JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 36, 513-533 (2020)
DOI: 10.6688/JISE.202005_36(3).0003

Detecting Duplicate Actions in a KDT Script Based on LRS and LCS

CHIEN-HUNG LIU, WOEI-KAE CHEN AND CHEN-YAN LIAO
Department of Computer Science and Information Engineering
National Taipei University of Technology
Taipei, 106 Taiwan
E-mail: {cliu; wkchen; t104598007}@ntut.edu.tw

In keyword-driven testing (KDT), having duplicate actions in the test script is perhaps the most common bad smell. Once the target user interface is changed, a KDT script with duplicate actions can be difficult to maintain. Thus, detecting and removing duplicate actions is an important task. However, so far, no KDT testing tools support automated duplicate detection. This paper proposes a method and tool, called DDT (Duplicate script Detection Tool), for the tester to quickly identify duplicate actions. Two detection algorithms based on Longest Repeated Substring (LRS) and Longest Common Subsequence (LCS) are presented. In addition, DDT provides a keyword extraction feature that can automatically remove duplicate actions. Our experimental results show that there are 21-42% of duplicate actions in a typical KDT script, DDT can detect these duplicate actions in 3-5 seconds, and up to 58-81% of these duplicate actions should be refactored.

Keywords: duplicate detection, keyword extraction, bad smell, KDT scripts, duplicate actions

1. INTRODUCTION

Duplicate code (*i.e.*, clones) means that the same code structure (*i.e.*, a sequence of source code) occurs in more than one places within a program maintained by the same entity [1]. It has been considered the most pervasive bad smell in software programs. In particular, duplicate code not only increases the code size, but also reduces the readability and maintainability of the code. Moreover, it also implies design problems, such as lacking of procedural abstraction. Thus, detecting and removing duplicate code are very important for software development and maintenance.

Similarly, duplicate actions can also occur within a keyword-driven testing (KDT) script, which uses keywords (or action words) to represent a functionality to be tested. A KDT script is composed of a sequence of keywords with associated parameters, corresponding to a sequence of user actions interacting with the application under test. Specifically, each keyword can be considered an abstraction of a sequence of user actions that can be executed automatically by a KDT tool, such as Robot framework [2]. This allows for the separation of test case design and test execution, and enables testers with no programming skills to develop KDT scripts.

When a KDT script contains duplicate actions, the script becomes difficult to read and maintain. Once the target user interface is changed, each of the duplicate actions related to the change must be modified at the same time. This is not only inefficient but also error prone. For example, it is essential to develop two different test cases to verify

Received September 17, 2018; revised March 2, 2019; accepted July 4, 2019.
Communicated by Hung-Yu Kao.

the behavior of an application with two different types of users, namely administrators and regular users. The login actions (entering username and password and clicking on the "login" button) are however the same for both types of users. Thus, the two different test cases may contain exactly the same sequence of login actions. If such duplicate actions can be detected and extracted into a higher level keyword (*e.g.*, a user keyword called login), the understandability and maintainability of the KDT script can both be greatly improved.

Although there exist many clone detection studies and tools for software programs [3-6], so far no KDT testing tools support duplicate detection, and no research related to the detection of duplicate actions has been reported. In particular, it is not even known whether typical KDT scripts contain duplicate actions. Thus, this paper proposes an approach that can detect the duplicate actions in a KDT test script. Particularly, two algorithms based on LRS (Longest Repeated Substring) and LCS (Longest Common Subsequence) [7, 8] are proposed to detect two different types of clones, and a tool called DDT (Duplicate script Detection Tool) is presented. DDT not only supports the proposed duplicate detection method, but also enables users to remove duplicate actions by extracting the specified actions into a new user keyword automatically.

To evaluate the effectiveness of the proposed method and tool, several experiments have been conducted (Section 5). The experimental results suggest that in a typical KDT script there are approximately 21-42% of duplicate actions. Thus, detecting and removing duplicate actions are indeed crucial and can be beneficial. Further, depending on the size of the KDT test script and the degree of duplicates, the duplicate actions can be efficiently detected by using DDT in 3-5 seconds. In addition, up to 58-81% of the duplicate actions detected by DDT should be removed.

The rest of this paper is organized as follows. Section 2 briefly describes related work. Section 3 proposes the detection algorithms and a measure to evaluate the degree of duplicates. The design and implementation of DDT are described in Section 4. Section 5 reports the results of experiments. The concluding remarks and future work are given in Section 6.

