Proceedings of the 9th International Conference on Applied Informatics
Eger, Hungary, January 29–February 1, 2014. Vol. 2. pp. 309–316
doi: 10.14794/ICAI.9.2014.2.309

A closer look at software refactoring using symbolic execution*

Csaba Szabó, Maroš Kotul, Richard Petruš

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Csaba.Szabo@tuke.sk, Maros.Kotula@student.tuke.sk,
Richard.Petrus@student.tuke.sk

Abstract

Fowler's classical definition of program refactoring as it appeared in [1] is many times broken in CASE tools, which claim non-refactorings to be refactoring operations. On other hand-side, there are often forgotten refactorings. According to the original definition [1], "Refactoring is the process of changing a software in a such way that it does not alter the external behavior of the code yet improves its internal structure." Many times, refactoring is done by hand, which needs a detailed verification. Based on the commutativity diagram of symbolic execution presented by King in [2] and the above definition of refactoring, we expressed the principle of program refactoring and our look at it using symbolic execution. These are presented in this paper as well as the overall principle of automated refactoring evaluation. For the question of "is it a refactoring?" we offer an answer using Java PathFinder and Symbolic PathFinder for the Java programming language.

Keywords: Java, refactoring, symbolic execution

MSC: 68N15; 68N20

1. Introduction

Computer programs are often the subject of inevitable evolution changes. These changes often bring a certain amount of uncertainty about wheter they were successful and thus the program is still valid. In general, as programs become bigger

*This work was supported by the Cultural and Educational Grant Agency of the Slovak Republic, Project No. 050TUKE-4/2013: "Integration of Software Quality Processes in Software Engineering Curricula for Informatics Master Study Programme at Technical Universities – Proposal of the Structure and Realization of Selected Software Engineering Courses."

and more complex, the problems that arise with these changes become more significant. To deterimine the validity of these changes tests are often performed. These tests can be manual or automatic. Still despite of these tests there is a fair chance that errors can persist, as the test cases can contain errors as well or there is simply a lack of some test cases. With this in mind, we tried to look at the problem from a slightly different point of view. We use symbolic execution to analyze program methods. In our setup, we have two programs or methods. These methods contain the same code, one before the refactoring changes and the other one after these changes. Due to the basic substance of refactoring process these methods should behave equally. To analyze this behaviour we use the aforementioned symbolic execution. We use these analysis to compare the behavior of these methods and then form a conclusion about the validity of this refactoring. To execute symbolic execution over a method it is possible to take a manual or an automatic approach. Logically the latter one comes as the better option. To do this we use a tool called Java PathFinder [3] and its supplement called Symbolic PathFinder [4].

2. Background

This section presents certain definition, which ones are relevant and necessary to provide for a complete understanding of our approach. The definitions presented will be referred in the second half of the paper.

2.1. Category theory

A category is a graph with a rule for composing arrows to give another arrow. This rule is subject to certain condition. The formal definition as presented in the works [5, 6, 7, 8] is given below.

Definition 2.1. A category is a collection of *objects* and *morphisms* such that for each pair of objects A, B exists a set of $\hom(A, B)$ of morphisms between them such that:

- for each object X the set $\hom(X, X)$ contains the *identity morphism*,
- for each triple of objects X, Y, Z and pair of morphisms $f \in \hom(X, Y)$ and $g \in \hom(Y, Z)$, there is a *composite morphism* $f \circ g \in \hom(X, Z)$,
- for each pair of objects X, Y and morphism $f \in \hom(X, Y)$, the identity morphisms are *left* and *right units for composition*: $id_X \circ f = f = f \circ id_Y$, and
- for each 4-tuple of objects X, Y, Z, W and triple of morphisms $f \in \hom(X, Y)$, $g \in \hom(Y, Z)$ and $h \in \hom(Z, W)$ composition is associative: $(f \circ g) \circ h = f \circ (g \circ h)$.

Morphism $f \in \hom(X, Y)$ is also written $f : X \to Y$.

The category is always based on a graph; therefore it is visual, clear and easy to understand [9].

2.2. Symbolic execution

In King's work [2], we can find the following definition of symbolic execution.

Definition 2.2. Symbolic execution is a natural extension of normal execution, providing the normal computations as a special case. Operators of the language are extended to operate on symbolic formulas. E.g. real data could be replaced by arbitrary symbols in the computations.

Symbolic execution could be used in program testing to identify path conditions, which could be used in test case generation for statement, path or state coverage testing as stated in King's work [2]. Others, like Pócza et al. [10] and Kozsik et al. [11], use symbolic execution in practice over the .NET programming platform and C/C++ programming languages. The main principle used here is to substitute a set of symbolic variables into the role of the real variables used in the program. A simulation of the program run is then executed while following and recording value changes of each symbolic variable.

Symbolic execution was expressed by King [2] as the commutativity diagram shown in Fig. 1.

