set is called a parallel corpus. In our case, we train an SMT model using a set of methods using both the original source code and the decompiled code. However, since the SMT techniques were originally designed to translate plain text, we need to
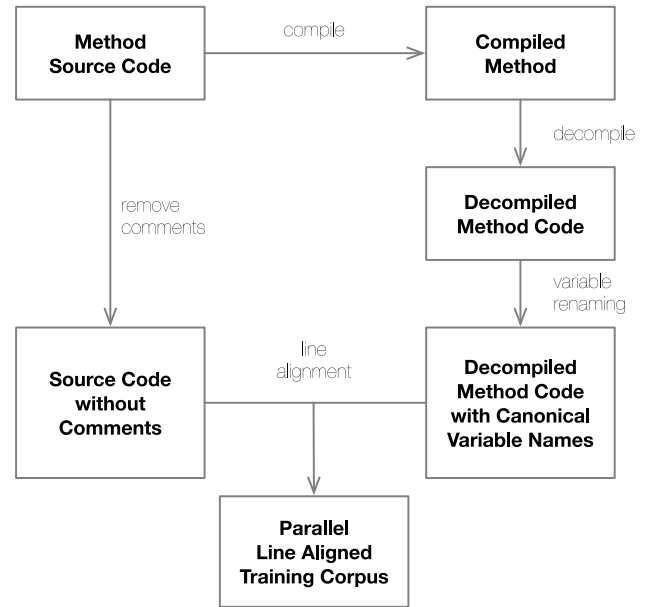
Fig. 3. Training corpus generation.

take a number of intermediate steps to use them in source code to (i) train an SMT model, and (ii) translate a decompiled code. The following subsections describe each of these steps.

3.1. Training an SMT model

MOSES needs a parallel corpus to train a model. Such a corpus needs to be provided line by line. This means that we need to provide a line of source code with its corresponding translation. For each method in the system, we take the original method source code and the decompiled code of the associated binary. Fig. 3 shows the steps that we follow to build a parallel corpus. The remainder of this section details a number of these steps.

Variable renaming. Consider the method in Fig. 2. Pharo decompilation assigns names to arguments and temporary variables using a prefix and an index (e.g. `arg1`, `tmp1`). These names may not benefit MOSES because the name of the variable is bound to the position of the argument [21]. For instance, consider the expression `tmp1 canvas: self`, where the original name of the variable `tmp1` is `theEventToSend`. However, there is no relationship between the word `tmp` with the original variable name and even less to the position in which the temporary variable was declared. In addition, there may be more temporary variables, and the name of this variable could have another number. For this reason, Jaffe et al. [21] proposes a number of strategies to rename the arguments using some context information, such as the type of the variable. However, in the case of Pharo, variable type cannot be used because Pharo is dynamically typed and, as such, Pharo does not have static type information. Therefore, we use a simple canonicalization strategy and rename all the arguments and temporary variables with `v` as a canonical name. Fig. 4 shows the method `announce:` (from Fig. 2) after the variable renaming.

Although there may be alternatives to the renaming of the generic decompiled variables, our results show that our simple canonicalization strategy presents good results.

Line Alignment. Decompiled source code does not necessarily have the same abstract syntax tree as the original source code. This is because the compiler and/or decompiler may perform a number of code transformations and optimizations. For example, consider the method in Fig. 5. The decompiled `addShape:before:` method presents an abstract syntax tree that is different from the original source code. In

```
announce: v                    Variable Renaming
| v |

    announcer ifNil: [ ^ self ].
    v := v value asAnnouncement.
    v canvas: self.
    announcer announce: v.
```

Fig. 4. Method *announce:* after variable renaming.

```
addShape:aShape before:otherShape    Original Source Code

    aShape addedIn: self.
    shapes remove: aShape .
    shapes add: aShape before: otherShape.
─────────────────────────────────────────
addShape:arg1 before:arg2            Decompiled Code

    arg1 addedIn: self.
    shapes remove:arg1;
            add: arg1 before: arg2.
```

Fig. 5. Example of unaligned decompiled method code.

this particular case, the original method code invokes two methods (`remove:` and `add:before`) in two different statements. But the decompiled code performs these two method calls in only one statement. In Pharo, this is known as cascade messages.

This fact makes it difficult to match the last two original lines with their translation (the decompiled code). In an ideal scenario, both the decompiled and the original code should match line by line, and each line must have the same number of keywords with the only difference that the keywords that correspond to variables are different.

After analyzing the ten projects in our dataset (Table 1). We found that about 20% of the decompiled methods have an AST that does not match the AST of the original source code due to the code transformations performed by the compiler. Therefore, their decompiled code is not properly aligned to build the MOSES model, which is trained line by line. To address this situation, we simply take the original method and manually rename the temporary variables and arguments to `v`. That is basically their decompiled version of the method without any code transformations. This is only possible because we have the original code in the training phase.

3.2. Translating decompiled code

Once we train and tune MOSES with the parallel aligned corpus, we evaluate how well MOSES translates the decompiled code to a code with variable names. Fig. 6 shows an overview of the steps our approach takes to obtain the translation from the decompiled code.

The input of the approach is a decompiled code, and the first step is to rename all the arguments and temporary variable names to a canonical name selected in the previous section `v`, because the MOSES model was trained using these variable names.

After the variable renaming, the decompiled code is sent to MOSES line by line. MOSES processes each line and returns a new line with a number of changes (the translation). Ideally, after processing each line, MOSES should provide the same name for each variable across the lines. However, this may not be the case. For example, consider Fig. 7, which shows a decompiled code example and its translation.