2. RELATED WORK

Duplicate code detection has been thoroughly studied and various methods have been proposed [3-6]. According to the report of Roy *et al.* [5, 6], clone detection techniques can be classified into four categories: textual, lexical, syntactic, and semantic. The textual approach basically uses the raw source code directly in clone detection process. The source code is treated as sequences of lines or strings and different techniques are applied to detect if two code fragments are similar. The lexical approach is also known as token-based approach. It commonly transforms the source code into a sequence of lexical tokens and compares the sequence of tokens to find duplicate subsequences of tokens for clone detection. The syntactic approach mainly converts the source code into a syntax tree and detect clones by using tree-matching techniques or using structural metrics. The semantic approach generally employs static program analysis techniques to obtain more information, such as program dependency graph (PDG), in addition to syntactic data and use the semantic information to detect similar code fragments.

Table 1 shows the taxonomy of the four different types of clones described in [5, 6]. The classification is based on the similarity of program text or program functionality. Types 1-3 are based on the textual similarity and Type 4 is based on functional similarity. Depending on the characteristics of target programs and the types of clones to detect, different kinds of techniques or hybrid methods can be applied.

Table 1. Types of clones.

Category	Description
Type 1	Identical clone (ignoring variations of whitespace, layout and comments)
Type 2	Syntactical clone (ignoring variations in identifiers, literals, types, whitespace, layout and comments)
Type 3	Transformed clone (Type 2 clone with modifications such as changed, added or removed statements)
Type 4	Semantic clone (code with the same computation but different implementation)

Like duplicate code [1], duplicate test code is generally considered undesirable. However, despite much effort has been devoted to clone detection for software programs, there exist very few studies on detecting duplicates for test scripts. V. Deursen *et al.* [9] identified a number of bad smells that can occur specifically in test code including Test Code Duplication. They found that test code may contain undesirable duplication especially in the same JUnit test class. Such duplicates can be removed by using Extract Method. Moreover, Meszaros [10] defined a set of test smells (*i.e.*, anti-patterns), which also included Test Code Duplication, and described the symptoms, impacts, causes and possible solutions of the smells.

Bavota *et al.* [11] conducted an empirical investigation to analyze the diffusion of test smells in 987 JUnit classes of 27 software systems. To detect test smells automatically, a simple rule-based tool was developed. Specifically, the tool used a code clone detection system called CCFinder [3] for detecting duplicate test code. They found that Test Code Duplication was quite diffused and occurred in 23 systems and a total of 345 classes. Their results also showed that Test Code Duplication were more diffused in open source systems than in industrial systems. Further, the results indicated that Test Code Duplication had a strong negative impact on program comprehension and maintenance.

Palomba *et al.* [12] conducted a large-scale empirical study on a set of 110 open source software projects to analyze the characteristics of JUnit test classes automatically generated by EvoSuite [13]. They used the test smell detection tool developed in [11]. The results showed that 83% of the test classes were affected by at least one test smell and test smells were highly diffused. Particularly, they also found that Test Code Duplication occurred frequently in the generated test classes (contained in 33% of the JUnit classes), often co-occurred with Indirect Testing smell, and had strong correlations with system size, such as the size and number of classes.

Lavoie *et al.* [14] presented an experiment on detecting and analyzing the duplicates in test suites written in TTCN-3, a standardized test scripting language for telecommunication systems. The duplicates were first identified by computing the syntactical similarity of script fragments using a tool called CLAN (CLone ANalyzer). Then the duplicates

were detected by computing the LCS on the token types and token images of two similar fragments. The experimental results showed that around 24% of script fragments were duplicated. Moreover, the distributions of clones were 82.9%, 15.3%, and 1.8% for Type 1, Type 2, and Type 3 clones respectively, which was statistically significant.

Suan [15] proposed an approach to detect duplicates in BDD (Behavior-Driven Development) specifications, a simple domain-specific language for specifying system behavior in terms of user stories [16]. In their approach, duplicates were identified by using text similarity matching based on a set of rules. Specifically, the set of rules can detect whether two or more entities (*e.g.*, feature, scenario, and steps) are syntactically or semantically equivalent. An Eclipse IDE plugin called SEED was developed to detect and mark the duplicates. The experimental results showed that SEED can discover the duplicates that went undetected by human experts.

