RefactoringScript: A Script and Its Processor for Composite Refactoring

Lincho Yang  
Waseda University  
Tokyo, Japan  
young@fuji.waseda.jp

Tomoyuki Kamiya  
Waseda University  
Tokyo, Japan  
kamiya7140@akane.waseda.jp

Kazunori Sakamoto  
National Institute of Informatics  
Tokyo, Japan  
exkazu@nii.ac.jp

Hironori Washizaki  
Waseda University  
Tokyo, Japan  
washizaki@waseda.jp

Yoshiaki Fukazawa  
Waseda University  
Tokyo, Japan  
fukazawa@waseda.jp

Abstract—Refactoring is widely recognized as a method to improve the internal qualities of source code. However, manual refactoring is time-consuming and error-prone. Consequently, many tools to support automated refactoring have been suggested, but most support only unit and simple refactoring, making it difficult to perform composite refactoring (e.g., introducing a design pattern) where a refactoring set is applied at one position or the same refactoring operation is applied at multiple positions. In this paper, we propose a novel script language and its processor to describe how and where to refactor by a model expressing source code\textsuperscript{*1}. Evaluations indicate that our language and processor allow refactoring steps to be described as scripts, which can be easily replayed and reused for multiple projects.

Keywords: Refactoring; Code Manipulation;

I. INTRODUCTION

Refactoring, which is defined as “a technique to improve the design of the internal structure of software without changing its external behaviors”, has become commonplace in recent years. Although the most common refactoring techniques (e.g. extract and change field names) have been organized into patterns, manual refactoring is time-consuming and error-prone. To resolve this problem, many automatic refactoring tools and methods have been proposed.

Here, we define the following two terms, basic refactoring and composite refactoring [6], to specify two types of refactoring. Basic refactoring refers to a simple refactoring that cannot be decomposed further. In this paper, basic refactorings indicate the standard refactoring features of Eclipse. On the other hand, composite refactoring refers to a combination of basic refactorings. Combinations include applying several refactorings in one place, the same refactoring to several places, or both.

Although many current tools support automatic execution of basic refactoring, there is no mechanism to define and apply composite refactoring. As suggested by Vakilian et al. [9] in their survey on the trend of using refactoring in Eclipse by examining recordings, composite refactoring is very common.

\textsuperscript{*1} The preliminary idea of RefactoringScript has been originally proposed in [12] in Japanese. [12] has proposed the RefactoringScript with JRuby implementation and did not achieve the statement-level operations. In contrast this paper introduces RefactoringScript with Scala implementation. Moreover, we have added the capability of manipulating statements to make RefactoringScript support more complicated refactoring.

The complex composite refactorings such as introducing a design pattern were explicated in [4] and [5].

However, manual composite refactoring has a high execution cost since programmers have to locate the target code and select a menu or use a keyboard shortcut to call a basic refactoring every time. Additionally, as the numbers of locations and refactorings increase, performing each basic refactoring correctly becomes more difficult, and the risk of omissions increases. Although many patterns have been established for refactoring, majority tools cannot record and reuse the processes of the patterns. Therefore, it is difficult to apply the frequently used composite refactorings to cross-projects. Eclipse can record and replay refactoring operations as a script, but it is used to help programmers upgrade an older version of a library once a newer version is distributed. It is impossible to create a script arbitrarily and describe the steps of refactoring freely.

In this paper, we propose RefactoringScript, which contains a script that can be used to describe the processes of composite refactoring and its corresponding processor to perform refactoring. We address the following research questions:

\begin{itemize}
  \item RQ1 Is it possible to script and apply composite refactoring (locations and actions) concisely and accurately?
  \item RQ2 Compared to the case without this tool, is composite refactoring performed correctly?
  \item RQ3 Compared to the case without this tool, is the cost of composite refactoring reduced?
  \item RQ4 Is it possible to reuse the composite refactoring in cross-projects?
\end{itemize}

The contributions of this paper are:

\begin{itemize}
  \item A RefactoringScript language to describe the processes of composite refactoring
  \item A RefactoringScript processor to apply scripted composite refactoring
  \item Implementation of a RefactoringScript language and its processor as an Eclipse plug-in
  \item Evaluation of a RefactoringScript language and its processor to show its utility
\end{itemize}

Our paper is organized as follows. Section II provides motivating examples. Then Section III describes the proposed
RefactoringScript language and its processor, while Section IV presents the results and discussion of our experimental evaluation. Finally, Section V concludes the paper.