Note that the variable `shapes` conserves its name because in Pharo, instance variable names are preserved during compilation. `arg2` has been renamed to `aShape`. However, in this translated code example, we have three different variable names (`shape, aShape, aBox`) that
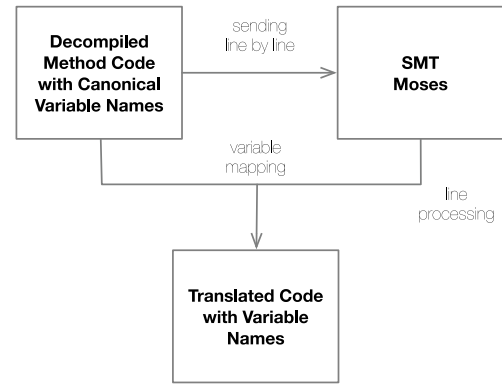

Fig. 6. Translating decompiled code.

```
arg1 addedIn: self.                  Decompiled Version
shapes remove: arg1.
shapes add: arg1 before: arg2.
─────────────────────────────────────────
v addedIn: self.                     Canonical Version
shapes remove: v.
shapes add: v before: v.
─────────────────────────────────────────
shape addedIn: self.                 MOSES Translation
shapes remove: aShape.
shapes add: aBox before: aShape.
```

Fig. 7. Example of a decompiled code, its canonical version, and its MOSES translation.

correspond to the variable `arg1`. This happens because MOSES translates each line independently and is not aware that there is a relation between the lines. To address this issue, similar to previous work, once we translate all the lines, we collect all the possible names for a given variable. This process is done by mapping the translated code with the decompiled code and contrasting if MOSES assigns different names to the same variable between lines. In case a variable has more than one suggestion, we create a code version of the method with each of the name suggestions. Then we use MOSES to evaluate how good the name is according to its trained model. For this, we send each code version to MOSES and it returns a value that represents the match probability. Once we process all versions, we select the name suggestion with the highest probability.

Another assumption is that for each line of decompiled code, MOSES returns a line normally with the same number of keywords in it, and they map perfectly with the decompiled code line except for the variable names that were replaced by the suggested names. However, there are cases where MOSES adds an additional keyword to the output line that does not necessarily match a variable name. This may happen, for instance, when the translation of a word is two words or more. When this happens, it is not trivial to map the words in the output with the variable names. Although there are few cases of this situation, we treat this issue as an instance of the sequence alignment problem, which aims at finding the similarity of two ordered sequences. The sequence alignment problem is commonly used to align multiple RNA or DNA sequences that may have gaps or extra subsequences and to score the alignment between both. But in our case, we may have gaps or additional keywords in the translated code. To address this, we implement the Needleman–Wunsch algorithm, which has shown the best results in previous work [21,26]. It finds the optimal alignment more quantitatively by giving scores for matches and mismatches between elements in both sequences. It uses dynamic programming to find the alignment with the highest score.

Table 1
Projects under study (# Variables = Instance variables, arguments, and temporary variables).

Project	#Methods	#Variables	Size (Kb)
Calypso	4,394	10,453	561.2
Fuel	846	3,629	140.9
Glamour	4,201	15,675	655.7
GT	1,773	5,544	254.1
Iceberg	3,859	6,883	367.3
Kernel	6,354	25,115	1080.9
Metacello	3,589	16,230	758.7
Morphic	8,795	31,168	1293.7
Seaside	6,039	18,441	1061.8
Spec2	6,245	13,967	640.5
Total	**46,095**	**147,006**	**6814.8**

4. Methodology

This section details the methodology we designed to address the two research questions formulated in Section 1.

4.1. Projects under study

We used a dataset composed of 10 Pharo projects, each containing more than 150K variables. These projects are frequently used by the Pharo community. Table 1 gives the names of these projects, their number of methods, their number of variables, and their weight in kilobytes. In contrast to the study done in C which uses 402,925 software projects [21], our dataset is quite small. However, this allows us to evaluate how effective the technique is with a smaller dataset.

In the case of Pharo, the compilation process only loses the names of the arguments and temporary variables. In this sense, instance variable names remain in the compiled code and are therefore recovered in the decompiled version. Since the previous work was done in C, it mostly considers local and global variables. This is another difference in the dataset since Pharo is an object-oriented programming language that also has instance variables.

4.2. Training, tuning, and testing

The previous study [21] randomly takes different projects from GitHub and divides the code compiled into two sets, one for training and the other for testing. Each compiled code was randomly assigned to the training, testing, tuning, or validation set with a probability of 94%, 3%, 0.5%, and 2.5%, respectively. They reserve a validation set to manually test their heuristics and alignment algorithms.

Since our data set is small compared to the previous study, we evaluate the SMT approach under three setups:

- *Setup A: all but one* – In this setup, we train 10 SMT models. Each model was trained and tuned using 9 projects from our basket of 10 projects and tested with the remaining one. For instance, to evaluate the approach in the `Calypso` project, we train an SMT model with all projects, except `Calypso` itself. This configuration represents a real usage scenario where we have the source code of different projects, and we want to recover the variable names of a decompiled code for which we do not have the original source code.
- *Setup B: all as one* – We prepare a global SMT model using the code from all projects as a global dataset. This setup is similar to the way SMT was evaluated in previous work, in which a sample of code files from different projects was taken and randomly selected the files from which the model is trained, tuned, and tested. We train our SMT model using 80% for training and 10% for tuning. Then, we evaluate the global model with the remaining 10%.

- *Setup C: one by one* – In this setup, we prepared an individual SMT model using the code of each project, respectively. This configuration mainly analyzes how SMT can help us improve the variable names of a new piece of code with a model trained with the code from the same project. We train and tune a model with 80% and 10% of each project. Then, we evaluate the model with the remaining 10% of the same project.

4.3. Variable name similarity

To answer our first research question, we use the SMT approach to recover a number of variable names from a corpus of Pharo decompiled code. For each method under evaluation, we send to MOSES their decompiled code versions, then we statically analyze the output to collect the name suggestions. If one variable has more than one suggestion, we select the suggestion that has a higher probability according to the trained model. Finally, we compare the suggested variable names with the original source code and count how many of the suggested names are similar to the original variable name.

We consider the following criteria to determine whether the two names are similar or not.