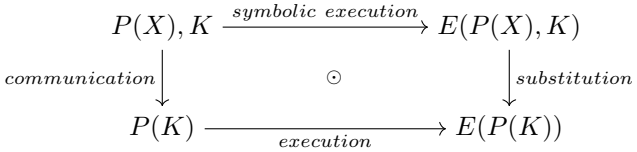

Figure 1: Commutativity diagram of symbolic execution by King [2]

2.3. Refactoring

Fowler [1] and Rutar et al. [12] define refactoring as shown below.

Definition 2.3. Refactoring is a process of changing code structure to improve its structural aspects while preserving its behavior.

Every refactoring produces a small change, a set of these changes can produce a bigger restructuralization of code. Due to the small changes every refactoring makes, there is a smaller chance of error introduction. The reasons to introduce refactoring to a program can be to improve the effectiveness of the code or to make the code better readable and understandable.

This change can thus bring:

1. better understanding of code
2. easier code modification in the future

3. easier enhancement of code
4. better scalability
5. easier code maintenance
6. better readability

3. Refactoring in terms of category theory

To preserve compatibility to King's commutativity diagram [2] in Fig. 1, we will use the following notation [13]:

- Program structure will be denoted as $P(X)$, where P is the program and X represents program interface.
- $P(X), K$ is the program in the environment called K.
- $P(K)$ is the program with environment values passed to its interface.
- E will be used for external behavior notation, which, in fact, can be expressed in two different ways:
  1. $E(P(K))$ denotes real external behavior,
  2. $E(P(X))$ denotes symbolic external behavior.

By the above notation, we can express the principle of refactoring as presented in Fig. 2.

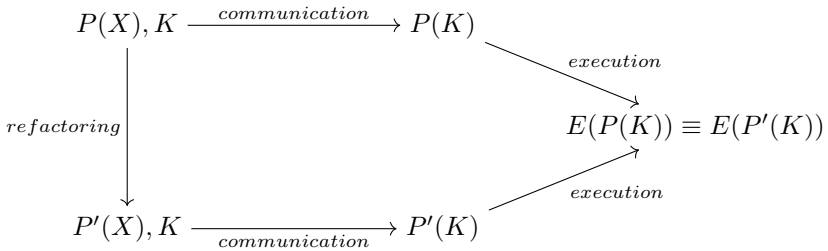

Figure 2: The principle of refactoring

This principle shows that morphism $refactoring : P(X) \to P(X)$ changes the program P to P' without a change neither in its interface nor external behavior (X, K remain unchanged and $E(P(K)) = E(P'(K))$).

Now, we can introduce symbolic execution into Fig. 2. It will be based on Fig. 1. As symbolic execution represents a powerful alternative to real execution, software refactoring using symbolic execution could be expressed as in Fig. 3.

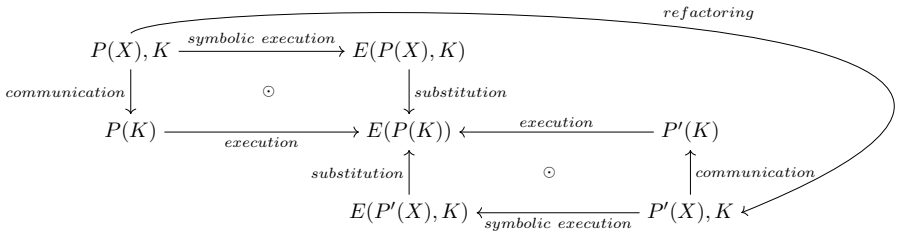

Figure 3: Software refactoring using symbolic execution

4. Implementing refactoring evaluation

Next step of our research is evaluation of practical applicability. As we already introduced, implementation is done in the Java programming language. More precisely, it is for the Java programming language. The reason is that we use the selected PathFinder [3, 4] libraries.

In our approach we use two programs, the original one (P) and the refactored one(P'). We put them through symbolic execution into the Symbolic Path?nder environment. Then, we use the output we get from the execution to analyze possible behavioral changes. There are two conditions these two programs must meet for the change to be a refactoring:

1. the output must be the same for both P and P' for the same input values and
2. the number of execution paths must be equal.

Next, we need to create our testware, which in this case implements symbolic execution and analyzes the above mentioned properties of the pair of programs. The implementation consists of a single *DriverClass*, which instantiates two classes – the *original* and *refactored* one. These classes contain the methods to compare. The Java Symbolic PathFinder setup is included with Fig. 4.

```
target=example.DriverClass
classpath=${jpf-symbc}/build/examples
sourcepath=${jpf-symbc}/src/examples
symbolic.method=example.ClassOriginal.myMethod(sym#sym)
listener=.symbc.SymbolicListener
```

Figure 4: Symbolic PathFinder setup for our example

The main principle of comparison is located in only two lines of code in the *DriverClass*, see Fig. 5.

```
int a = original.myMethod(1, 2);
int b = refactored.myMethod(original.getX(), original.getY());
```

Figure 5: The most important lines of code in the *DriverClass*

For a demonstration we provide the following pieces of code, which contain one simple method. Fig. 6 shows the original code.

```
public int myMethod(int x, int y) {
     if(x > 0){
          return 1;
     }else{
          if(y > 0){
               return 2;
          }
          else{
               return 3;
          }
     }
}
```