II. BACKGROUND

We consider three cases as motivating examples.

A. Renaming Relevant Elements

According to [9], a common combination of refactorings is renaming both of a field and its related elements. For example, after changing a field name, the names of its accessors should also be changed.

However, current refactoring tools only change the definition and references when applying rename refactoring to a field. For instance, if rename refactoring is invoked to change the name of field “page” in Listing 1 to “pageCount”, the name of the corresponding accessor will not be changed automatically. Hence, programmers have to invoke the rename refactoring again for renaming the accessor. Figure 1 is an example of RefactoringScript to do these two refactorings.

B. Applying Coding Conventions

Many projects have their own coding conventions. To enhance maintainability of the entire code, especially for team development, all team members must observe the coding convention established before development. Reference [7] and [8] summarize some underlying rules, which can be used and modified freely. For example, one rule (27) places the underscore prefix or suffix of the name for a private, protected field, while another rule (44) avoids overloading the method.

For a project with an already inflated scale, the following is necessary to apply these conventions:

   a) Among the protected or private fields, extract all names without an underscore prefix (or suffix), and execute the rename method.(Example script is shown in Figure 2.)

   b) Acquire all methods that have the same name and the same number of arguments from a specific class, and execute the rename method.

Coding conventions can be used in multiple projects. When a new coding convention is applied to an existing source code or the old coding convention of a project is changed, the execution cost for refactoring all relevant places is very high. Furthermore, as the number of places that need refactoring increases, the risk of making mistakes increases.

C. Introducing Design Patterns

The transformation to introduce a design pattern involves many iterations of refactoring. For example, introducing a Visitor Pattern includes the following two types of refactoring:

   a) Move Method

   b) Rename Method (to avoid name collisions, add “visit” to the beginning of the name of the method that has been moved.)

III. REFACTORINGSCRIPT LANGUAGE AND ITS PROCESSOR

In this section, we describe the design of the proposed RefactoringScript language and its processor.

A. Overview

On account of the excellent features of Scala, such as the seamless Java interoperability and the scripting language. We adopted Scala as our script language and developed a DSL base on it. In addition, we employed Eclipse JDT to perform the basic refactorings provided by Eclipse. The RefactoringScript consists of two components.

- **RSCore**: The fundamental part, which includes elements of the RefactoringScript language and its processor.
- **RSUI**: The user interface part, such as an editor to create or modify script, and a menu to execute a script by inputting script into the interpreter of RefactoringScript in RSCore.

The procedure for a user to apply a script to a workspace, and the interaction between the user and RefactoringScript processor are as follows:

1. User creates and edits the script in the editor.
2. User activates the core component by specifying a script file.
3. Processor inputs the script into the interpreter.
4. Interpreter runs the script and applies it to the user’s workspace.

*2 https://github.com/hugh3166/RSCore
*3 https://github.com/hugh3166/RSEditor
*4 https://github.com/hugh3166/RSLauncher
(5) User is notified of the script execution result.

B. Language

Herein, we describe the elements of the RefactoringScript language.

1) Code Entity and Code Entity Collection

Java elements of JDT provide APIs, which are suitable for searching a particular element from the workspace. However, the Java elements do not have APIs that allow the conditions to be specified in detail to determine the specified elements. Two types of APIs are added to Code Entity (CE) which is a class based on Java Element:

- APIs to analyze and search. For example, the select method which we will describe later.
- APIs to trace the tree structure of the code in the description similar to the simple natural language. For example, we prefer to use c.methods rather than c.getMethods() to acquire all methods of c.