- *Equal* – when the suggested name is identical to the original name.
- *Prefix* – when a suggested name is a prefix of the original name and at least half as long. For example, `projectReference` - `projectReferenceSpec`.
- *Prefix plus Number* – when the only difference between the suggested name and the original name is a number at the end of one of them. For example, `aCommand1` - `aCommand2`.
- *Contains* – when the suggested name is contained in the original name, or vice versa. And the contained name is at least half as long. For instance, `selector` - `aSelector`, `integer` - `anInteger`.

The first three criteria are the same as those used in the previous work. We add the last criterion (Contains Half) to consider special variable name cases particular to the Pharo programming language (i.e. `item` and `anItem`). It is idiomatic in Pharo to add `a` and `an` to an argument. In case two variable names match one of these criteria, we consider them similar. In addition, we manually reviewed all suggested names and original names to verify that we did not miss any potential matches. This step was carried out by two of the authors of this paper. For instance, `aLGitCommit` and `aGitCommit` do not meet any of the previous rules and are therefore not considered similar, despite their high similarity to the original name. We use these similarity criteria to evaluate the output of the three set-ups to answer our research questions.

4.4. Assessing original variable names

To address our second research question, we manually analyze a sample of our dataset due to the large number of variables it contains. We evaluate both the original variable name and the one recommended by the SMT to determine whether the suggestion enhances the readability and comprehensibility of the source code. This experiment is based on the suggestions of variables obtained through Setup C, which involves training and fine-tuning a model using the same project. This configuration replicates a scenario in which developers might train a model using their existing code project and leverage the model to propose variable names for their future code.

The process begins with the random selection of a sample of variables from the dataset. The data set used to test the model comprises 4756 variables. After our initial analysis, we found that the model suggests a total of 2848 variable names that are similar to the original. Consequently, there are 1908 different name suggestions. Therefore, we focus on analyzing the suggestions that are different from the original. Since we did not have prior information on the distribution, we used

random sampling without replacement to extract a representative sample from the population of variables [27]. We determine the sample size (n) using the following formula:

$$n = \frac{N \cdot \hat{p}\hat{q}(Z_{\alpha/2})^2}{(N-1)E^2 + \hat{p}\hat{q}(Z_{\alpha/2})^2}$$

Since we had no prior information about the distribution, we used a value of 0.50 for $\hat{p}$ and $\hat{q}$, as recommended in the literature [27,28]. We use a standard confidence level of 95% and an error margin of 5%. The size of the population is $N = 1908$, as we aim to analyze only the variables for which SMT provides a different name suggestion. Using the formula above, we obtained a minimum sample size of $n = 320$. We decided to round the sample size up to 400 for practicality in the review process.

The suggested variable names were evaluated by two groups: G_1, which includes five of the authors, and G_2, made up of five independent Pharo developers. We separated the participants into two groups to differentiate between those who were part of the tool's development, who might have a bias toward or against a variable due to their familiarity with the approach, and Pharo developers who are unaware of the approach's workings. Each group independently reviewed a sample of 400 variables. Each variable underwent a separate examination by two participants of the same group (called "reviewers" hereafter). Consequently, each variable was examined by four reviewers in total, two from G_1 and two from G_2.

For each variable, the source code containing the variable, with the name of the variable under analysis obfuscated, was presented along with two name options for the variable. This gave context to the participant to decide on the potential name for the analyzed variable. Note that only one variable was obfuscated at a time, as our goal was to address research question 2. This question aims to assist developers in improving the original variable names. The reviewers were required to select one of the two variable names without knowing which one was the original name and which was the suggestion, to avoid any bias towards either choice. They also had the option to choose "None of Them" in case the reviewer disagreed with both names.

While previous research papers have consistently emphasized the importance of "meaningful identifiers", there is not a clear definition for what constitutes a "meaningful" variable name [29–33]. From the standpoint of software developers, the main goal is to guarantee that the source code remains comprehensible. To evaluate the variable names, the reviewers initially examined the correlation between the variable name and the value that it holds. Subsequently, they analyze whether the variable names under consideration, both the original and the suggested ones, contribute to the readability and comprehensibility of the source code.

After the reviewers completed their analysis of the variables, we compiled their agreements and disagreements. In cases of disagreement, the reviewers must collaborate to reach a consensus and arrive at a final decision between the selected options. Ultimately, we record the following statistics: the number of times the original name was preferred, the number of times the SMT suggestion was favored, and the number of instances where neither of the names was selected.

5. Results

This section summarizes our results and provides the answer to our research questions.

5.1. Recovering variable names (RQ1)

To answer the first research questions we train, tune, and test an SMT model under the three previously mentioned setups.

Setup A: All but one. We recover the variable names for each project using an SMT trained and tuned with all the projects but the one under test. Table 2 shows the results for each project. Overall SMT identifies 12.63% of variables similar to the original ones. Depending on the project, the results range from 8.9% to 20.89%. In particular, the `Iceberg` project had the best results, and `Metacello` had the worst results. Note that the setup *all but one* is different from the one used in the previous study, but represents a real-life scenario.

> **Finding 1.** The suggested variable names by an SMT trained with different projects than the one under analysis only provide between 8.9% and 20.89% variable names similar to the original code.

Setup B: All as one. We tested, trained, and tuned the model by taking the ten projects under study as one data set. Similarly to the Jaffe et al. work [21], we take the source code of all projects and divide them into three subsets: training, tuning, and testing. But, in this case, we use 80% for training, 10% for tuning, and 10% for testing. Table 3 details how many variable names recovered by the SMT are similar to the original ones. In general, the trained model was able to recover 40.05% of names that are similar to the original. Note that in contrast to setup A, the training set was selected from a basket that contains the source code of all projects, and therefore the SMT was trained with code portions of all projects. This shows that precision increases if the training corpus shares similarities with the code under tests, for instance, the similar keywords used. For this reason, setup B provides better results than setup A, where the training set does not have any relation to the code under test.