Binamungu *et al.* [17] conducted an industry survey to explore the use of BDD, the benefits and challenges of using BDD, and specially the challenges of maintaining BDD specifications. The results indicated that duplicates can make BDD specifications difficult to understand and extend, and can also reduce execution performance. Particularly, the results also showed that most BDD practitioners still performed duplicate detection manually. Thus, they identified that duplicate detection tools and techniques are important research opportunities in the context of BDD.

Although extensive studies have been reported in the literature for detecting duplicates of software programs, test code, and BDD specifications, there is a lack of investigation aimed at detecting the duplicates of KDT scripts. One of the related researches is our previous work [18], which identified five different kinds of smells in a KDT script, including unsuitable naming, duplicate actions, long keyword, long parameter list, and shotgun surgery. This paper focuses on the detection of duplicate actions and provides the support of keyword extraction.

Note that the structure of a KDT script is quite different from that of a typical software program, test code, or BDD specification. For example, variables, branches, and loops are pervasive in a program. However, a typical KDT script does not use a lot of variables, branches, or loops, and contains mostly only sequences of actions. Thus, its structure is a lot less complicated than a program, and detecting Type 4 duplicate actions becomes unnecessary. Since detecting Type 1 duplicate actions are trivial (can be done by exact textual matching), for the clone detection of KDT scripts, we report two algorithms that can detect Type 2 and Type 3 duplicate actions.

3. DUPLICATE DETECTION

This section presents two algorithms, namely DDT-LRS and DDT-LCS, that can detect duplicate actions in a KDT script. We also define *duplicate percentage* as a measure of duplicates.

3.1 The Proposed Approach

A KDT script is composed of a list of actions along with the parameters passed to the actions (*e.g.*, Fig. 1). Each action calls a keyword (a keyword can be either a library

keyword defined by the KDT tool, or a user keyword defined by the tester). When a sequence of actions S is structurally identical (or similar) to another sequence S', we say that S' is a *duplicate* of S (and vice versa) and the set $\{S, S'\}$ is a *duplicate group*. In general, a duplicate group may contain two or more duplicates that are possible to be replaced with a new keyword, which performs the same actions. Since keyword parameters can be substituted (or changed), two pieces of scripts form a duplicate group as long as they have the same action sequences. For example, in Fig. 1, lines 3-4 and 7-8 are duplicates. Though, the parameters of the two type actions are not the same, we can create a new keyword called typeAndClick (with an email address parameter) to perform both type and click actions, and thus the new typeAndClick keyword can replace both lines 3-4 and 7-8.

```
01 ….
02 // piece #1
03 type “email” “a@b.c”
04 click “enter”
05 ….
06 // piece #2
07 type “email” “d@e.f”
08 click “enter”
09 …
```

Fig. 1. Two pieces of script that are duplicates.

In other words, when detecting duplicates, we do not need to consider whether their parameters match exactly. Therefore, the problem is reduced to detecting whether there exist two or more pieces of scripts that use the same sequence of actions. More precisely, we can ignore parameters and think of a script s as a very long string, composed of actions only. The duplicate detection is thus transformed into the problem of identifying whether there exist repeated substrings in s. Each set of repeated substrings represents a duplicate group.

Two of the most famous algorithms that can find the longest repeated or common substrings are LRS and LCS. LRS and LCS can be used to detect Type 2 and Type 3 duplicates, respectively. We extend both LRS and LCS algorithms to perform duplicate detection for KDT scripts.

3.2 Longest Repeated Substring (LRS)

Fig. 2 presents the DDT-LRS algorithm, which uses LRS to perform Type 2 duplicate detection. The algorithm first translates the input script s into s′ to reduce the input size (line 5). Suppose s uses n actions (keywords), and each keyword has m characters on average. The input size is $n \times m$. In practice, a keyword can be a lengthy string (*e.g.*, the keyword DoubleClickSuggestListItem has 26 characters). Thus, m becomes a factor that affects the efficiency of detection. Note that the number of distinct keywords is a constant. Thus, we can translate and represent each distinct keyword into a special, short 16-bit Unicode character (we use a hash function to do so) so that the keyword length becomes a constant. Thus, the overall input size is reduced to n.

Line 9 repeatedly uses LRS to find the longest repeated substring in s′ until the length of repeated substring d′ is less than a certain threshold (the threshold is reported in

Section 5). Since d′ can appear in multiple (more than two) places, the algorithm finds each repeated d′ and add it into dGroup (lines 14-19), which stores the entire group of duplicates for d′. Then, dGroup is included into the detection results, duplicateGroups (line 20). The advantage of using LRS to perform duplicate detection is that every member of the same dGroup contains exactly the same sequence of actions. Thus, extracting a new keyword for each dGroup is always doable. Once d′ (*i.e.*, the current LRS) is detected, the next iteration removes d′ from s′ (line 23) and tries to find the next LRS. By repeatedly detecting the remaining LRS, all duplicate groups are identified.