Table I shows the correspondence between Java Element and CE. An indentation in the table represents the containment relationship of the package or the class inheritance. It should be noted that RSWorkspace differs slightly from the other CEs; RSWorkspace represents a reference to the current workspace, and is a starting point to find the other CEs.

The Code Entity Collection (CEC) represents a set of CEs and provides APIs that can search for CEs included in the set. An example script to perform a search is introduced in the next unit.

2) Query Selector and Qualifier

To search for a CE from CEC, we can use select method by combining SearchParams, QuerySelector, and Qualifier in the following format:

```
CEC.select(QuerySelector (Qualifier (SearchParams)))
```

```
QuerySelector ::= "By.name"|"By.namereg"|"By.modifier"|"By.typename"
Qualifier ::= ""|"With.or"|"With.and"|"With.out"
```

QuerySelector is a keyword that specifies the Search Key, which can be one of: name, regular expression of name, access modifier, and type name. Table II shows each CE and the corresponding combination of the search key and query selector, where O indicates that QuerySelector can search CE using the search key, and X indicates it cannot. For example, a set of RSProjects can be searched by key elements with a name, but cannot with an access modifier. Qualifier is a keyword to specify how to interpret the given search parameters with OR, AND, or NOT. However, it can be omitted when a qualifier is not required (if there is only one search parameter). Figure 3 shows three examples of using the select method. The select method has also been used in Figure 1 and Figure 2 in the first line.

3) Action

A refactoring operation for CE / CEC is called an action. With parameters (params), an action is expressed in the following format:

```
CE/CEC.Action(params)
```
TABLE IV. COMPARISON OF NUMBER OF LINES OF SCRIPT IN REFAC TORIZATIONSCRIPT WITH JAVA

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>RefactoringScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX1</td>
<td>42</td>
<td>10</td>
</tr>
<tr>
<td>EX2</td>
<td>143</td>
<td>14</td>
</tr>
<tr>
<td>EX3</td>
<td>107</td>
<td>9</td>
</tr>
<tr>
<td>EX4</td>
<td>48</td>
<td>12</td>
</tr>
</tbody>
</table>

Unit: Lines

TABLE V. COMPARISON OF EXECUTION TIME FOR REFACTORING BY SCRIPT AND MANUALLY

<table>
<thead>
<tr>
<th>Experiment</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX1 (Manually)</td>
<td>7</td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>6.0</td>
</tr>
<tr>
<td>EX1 (Script)</td>
<td>-</td>
<td>10</td>
<td>-</td>
<td>5</td>
<td>14</td>
<td>9.7</td>
</tr>
<tr>
<td>EX4 (Manually)</td>
<td>-</td>
<td>30</td>
<td>-</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>EX4 (Script)</td>
<td>22</td>
<td>-</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>16.0</td>
</tr>
</tbody>
</table>

Unit: Minutes

TABLE VI. COMPARISON OF ACCURACY OF REFACTORING BY SCRIPT AND MANUALLY

<table>
<thead>
<tr>
<th>Experiment</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX1 (Manually)</td>
<td>27</td>
<td>-</td>
<td>30</td>
<td>-</td>
<td>-</td>
<td>28.5</td>
</tr>
<tr>
<td>EX1 (Script)</td>
<td>-</td>
<td>30</td>
<td>-</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>EX4 (Manually)</td>
<td>-</td>
<td>95</td>
<td>-</td>
<td>96</td>
<td>93</td>
<td>94.7</td>
</tr>
<tr>
<td>EX4 (Script)</td>
<td>96</td>
<td>-</td>
<td>96</td>
<td>-</td>
<td>-</td>
<td>96</td>
</tr>
</tbody>
</table>

Unit: Places

TABLE VII. APPLIED REFACTORINGSCRIPT TO OPEN SOURCE PROJECTS

<table>
<thead>
<tr>
<th>Project</th>
<th>Experiment</th>
<th>Number of Files</th>
<th>Number of Lines</th>
<th>Applying Places</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>EX1</td>
<td>3</td>
<td>68</td>
<td>16 fields</td>
</tr>
<tr>
<td>P1</td>
<td>EX4</td>
<td>2</td>
<td>18</td>
<td>6 fields 12 methods</td>
</tr>
<tr>
<td>P2</td>
<td>EX3</td>
<td>6</td>
<td>20</td>
<td>6 classes 6 methods</td>
</tr>
</tbody>
</table>

Figure 4. Example script of EX2

An action parameter may be specified as the minimum required when performing refactoring. Table III summarizes the types of the actions which have been supported and the parameters which can currently be specified. For example, CE.rename("newname") will apply rename refactoring to CE. As we showed at line 3 and 4 in Figure 1, or line 5 in Figure 2.