This fact is also shown in Jaffe et al.'s study [21], where they explore ways to improve precision, for instance, by adding an older code version of the project under analysis to the training set. This additional information increases the precision of their approach between 26.5% and 37.1%. Previous work also improves the accuracy of the approach by providing variable renaming strategies that are applied to the decompiled code before training and testing. Most of these strategies use the variable's type. In the case of Pharo, we obtain better results besides we use a simple canonical renaming strategy due to the lack of type information.

Other factors that could help to achieve higher accuracy are the simplicity of the Pharo syntax, the satisfactory alignment of the decompiled code, and a relatively small code corpus. As we mentioned earlier, only 20% of the decompiled code has an AST different from the original code. This is partly because the decompilation is done from bytecode and not from the assembly, and the aggressive optimizations and code transformations are performed in another layer in the Virtual Machine.

> **Finding 2.** Using a setup similar to Jaffe et al. (all-as-one), an SMT model achieves an accuracy of 40.05%. This result is comparable to the previous work, apart from the differences between Pharo and C, in particular the lack of type information, the syntax, and the structure of the decompiled code.

Setup C: one but one. SMT model has better results using setup C, ranging from 23.97 to 69.88 depending on the project. Remember that this setup uses a particular model for each project training with part of its source code. Therefore, there is a greater similarity between the training and testing set, since the code most likely uses the same variable names and method names since the testing set belongs to the same project. This fact is known as overfitting [34].

Table 4 presents the results for Setup C, where Glamour, a framework designed for creating browsers and defining navigation flows between components, scored the lowest in the dataset. Glamour has a declarative syntax and a distinctive naming convention for its variables. For example, let us look at the Glamour code snippet in Fig. 8, which is used to create a browser. In Glamour, is common to chain multiple messages on the same object. It is also common to pass method names as arguments and utilize reflection mechanisms for browser construction.

Table 2

Results using the setup A - all but one.

Project	#var	Similar variable names (RQ1)					
		Equals	Prefix	Prefix + num	Contains	Total	%
Calypso	3,370	419	5	30	20	474	14.07%
Fuel	923	85	8	4	10	107	11.59%
Glamour	4,453	549	11	73	37	670	15.05%
GT	1,621	225	1	18	20	264	16.29%
Iceberg	2,446	468	10	21	12	511	20.89%
Kernel	7,448	466	29	71	126	692	9.29%
Metacello	4,527	367	5	19	12	403	8.90%
Morphic	10,351	980	35	214	49	1278	12.35%
Seaside	5,112	474	10	62	24	570	11.15%
Spec2	4,844	614	22	50	41	727	15.01%
Total	**45,095**	**4647**	**136**	**562**	**351**	**5696**	**12.63%**

Table 3

Result using the setup B - all as one.

Project	#vars	Similar variable names (RQ1)						
		Equals	Prefix	Prefix + num	Contains	Manual	Total	% similar
Calypso	342	144	2	12	3	17	178	52.05%
Fuel	112	43	3	1	5	4	56	50.00%
Glamour	438	189	0	15	11	16	231	52.74%
GT	196	64	1	14	2	5	86	43.88%
Iceberg	268	92	3	12	0	17	124	46.27%
Kernel	811	236	5	12	11	49	313	38.59%
Metacello	420	102	1	10	6	8	127	30.24%
Morphic	1105	315	3	21	14	32	385	34.84%
Seaside	535	162	2	18	7	10	199	37.20%
Spec2	529	165	0	19	11	11	206	38.94%
Total	**4756**	**1512**	**20**	**134**	**70**	**169**	**1905**	**40.05%**

```
browser := GLMTabulator new.
browser column: #one.
browser transmit
    to: #one;
    andShow: [:a |  a accordionArrangement.
        a title: 'title'.
        a list.
        a text ].
window := browser openOn: 42.
```

Fig. 8. Glamour syntax convention example.

Moreover, variable names within block closures are typically denoted by the letter "a". This particular convention is embraced in Glamour to improve code readability during browser construction. These distinct characteristics contribute to the lower score of the project. This suggests a future extension of this work by considering domain-specific variable name heuristics.

> **Finding 3.** By training, tuning, and testing an SMT only using one project, the SMT provides between 23.97% and 69.88% of variable names similar to the original source code, depending on the project. Only one project has an accuracy of 23.97%, and the other 7 projects have an accuracy greater than 60%.

5.2. Assessing original variable names (RQ2)

To address the second research question, we conducted a manual review of 400 variables, analyzing both the original variable names and the suggestions provided by SMT. For this process, two distinct groups were formed, each independently assessing the same set of 400 variables. Within each group, two reviewers analyzed each variable, ensuring a thorough evaluation. The distribution of preferences for each pair of reviewers is presented in Table 5. The Kappa interagreement scores are 0.264 and 0.206, indicating a fair and slight level of agreement, respectively.

As shown in Table 5, both reviewers in G_1 favored the original variable name 213 times, while those in G_2 preferred the original variable names 203 times. The suggested variable names were chosen 16 and 19 times, respectively, in each group. Surprisingly, for 29 and 22 of the 400 variables, neither group of reviewers found the original name or the suggestion suitable as a variable name. This discovery indicates that within the source code, there are original variable names that could benefit from further refinement.

After conducting the first round of analysis, the reviewers in both groups thoroughly examined the disagreements and arrived at a final decision. Group G_1 favored the suggested name for 25 of the 400 variables (6.25%). In contrast, group G_2 favored the suggested names for 35 of the 400 variables (10.25%). G_2 favored 10 variables more than G_1, and both groups agreed on 15 variables, resulting in a total of 45 suggested names that were favored by at least one group. This observation suggests that the reviewers found these suggestions more appropriate than the original names in terms of readability and comprehensibility. Table 6 shows the 45 instances where suggestions were preferred during manual analysis.