Figure 6: Example method before refactoring

Code in Fig. 7 contains the same code after a small refactoring consisting of two reverse conditional changes.

```
public int myMethod(int x, int y) {
     if(x <= 0){
          if(y <= 0){
               return 3;
          }else{
               return 2;
          }
     }else{
          return 1;
     }
}
```

Figure 7: Example method after refactoring

As we can see from the output presented in Fig. 8, both methods provided the same output values for the same input values, thus preserving the same behavior.

```
*************Summary***************
PC is:constraint # = 1
x_1_SYMINT[1] > CONST_0
Return is: CONST_1
Method original: output[1]
Method refactored: output[1]
*************Summary***************
PC is:constraint # = 2
y_2_SYMINT[1] > CONST_0 &&
x_1_SYMINT[-1000000] <= CONST_0
Return is: CONST_2
Method original: output[2]
Method refactored: output[2]
*************Summary***************
PC is:constraint # = 2
y_2_SYMINT[-1000000] <= CONST_0 &&
x_1_SYMINT[-1000000] <= CONST_0
Return is: CONST_3
Method original: output[3]
Method refactored: output[3]
```

Figure 8: Results of refactoring evaluation

5. Conclusion

In this work we aimed to find a relation between symbolic execution and refactoring. We used commutativity diagrams to express this relation. This is the main contribution of present article.

Theories also need an evaluation. We did this by experimentally using our approach on test codes. These codes represented one actual program before and after a refactoring. To actually execute symbolic execution over code we used a tool called Java PathFinder and Symbolic PathFinder.

Using these tools we could get output form the executions over our pieces of test code. Then, we used these outputs to analyze the change in behavior between these two codes. In our experiments, we discovered that the refactorings made no change in behavior and therefore where valid. It also proved, that our approach is suitable to give an answer to the question "is the change a refactoring?".

Even though we got satisfiable results for our test cases, one must consider the restrictions that come with symbolic execution and Java PathFinder. Not any piece of code can be executed this way. Because of that our examples where designed to fit into the capabilities of Java PathFinder. We can still say that based on the results we achieved, that this approach is suitable. A usage in real development and in bigger scales is still questionable due to the mentioned restrictions.

References

[1] Fowler, M., *Refactoring – Improving the Design of Existing Code*, Addison-Wesley Professional, 1st ed. (1999).

[2] King, J. C., Symbolic Execution and Program Testing, *Communications of the ACM* Vol. 19 No. 7 (1976).

[3] *Java PathFinder*, `http://babelfish.arc.nasa.gov/trac/jpf`

[4] *Symbolic Java PathFinder*, `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc`

[5] Novitzká, V., Mihályi, D., Verbová, A. *Coalgebras as Models of System's Behaviour,* in Proc. of AEI 2008, Int. Conf. on Applied Electrical Engineering and Informatics ?2008, Athens, Greece, pp. 31–36. (2008)

[6] Slodičák, V., Szabó, Cs., *Recursive Coalgebras in Mathematical Theory of Programming*, 8th Joint Conference on Mathematics and Computer Science, Selected Papers, Komárno, Slovakia, July 14-17, 2010, Győr, Hungary, Novadat Bt., pp. 385-394 (2011)

[7] Goguen, J.A., A Categorical Manifesto, *Math. Structures in Comp. Sci.,* Vol. 1, No. 1, (1991)

[8] Stay, M., *Category Theory for the Java Programmer,* Nov. 3, 2007, url: `http://reperiendi.wordpress.com/2007/11/03/category-theory-for-the-java-programmer/` (2007)

[9] Zhang, X., Miao, H., Zeng, H., *The Syntactic and Semantic Model of Web Services Composition Based Category,* in Proc. of Int. Conf. on Advanced Comp. Theory and Engineering (ICACTE'08), Phuket, Thailand, pp. 444–449 (2008)

[10] Pócza, K., Biczó, M., Porkoláb, Z., Towards Effective Runtime Trace Generation Techniques in the .NET Framework, *Journal of .NET Technologies*, Vol. 4., Number 1-3, pp. 141-150 (2006)

[11] Kozsik, T., Pataki, N., Szügyi, Z., C++ Standard Template Library by Infinite Iterators, *Annales Mathematicae et Informaticae*, 38:75-86, (2011)

[12] Rutar N., Almazan Ch. B., Foster J.S., *A Comparison of Bug Finding Tools for Java*, ISSRE '04 Proceedings of the 15th International Symposium on Software Reliability Engineering, Pages 245-256. (2004)

[13] Szabó, Cs., Kotuľa, M., Petruš, R., *The First Proposal on Objects and Morphisms of the Software Evolution Category*, In: SAMI 2014 : IEEE 12th International Symposium on Applied Machine Intelligence and Informatics : proceedings : January 23-25, 2014, Herlany, Slovakia. - Danvers : IEEE, P. 59-62. (2014)