C. Processor

In this tool, the RefactoringScript language can be regarded as an internal DSL of Scala, and its processor is also implemented with Scala. Programmers only need to focus on the descriptions of searching and handling CE, because Scala expressions and the built-in functions or libraries of both of Java and Scala are available in the script. Moreover, the Scala Interpreter was incorporated directly into the processor without implementing a new interpreter.

IV. EVALUATION

A. Evaluation Design and Results

To evaluate the describability, accuracy, execution cost, and reusability of RefactoringScript, we conducted subject experiments and case studies for the four composite refactorings, which were selected by considering trends of refactoring [9] and coding conventions [7, 8].

EX1. Assign a prefix to the name of every private field for all classes in a specific package.
EX2. Generate template method from subclasses into a superclass.
EX3. Encapsulate classes with Factory.
EX4. Change the name of a specified field in a package and the name of the corresponding accessor.

1) Describability

For EX1, 2, 3, 4, we measured the lines of code for processing refactorings in the Java language and the RefactoringScript language (Table IV). Note that the Java projects used as experimental objects are also the test data used for testing RSCore.

For EX2, we used the simplified experimental code in Listing 2. Two subclasses have the same method name, “startGame”, but they have different conditional structures. For simplicity, we just extract the conditional structures to form two new methods with same name, and pull up the two “startGame” methods and one of the extracted methods into a superclass. Because the two “startGame” methods are the same method in the superclass, they can be regarded as a template method. Afterwards, the subclasses can change the behavior of the “startGame” by overriding the method “start” or “extract” without overriding the “startGame” method directly, such as the class Game2 shown at the right of Listing 2.

2) Accuracy and Execution Cost

We conducted the following subject experiments to compare the accuracy and execution of composite refactoring cost between using RefactoringScript and simply using Eclipse refactoring. The experimental objects are the sample projects prepared for the experiments. It should be noted that in consideration of the similarity of operation difficulty and the influence of prior knowledge, we only used EX1 and 4 as the experimental objects. In addition, for simplicity, we chose int type fields to be the target fields, and wrote the script to extract the fields by type in EX4.
Experimental Subjects: Five Information Engineering undergraduate and graduate students (P1–P5)

Procedure: Divide subjects into two groups. Make one group conduct EX1 by Eclipse, then EX4 by RefactoringScript, and make the other group conduct EX1 by RefactoringScript, then EX4 by Eclipse. Then measure the time necessary to complete refactorings and the places where refactoring is applied correctly.

Tables V and VI summarize the results of this subject experiment. Table V shows that subject P1 took seven minutes to do EX1 by Eclipse, 22 minutes to do EX4 by script, while Table VI shows that subject P1 applied refactoring correctly at 27 of 30 places in EX1 by Eclipse, but 96 of 96 places in EX4 by script.

1) Reusability (Case Study)

We applied the EX1, 3, 4 to open source projects P1*5 and P2*6 with both manual refactoring and RefactoringScript, and then got the mechanical differences. By verifying the results with the goals, it is confirmed that the proposed method is able to process the object refactoring. Because the focus is on only the refactoring operation, we selected projects with a moderate scale as the experimental material. Table VII lists the projects, kinds of experiments, numbers of files, numbers of lines affected by refactoring, and the applied places.

B. Discussion

1) Describability

RQ1 Is it possible to script and apply refactoring operations (applying places and actions) concisely and accurately?