We made a manual comparison of the original and suggested names for each of these 45 cases. It should be noted that in 16 out of the 45 variables, the original name was related to the generic type while the suggestion gave information about the content of the variable (e.g., `anObject` and `aVersion`). In 6 cases, the suggested name was an extended version of the original name, which was a one-letter abbreviation (e.g., `e` and `entity`). A similar case occurred in 4 cases in which the variable used as an iterator had a generic name and was suggested one more related to the content of the variable (e.g., `each` and `action`). Finally, in 16 cases, the suggested name was considered to be more appropriate to the context than the original name (e.g., `globalKey` and `globalName`)

Overall, only 3.75% of the cases (15 out of 400) reviewers in both groups agree that the suggested variable names contribute more to the readability than the original name. On the other hand, there were 11%

Table 4
Result using the setup C - one as one.

Project	#vars	Similar variable names (RQ1)						
		Equals	Prefix	Prefix + num	Contains	Manual	Total	% similar
Calypso	342	200	2	14	5	18	239	69.88%
Fuel	112	62	1	1	3	5	72	64.29%
Glamour	438	73	0	8	11	13	105	23.97%
GT	196	98	0	12	7	5	122	62.24%
Iceberg	268	120	0	12	3	16	151	56.34%
Kernel	811	374	6	17	17	46	460	56.72%
Metacello	420	244	2	11	10	26	293	69.76%
Morphic	1105	592	9	31	25	67	724	65.52%
Seaside	535	281	1	23	4	13	322	60.19%
Spec2	529	301	4	25	15	15	360	68.05%
Total	**4756**	**2345**	**25**	**154**	**100**	**224**	**2848**	**59.88%**

Table 5
Inter-agreement results. The rows show the categorization by the first reviewer, while the columns display the categorization by the second reviewer.

Group G_1	Original	Suggested	None	Group G_2	Original	Suggested	None
Original	213	23	21	Original	203	26	36
Suggested	22	16	5	Suggested	31	19	9
None	57	14	29	None	41	13	22

variables in which the reviewers in both groups disagreed with either the original or the suggested variable name. This shows that there is still an important number of variables that could be beneficial for a variable renaming.

> **Finding 4.** After reviewing a sample of 400 variable name suggestions, only 3.75% of the suggested variable names were deemed by all reviewers as potential improvements to the original variable names. And 11.25% of the suggested names were considered as an improvement by at least two reviewers. Additionally, for 11% of the variables, neither the original name nor the suggestion was chosen, indicating that there are instances where the name of the variables could be improved, but the SMT approach did not provide a suitable alternative.

6. Discussion

This section discusses a number of aspects of our approach that are different from the previous study.

Amount of Training Data. Our study uses a small data set compared to the previous work. However, in addition to this, we present a greater percentage of similar variables than the original source code using the same setup (setup B). Additionally, we also evaluated training a model with the code for each project individually, which means that the training set was even smaller. This last evaluation slightly increases the precision. One reason may be that, although Pharo developers follow a common programming style, each project may have some particularities, for instance, in the variable names. We also test SMT in a real-world scenario where we train a model using different projects for which we have the original source code and then we use the model to recover the variable names for a project for which we only have the compiled code. Our experiment shows that SMT recovers only 12.63% of the variable names.

Alignment. Previous work has shown that using an alignment strategy to generate a parallel corpus produces an SMT model that can recover more variable names than no alignment at all. One difference between our approach and previous work is that the Pharo compiler performs a limited number of code transformations and optimizations compared to C. In addition, the decompilation in Pharo is done from the generated bytecodes and not from the assembly: the most aggressive optimizations are performed in another layer within the Just-in-Time translator (JIT) of the Pharo Virtual Machine. All this makes the Pharo

Table 6
Potential improvement name suggestions for the original variable name.

#	Original	Suggested	G_1	G_2
1	globalName	globalKey	✓	✓
2	m	aMorph	✓	✓
3	anObject	anIntegerOrString	✓	✓
4	anObject	aVersion	✓	✓
5	actual	html	✓	✓
6	s	stream	✓	✓
7	i	index	✓	✓
8	anObject	aPosition	✓	✓
9	new	each	✓	✓
10	each	method	✓	✓
11	result	materializedInstance	✓	✓
12	anObject	aColor	✓	✓
13	e	each	✓	✓
14	sender	each	✓	✓
15	aNumber	key	✓	✓
16	event	item		✓
17	byteShift	start		✓
18	each	action		✓
19	each	item		✓
20	pair	assoc		✓
21	result	allItems		✓
22	anObject	aCollection		✓
23	anInteger	widthTop		✓
24	aString	labelId		✓
25	aString	varName		✓
26	aSelector	html		✓
27	lines	result		✓
28	tenMissing	absent		✓
29	fs	each		✓
30	mustFit	args		✓
31	materialized	result		✓
32	mock	presenter		✓
33	importSpec	prj		✓
34	operation	conflict		✓
35	myItem	existing		✓
36	anObject	aMorph	✓	
37	aModel	morph	✓	
38	mock	presenter	✓	
39	anObject	id	✓	
40	aString	message	✓	
41	allItems	stream	✓	
42	anObject	composite	✓	
43	anObject	transformation	✓	
44	each	aSelector	✓	
45	e	entity	✓	

decompiler able to produce decompiled code that is close to the original one, facilitating alignment and variable matching. We do believe that our SMT model presents better precision due to this fact, therefore, we argue that the alignment is an important factor.

Canonicalization. Pharo decompiler assigns generic variable names to arguments and temporary variables (e.g. `tmp1` or `arg1`). Previous work argues that these names are merely placeholders and do not convey any meaningful information by themselves. Therefore, they propose a number of strategies to rename the names of the decompiled

variables to improve the performance of SMT. The one who gave better results in their original experiment used the type of variable and the position of the argument. In our case, Pharo does not have type information within the source code. Therefore, we use a simple canonicalization from which we were able to recover between 8.9% and 69.88% variable names (depending on the setup). However, we believe that better renaming strategies may help improve the precision of the approach.