Note that the algorithm should not take two consecutive modules (a module is either a test case or a user keyword) as a single unit for detection. For example, let UK1 (user keyword 1) and UK2 be two consecutive modules in s. Suppose that the last 2 steps of UK1 are AB, and the first 2 steps of UK2 are XY. The algorithm should not consider ABXY as a legal sequence, since both UK1 and UK2 are supposedly executed independently. We resolve this issue by putting an extra token k in the boundary of every module (line 6). Such a token prevents ABXY inside a module from being mistakenly matched with ABkXY across two modules.

```
01 // Input: s is the input script
02 // Output: duplicateGroups containing all duplicates
03 procedure DDT-LRS(s)
04 begin
05     s′ = replace each keyword used in s with a hash code char;
06     Add a special token k in the boundary of every module in s′;
07     duplicateGroups = ϕ;
08     while (true)
09         d′ = LRS(s′);
10         if (d'.length <= MIN_DUPLICATE_LENGTH_THRESHOLD)
11             break;
12         end if
13         // Locate and store a group of duplicates
14         d = the corresponding source code of d′
15         dGroup = ϕ;
16         for each repeated d in s
17             l = the location of d in s;
18             dGroup.add(<d, l>);
19         end for
20         duplicateGroups.add(dGroup)
21         // Remove d′ from s′ so that the next iteration
22         //      finds the next LRS
23         s′= remove all repeated d′ from s′;
24     end while
25     return duplicateGroups;
26 end
```

Fig. 2. DDT-LRS algorithm.

To illustrate DDT-LRS, Fig. 3 shows an example script s that has two test cases TC-01 and TC-02, where TC-01 contains a sequence of 13 actions that is exactly same as that of TC-02 (the duplicates are marked as red). To detect duplicate actions, DDT-LRS will first generate a string s′ by replacing each keyword in s with a hash code char (line 5). It then

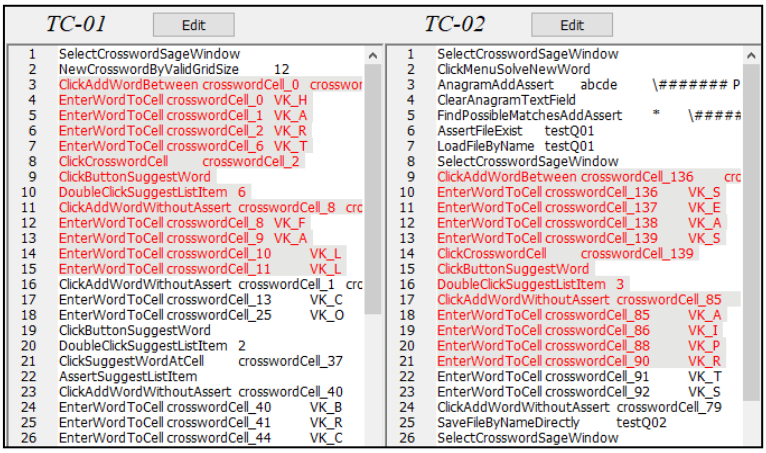


Fig. 3. An example of using DDT-LRS.

detects the longest duplicate in s′ using LRS (line 9). In this case, since both TC-01 and TC-02 are in s′, the hash-coded 13 actions (assuming they are the longest) will be detect ed and stored in d′ (line 9). The source code of the 13 actions, d, is obtained by reversing d′ (line 14) and is used to find all the duplicates in s (line 16), along with their positions (line 17). Thus, both the 13 actions of TC-01 and TC-02 are found and added to dGroup (line 18). After that, DDT-LRS removes all appearances of d′ (the hash-coded 13 actions in both TC-01 and TC-02) from s′ (line 23) and continues to detect the next longest duplicate in s′ using LRS iteratively until the length of the detected duplicate is less than or equal to the threshold.