In all four cases, the number of lines of script written in RefactoringScript is 1/4 to 1/10 of the script written in Java. The reduction is attributed primarily to two reasons:

- API allows CE to be flexibly searched, and conditional statements are less likely to become a nest structure.
- Processes not directly related to refactoring (e.g. acquiring workspace) do not have to be described.

In EX2, the project, package, class, and method entities are searched by name, but a statement is searched by the type defined at the ASTNode class in JDT. Figure 4 shows the example script of EX2. Additionally, even on a scale like EX3, the number of lines of script written in RefactoringScript can be as few as 10. Because the RefactoringScript language is specialized to describe the processes and search locations for refactoring, concise scripting can be realized.

2) Accuracy and Execution Cost

RQ2 Compared to the case without this tool, is composite refactoring executed correctly?

RQ3 Compared to the case without this tool, is the cost of composite refactoring reduced?

With regard to the execution cost, manual refactoring required slightly less time in each experiment. Based on the feedback from the subjects, this is likely because learning RefactoringScript takes some time. In fact, some of the subjects commented:

- “I was confused by the script language idiom.”
- “I think that RefactoringScript can reduce the time once I learned how to write with it.” (In this experiment, the answers to the examples and script pieces required by the experiment were distributed as material.)

On the other hand, feedback regarding manual composite refactoring indicated a desire for an automatic method:

- “I do not want to refactor more complex objects manually.” (For example, when the number of applicable places is enormous).
- “Firstly, I prefer not to do simple mechanical work manually.”

Therefore, we believe that once programmers become familiar with RefactoringScript, the burden of refactoring can be reduced.

With regard to accuracy, all the scripts written by the subjects worked properly using the script case, whereas the manual composite refactoring with Eclipse contained the following mistakes: renamed fields that are not specified (EX1) and fields renamed correctly, but the names of the accessors were incorrect (EX4). These mistakes indicate that the script contributes to correct composite refactoring.

3) Reusability

RQ4 Is it possible to reuse the refactoring operations in

Listing 2. Example to generate template method (Left: before refactoring; Right: after refactoring)

<table>
<thead>
<tr>
<th>package p;</th>
<th>package p;</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class Game{</td>
<td>public class Game{</td>
</tr>
<tr>
<td>protected int playerCount = 0;</td>
<td>protected int playerCount = 0;</td>
</tr>
<tr>
<td>public void start(){}</td>
<td>public void start(){}</td>
</tr>
<tr>
<td>System.out.println(&quot;Game Start&quot;);</td>
<td>System.out.println(&quot;Game Start&quot;);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>public class Game1 extends Game{</td>
<td>public class Game1 extends Game{</td>
</tr>
<tr>
<td>public void startGame(){</td>
<td>public void startGame(){</td>
</tr>
<tr>
<td>start();</td>
<td>start();</td>
</tr>
<tr>
<td>if (playerCount != 0){</td>
<td>if (playerCount != 0){</td>
</tr>
<tr>
<td>System.out.println(&quot;Restart&quot;);</td>
<td>System.out.println(&quot;Restart&quot;);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>public class Game2 extends Game{</td>
<td>public class Game2 extends Game{</td>
</tr>
<tr>
<td>public void startGame(){</td>
<td>public void startGame(){</td>
</tr>
<tr>
<td>start();</td>
<td>start();</td>
</tr>
<tr>
<td>if (playerCount &gt;= 0){</td>
<td>if (playerCount &gt;= 0){</td>
</tr>
<tr>
<td>playerCount++;</td>
<td>playerCount++;</td>
</tr>
<tr>
<td>System.out.println(&quot;Join&quot;);</td>
<td>System.out.println(&quot;Join&quot;);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>public class Game1 extends Game{</td>
<td>public class Game1 extends Game{</td>
</tr>
<tr>
<td>protected void extracted(){</td>
<td>protected void extracted(){</td>
</tr>
<tr>
<td>if (playerCount != 0){</td>
<td>if (playerCount != 0){</td>
</tr>
<tr>
<td>System.out.println(&quot;Restart&quot;);</td>
<td>System.out.println(&quot;Restart&quot;);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>public class Game2 extends Game{</td>
<td>public class Game2 extends Game{</td>
</tr>
<tr>
<td>protected void extracted(){</td>
<td>protected void extracted(){</td>
</tr>
<tr>
<td>if (playerCount &gt;= 0){</td>
<td>if (playerCount &gt;= 0){</td>
</tr>
<tr>
<td>playerCount++;</td>
<td>playerCount++;</td>
</tr>
<tr>
<td>System.out.println(&quot;Join&quot;);</td>
<td>System.out.println(&quot;Join&quot;);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