Keyword-based Message Syntax. One main difference between the C and Pharo syntaxes is the way methods are invoked. For instance, consider the Pharo message `canvas addShape: box before: shape`. First, since Pharo is an object-oriented programming language that has an object receiver at the beginning, the two arguments are separated by keywords and do not use a comma, as in C. This brings the Pharo syntax closer to a natural language. Therefore, we believe that this is another factor that helps achieve greater precision in the Pharo syntax.

Manual Categorization. One challenge we face in our manual categorization process is the inherent subjectivity involved in assessing the names of variables. To reduce this bias, the manual analysis was performed by ten Pharo developers, split into two groups: Group G_1, consisting of five authors, and Group G_2, made up of five independent Pharo developers. This division was aimed at identifying potential biases, especially since the authors could have biased opinions on the suggested names due to their comprehensive understanding of the approach.

Each variable was evaluated by four reviewers, two from each group. All reviewers were not informed of which names were originally assigned and which were suggested. The results, as shown in Tables 5 and 6, showed only minor differences in the preferences of the two groups after the initial review round. For example, G_1 preferred the original names 213 times and 203 times for G_2. After the disagreement session, both groups agreed that 15 name suggestions were more descriptive than the original variable names. It should be noted that 45 name suggestions were favored by at least two reviewers in one group. This small difference may be attributed to reviewers' variable name preferences. However, our results suggest that only a limited number of variables could benefit from SMT name suggestions.

Code Style and Conventions. Across various setups, we observed that some projects scored notably lower compared to the rest of the dataset. For example, this is the case of the Glamour project in setup C, on which we train and tune a model with 80% and 10% of the Glamour code, and test with the remaining 10%. We partially attributed the low score to the fact that Glamour, along with its methods, exhibits a distinct coding style and conventions For instance, part of the methods use builders and fluent interfaces as is shown in Fig. 8. This highlights how even projects in the same language may influence the accuracy of the approach.

7. Related work

Decompilation plays an important role in the security and maintenance of software. Due to its relevance, various techniques have been proposed to obtain decompiled code from binaries [35–38]. However, most existing decompilation tools have low accuracy in identifying variables, functions, or composite structures. Therefore, decompiled code often has poor readability. For this reason, various attempts have been made to improve the readability of the decompiled code [13,39–43].

In particular, this paper presents a replication study of Jaffe et al. which uses SMT to recover the variable names of C decompiled [21]. Unlike that work, we apply SMT to Pharo's decompiled code. Pharo has a different syntax than C-like languages and does not have type information within the source code. Our motivation is to test the applicability and generality of this approach. Another key difference is that Pharo Smalltalk syntax is that the compiler performs quite a few code optimizations and transformations; therefore, the structure of the decompiled code is most of the time the same as the original code [24].

This fact is useful to compare the suggested variable name with the original name.

Vasilescu et al. presents Autonym, a SMT-based approach to recover the variable names of the obfuscated JavaScript variable names [44]. Autonym has a median precision of 30% but ranges between 10% and 60%. However, the median and distribution reported in this study are performed on a file basis. This fact makes a fair comparison difficult to establish. Jaffe et al. [21] and our study report a precision measure using all projects at the same time. Vasilescu et al. also introduces JSNaughty which combines Autonym and JSNice [44]. JSNice [45] aims to recover minimized variable names using sophisticated static analysis. We do not contrast our results with JSNaughty, since we focus on analyzing the effects of the programming language on the precision with a fully SMT-based approach.

In addition to these works, our study analyzes whether it is more convenient to have a specific model for each project than to have a global model. In addition, we evaluated how well SMT may suggest better variable names than the code from which it was trained.

8. Conclusion

This study extends the work of Jaffe et al. [21], by evaluating the statistical machine translation (SMT) approach to suggest variable names in decompiled code, using projects written in the Pharo Programming Language. Our investigation involved diverse experimental setups across ten Pharo projects, demonstrating that the model's ability to accurately suggest variable names significantly depends on the training data.

Using a setup similar to that of Jaffe et al. we used all the code from the ten projects as a data set, where 80% was used for training, 10% for tuning, and 10% for testing, the SMT model achieves an accuracy of 40.05%. This result is comparable to previous work, apart from the differences between Pharo and C, in particular, the lack of type information, syntax, and the structure of the decompiled code. These findings underline the adaptability of the SMT approach to the highly reflective nature of Pharo. Our findings also show that when the model is trained using the source code from nine projects and then tested on the remaining project, it achieves an accuracy between 8.9% and 20.89%. This highlights that the degree of code similarity between the training and testing data sets plays an important role in model performance.

Our analysis further explored the utility of the SMT approach in assisting developers to refine original variable names. After analyzing 400 name suggestions with four reviewers per variable, we found that only 3.75% of the cases received unanimous approval that the suggested names were preferred to the original. Furthermore, 11.25% of the suggestions were considered preferred by at least two reviewers. However, for 44 variables (11%), the developers did not show preference between the original and suggested names. Future research will investigate various techniques for variable name suggestion, to provide developers with more sophisticated tools for code enhancement.

CRediT authorship contribution statement

Juan Pablo Sandoval Alcocer: Writing – original draft, Software, Methodology, Funding acquisition, Conceptualization. **Harold Camacho-Jaimes:** Software, Investigation. **Geraldine Galindo-Gutierrez:** Writing – review & editing, Formal analysis. **Andrés Neyem:** Writing – review & editing. **Alexandre Bergel:** Writing – review & editing, Methodology, Conceptualization. **Stéphane Ducasse:** Writing – review & editing, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Acknowledgments

References

[1] D. Lawrie, H. Feild, D. Binkley, Syntactic identifier conciseness and consistency, in: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006, pp. 139–148, http://dx.doi.org/10.1109/SCAM.2006.31.

[2] D. Lawrie, C. Morrell, H. Feild, D. Binkley, Effective identifier names for comprehension and memory, ISSE 3 (2007) 303–318, http://dx.doi.org/10.1007/s11334-007-0031-2.