---

*5 https://github.com/shigenobu/acbook-wa710
*6 http://code.google.com/p/jslideshare/
other projects?

The scripts used in EX1 and EX4 can be applied to other projects without substantial modification. Most projects require the following changes: package name of the object and action parameters (e.g. in EX4, P1 changes field names ‘created’, ‘updated’, ‘executed’ to ‘createdAt’, ‘updatedAt’, ‘executedAt’ as well as the corresponding accessor names). However, these elements are project-specific and are the minimum parameters that the user must specify for each project.

C. Limitations

1) Lazy Evaluation

In this tool, it is impossible to reflect the effect of refactoring on the CE. When multiple actions are performed on the same CE, the specified CE must be searched in each case. This problem should be resolved by introducing the lazy evaluation, which is a mechanism to set aside the query search for CE until execution. In addition, there is almost no issue to apply many basic refactorings to a particular CE.

2) Behavior Preservation

We have tried to reduce the risk of breaking code by using Eclipse’s refactoring features in our tool. The preconditions provided by JDT are checked before performing every basic refactoring, and warnings will be prompted as well if operations will cause compilation errors. However, there are still some operations that may change the behavior of programs [13]. Thus, we cannot ensure that the behavior will not be changed while executing a script. Soares et al. [13] proposed an automated approach for testing Java refactoring engines based on program generation. It will help us to obtain more complete and more proper preconditions. Lahiri et al. [14] proposed a semantic differencing tool to help the developers understand the impact of changes faster. It is very useful for verifying behavior changes in a narrow scope such as inner functions. However, our focus is on helping programmers to perform refactoring more easily. After refactoring, a comprehensive test to check the behavior of a program is still essential.

V. RELATED WORK

Vakilian et al. [9] proposed Compositional Paradigm to implement composite refactoring by setting the operations in detail. Vakilian et al. [10] also reported that setting the dialog may disrupt coding workflow, bring about more overhead costs and decrease productivity. Mens et al. [11] reported that even if options can be finely set, extensions and settings to match the domain of the object are insufficient in current tools. RefactoringScript script was realized as an easy composite refactoring approach with minimized action parameters and simplified search methods.

Li et al. [15][16] proposed a DSL for scripting refactoring in Erlang. Similarly, it uses a script that contains refactoring processes and locations to perform composite refactoring. It also executes scripted refactoring operations and checks preconditions successively like RefactoringScript. However, their refactoring locations scripting uses template-based pattern matching. As a result, it is difficult to apply to Java or other programming languages. Generally, pure object-oriented programming languages like Java can be represented in a tree structure intuitively with containment relationships between packages or classes, fields, and methods. Therefore, we believe it is more concise to use an interface searching a tree structure from the top down, such as XPath [17] and DOM selector of JQuery [18], in Java and many other languages. The search APIs of RefactoringScript are modeled after this.

VI. CONCLUSION AND FUTURE WORK

We have proposed a RefactoringScript language and its processor to script refactoring processes and apply them to appropriate places. By implementing the CE searching API and Scala, we realized a user-friendly scripting language. This tool should significantly reduce the cost of applying refactoring to many places or repeatedly applying refactorings across projects. We are currently working on improving RefactoringScript and developing a refactoring library, which should cover the most popular refactoring processes, based on this tool. Furthermore, although the refactoring types supported by RefactoringScript are limited to the refactoring features provided in Eclipse, we intend to expand it and realize more flexible code deformation. Besides, we will also design a new experiment excluding the learning time to evaluate our tool more accurately.

REFERENCES