[3] A. Schankin, A. Berger, D.V. Holt, J.C. Hofmeister, T. Riedel, M. Beigl, Descriptive compound identifier names improve source code comprehension, in: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 31–40, http://dx.doi.org/10.1145/3196321.3196332.

[4] E. Avidan, D.G. Feitelson, Effects of variable names on comprehension an empirical study, in: Proceedings of the 25th International Conference on Program Comprehension, ICPC '17, IEEE Press, 2017, p. 55?65, http://dx.doi.org/10.1109/ICPC.2017.27.

[5] D. Lawrie, C. Morrell, H. Feild, D. Binkley, What's in a name? A study of identifiers, in: 14th IEEE International Conference on Program Comprehension, ICPC'06, 2006, pp. 3–12, http://dx.doi.org/10.1109/ICPC.2006.51.

[6] S. Butler, M. Wermelinger, Y. Yu, Investigating naming convention adherence in Java references, in: 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME, 2015, pp. 41–50, http://dx.doi.org/10.1109/ICSM.2015.7332450.

[7] F. Deissenbock, M. Pizka, Concise and consistent naming [software system identifier naming], in: 13th International Workshop on Program Comprehension, IWPC'05, 2005, pp. 97–106, http://dx.doi.org/10.1109/WPC.2005.14.

[8] F.S. Foundation, GNU coding standards, 2021, URL: https://www.gnu.org/prep/standards/html_node/Names.html.

[9] Oracle, Code conventions for the Java programming language, 1999, URL: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html.

[10] P.T. Breuer, J.P. Bowen, Decompilation: The enumeration of types and grammars, ACM Trans. Program. Lang. Syst. (TOPLAS) 16 (5) (1994) 1613–1647.

[11] C. Cifuentes, K.J. Gough, A methodology for decompilation, in: Proceedings of the XIX Conferencia Latinoamericana de Informatica, Citeseer, 1993, pp. 257–266.

[12] D. Brumley, J. Lee, E.J. Schwartz, M. Woo, Native x86 decompilation using Semantics-Preserving structural analysis and iterative Control-Flow structuring, in: 22nd USENIX Security Symposium (USENIX Security 13), USENIX Association, Washington, D.C., 2013, pp. 353–368, URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz.

[13] M. Emmerik, T. Waddington, Using a decompiler for real-world source recovery, in: 11th Working Conference on Reverse Engineering, 2004, pp. 27–36, http://dx.doi.org/10.1109/WCRE.2004.42.

[14] L. Ďurfina, J. Křoustek, P. Zemek, PsybOt malware: A step-by-step decompilation case study, in: 2013 20th Working Conference on Reverse Engineering, WCRE, 2013, pp. 449–456, http://dx.doi.org/10.1109/WCRE.2013.6671321.

[15] K. Yakdan, S. Dechand, E. Gerhards-Padilla, M. Smith, Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study, in: 2016 IEEE Symposium on Security and Privacy, SP, 2016, pp. 158–177, http://dx.doi.org/10.1109/SP.2016.18.

[16] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, M. Smith, No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations, in: NDSS, 2015.

[17] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, REcompile: A decompilation framework for static analysis of binaries, in: 2013 8th International Conference on Malicious and Unwanted Software: "the Americas", MALWARE, 2013, pp. 95–102, http://dx.doi.org/10.1109/MALWARE.2013.6703690.

[18] C. Cifuentes, K.J. Gough, Decompilation of binary programs, Softw. - Pract. Exp. 25 (7) (1995) 811–829.

[19] J. Miecznikowski, L. Hendren, Decompiling java bytecode: Problems, traps and pitfalls, 2002, pp. 111–127, http://dx.doi.org/10.1007/3-540-45937-5_10.

[20] Y. Tashtoush, Z. Odat, I. Alsmadi, M. Yatim, Impact of programming features on code readability, Int. J. Softw. Eng. Appl. 7 (2013) 441–458, http://dx.doi.org/10.14257/ijseia.2013.7.6.38.

[21] A. Jaffe, J. Lacomis, E.J. Schwartz, C.L. Goues, B. Vasilescu, Meaningful variable names for decompiled code: A machine translation approach, in: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 20?30, http://dx.doi.org/10.1145/3196321.3196330.

[22] A.P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, 2009.

[23] A. Goldberg, D. Robson, Smalltalk-80: The Language, in: Addison-Wesley series in computer science, Addison-Wesley, 1989, URL: https://books.google.com.bo/books?id=R7ZQAAAAMAAJ.

[24] C. Béra, M. Denker, Towards a flexible pharo compiler, in: IWST, 2013.

[25] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, E. Herbst, Moses: Open source toolkit for statistical machine translation, in: Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions, Association for Computational Linguistics, Prague, Czech Republic, 2007, pp. 177–180, URL: https://aclanthology.org/P07-2045.

[26] S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, J. Mol. Biol. 48 (3) (1970) 443–453, http://dx.doi.org/10.1016/0022-2836(70)90057-4, URL: https://www.sciencedirect.com/science/article/pii/0022283670900574.

[27] M. Triola, Elementary Statistics, Pearson/Addison-Wesley, 2004, URL: https://books.google.cl/books?id=MX5GAAAAYAAJ.

[28] A. Bacchelli, M. Lanza, R. Robbes, Linking e-mails and source code artifacts, in: 2010 ACM/IEEE 32nd International Conference on Software Engineering, Vol. 1, 2010, pp. 375–384, http://dx.doi.org/10.1145/1806799.1806855.

[29] G. Beniamini, S. Gingichashvili, A.K. Orbach, D.G. Feitelson, Meaningful identifier names: The case of single-letter variables, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension, ICPC, IEEE, 2017, pp. 45–54.

[30] E. Daka, J.M. Rojas, G. Fraser, Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? in: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017, pp. 57–67.

[31] R. Gresta, V. Durelli, E. Cirilo, Naming practices in java projects: An empirical study, in: Proceedings of the XX Brazilian Symposium on Software Quality, 2021, pp. 1–10.

[32] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, M. Mirakhorli, DeepTC-Enhancer: Improving the readability of automatically generated tests, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20, Association for Computing Machinery, New York, NY, USA, 2021, pp. 287–298, http://dx.doi.org/10.1145/3324884.3416622.

[33] T. Sedano, Code readability testing, an empirical study, in: 2016 IEEE 29th International Conference on Software Engineering Education and Training, CSEET, IEEE, 2016, pp. 111–117.

[34] T. Dietterich, Overfitting and undercomputing in machine learning, ACM Comput. Surv. (CSUR) 27 (3) (1995) 326–327.

[35] C. Cifuentes, Reverse Compilation Techniques, Citeseer, 1994.

[36] D.S. Katz, J. Ruchti, E. Schulte, Using recurrent neural networks for decompilation, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2018, pp. 346–356.

[37] M. Shudrak, V. Zolotarev, The new technique of decompilation and its application in information security, in: 2012 Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation, 2012, pp. 115–120, http://dx.doi.org/10.1109/EMS.2012.20.

[38] G. Stiff, F. Vahid, New decompilation techniques for binary-level co-processor generation, in: ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005, IEEE, 2005, pp. 547–554.

[39] R. Bavishi, M. Pradel, K. Sen, Context2Name: A deep learning-based approach to infer natural variable names from usage contexts, 2018, arXiv preprint arXiv:1809.05193.

[40] G. Chen, Z. Wang, R. Zhang, K. Zhou, S. Huang, K. Ni, Z. Qi, K. Chen, H. Guan, A refined decompiler to generate C code with high readability, in: 2010 17th Working Conference on Reverse Engineering, 2010, pp. 150–154, http://dx.doi.org/10.1109/WCRE.2010.24.

[41] Q. Chen, J. Lacomis, E.J. Schwartz, C. Le Goues, G. Neubig, B. Vasilescu, Augmenting decompiler output with learned variable names and types, in: 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 4327–4343.

[42] X. Lv, Y. Xie, X. Zhu, L. Ren, A technique for bytecode decompilation of PLC program, in: 2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference, IAEAC, 2017, pp. 252–257, http://dx.doi.org/10.1109/IAEAC.2017.8054016.

[43] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, B. Vasilescu, Dire: A neural approach to decompiled identifier naming, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2019, pp. 628–639.

[44] B. Vasilescu, C. Casalnuovo, P. Devanbu, Recovering clear, natural identifiers from obfuscated JS names, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, 2017, p. 683?693, http://dx.doi.org/10.1145/3106237.3106289.

[45] V. Raychev, M. Vechev, A. Krause, Predicting program properties from" big code", ACM SIGPLAN Not. 50 (1) (2015) 111–124.

Juan Pablo Sandoval Alcocer is an Assistant Professor at the Department of Computer Science, School of Engineering, Pontificia Universidad Católica de Chile. He received a Ph.D. degree in computer science from the University of Chile. He is part of the Software Engineering and Intelligent Systems Laboratory (SEIS Lab). His research interests lie in software engineering, more specifically in the fields of software maintenance, mining software repositories, software performance, software visualization, and software testing. He participated as a reviewer expert in various prestigious conferences and journals in the software engineering field. He is also a member of the Pharo community.

Harold Camacho Jaimes received his B.Sc. in software engineering from the Universidad Catolica Bolivana San Pablo, Cochabamba, Bolivia. He was part of the Exact Sciences and Engineering Research Center (CICEI) at the same university. He currently works as a software engineer at AssureSoft, an outsourcing software company situated in Bolivia.

Geraldine Galindo-Gutierrez is a Ph.D. student in Computer Science and an auxiliary researcher at the Exact Sciences and Engineering Research Center (CICEI) at the Universidad Católica Boliviana. Her research interests include software testing and code quality.

Andrés Neyem is a professor in the Computer Science Department at Pontificia Universidad Católica de Chile. He received his Ph.D. in Computer Science from the Universidad de Chile. He is a researcher at the National Center for Artificial Intelligence (CENIA) and the Software Engineering and Intelligent Systems Laboratory (SEIS Lab). His research interests include Software Engineering, Mobile and Cloud Computing, Machine Learning for Intelligent Systems, Engineering and Medical Education, and Extended Reality. In these research areas, on the one hand, he has published a wide range of papers in conference proceedings and journals, and, on the other hand, he has developed several software products for these types of cloud-based mobile platforms.

Alexandre Bergel is a computer scientist at RelationalAI, Switzerland. Until 2022, he was an Associate Professor and researcher at the University of Chile. Alexandre Bergel and his collaborators carry out research in software engineering. His focus is on designing tools and methodologies to improve the overall performance and internal quality of software systems and databases by employing profiling, visualization, and artificial intelligence techniques. Alexandre Bergel is a member of the editorial board of Empirical Software Engineering and has authored four books in the fields of data visualization, artificial intelligence, and the Pharo programming language.

Stéphane Ducasse is the Inria Research Director. He leads the RMoD team http://rmod.lille.inria.fr. Stéphane is an expert in language design and re-engineering. He works on traits. Traits have been introduced in Pharo, Perl, PHP, and under a variant into Scala, Groovy, and Fortress. He is a software quality expert on program understanding, program visualizations, reengineering, and metamodeling. He is one of the developers of Moose, an open-source software analysis platform http://www.moosetechnology.org/ Stéphane created Synectique, a company that builds dedicated tools for advanced software analysis. He is one of the leaders of Pharo http://www.pharo.org/ a dynamic reflective object-oriented language supporting live programming. Since 2013, Stéphane built the industrial Pharo consortium http://consortium.pharo.org. He works regularly with companies (Thales, Wordline, Siemens, Berger-Levrault, Arolla, ...) on software evolution problems and has written a couple hundred articles and several books.