3.3 Longest Common Subsequence (LCS)

Fig. 4 presents the DDT-LCS algorithm, which uses LCS to perform Type 3 duplicate detection. The algorithm also translates the input script s into s′ to reduce the input size (line 5). However, an LCS is obtained from two different input strings. Thus, the algorithm takes every pair of modules as the inputs of LCS (line 8) and stores the results in d′, which has three attributes lcs, l1, and l2 (explained in lines 10-12). Note that d′.lcs found from the LCS algorithm is a common subsequence, which is generally not a (consecutive) substring of the two inputs strings. Thus, d′.l1, and d′.l2 are used to keep track the positions of characters of the two input strings that make up d′.lcs. All duplicates corresponding to d′.lcs are stored in dGroup (lines 17-27), which is included in the result duplicateGroups (line 28).

For the illustration of DDT-LCS, Fig. 5 shows an example script that has a user keyword TestSloveAnagram and a test case TC-02, where both TestSloveAnagram and TC-02 have a common subsequence of 4 actions (marked as red). To detect these 4 actions, after replacing each keyword with a hash code char (line 5), DDT-LCS will use LCS to detect the longest common subsequence between TestSloveAnagram and TC-02 (line 13 inside the loop of line 8). In this case, the hash-coded 4 actions are detected (assuming they are the longest) and stored in d′. By reversing d′ (line 16), the source code of the 4 actions, d, is obtained which contains both the source code of the 4 actions (d′.lcs) and

their positions in TestSloveAnagram and TC-02 (d′.l1 and d′.l2). Then, the two common action sequences and their positions are added to dGroup.

The advantage of using LCS is that the sequence does not need to be consecutive, allowing the detection of Type 3 duplicates. However, since the results are not always consecutive, it is not always possible to extract keywords from the results. Note that both DDT-LRS and DDT-LCS have their own strengths and weaknesses (Table 2). It is up to the user to choose the right kind of methods to use. In Section 5, we will report the effectiveness of using DDT-LRS and DDT-LCS.

```
01 // Input: s is the input script
02 // Output: duplicateGroups containing all duplicates
03 procedure DetectDuplicateLCS(s)
04 begin
05    s′ = replace each keyword in s with a hash code char;
06    duplicateGroups = ϕ;
07    // A module is either a test case or a user keyword
08    for every pair of modules <m1, m2> in s′
09       // d′ is an object that contains the following attributes
10       //       lcs: the LCS string
11       //       l1: locations of each char in LCS for the first input string
12       //       l2: locations of each char in LCS for the second input string
13       d′ = LCS(m1, m2);
14       if (d′.lcs.length >= MIN_DUPLICATE_LENGTH_THRESHOLD)
15          // Locate and store a group of duplicates
16          d = the corresponding source code of d′;
17          dGroup = ϕ;
18          dGroup.add(<d.lcs, d.l1>);
19          dGroup.add(<d.lcs, d.l2>);
20          for each module m in s
21             if (m != m1 and m != m2)
22                d′′ = LCS(m, d.lcs);
23                if (d′′.lcs == d.lcs)
24                   dGroup.add(<d.lcs, d′′.l1>);
25                end if
26             end if
27          end for
28          duplicateGroups.add(dGroup)
29       end if
30    end for
31 end
```

Fig. 4. DDT-LCS algorithm.

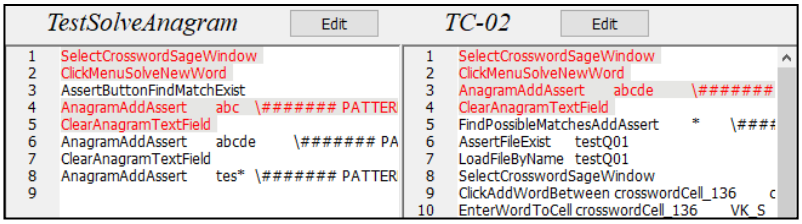


Fig. 5. An example of using DDT-LCS.

Table 2. DDT-LRS vs. DDT-LCS.

Capability	DDT-LRS	DDT-LCS
Can detect non-consecutive actions	No	Yes
Can be directly used in conjunction with extract keyword	Yes	No
Can be used to detect the duplicates inside a module	Yes	No
Detection capability	Type 2	Type 3

3.4 Duplicate Percentage

We use *duplicate percentage* as the measure of evaluating the degree of duplicates a particular test script contains. A test script is in general like the source code of a program, the smaller the better. Thus, we define duplicate percentage as the percentage of space savings that can be achieved when all duplicates are extracted as keywords. Let a script *s* contain some duplicates, and the set *dgs* contain all the duplicate groups of *s*. Let *dg* be a member of *dgs*, *dg* contains $n(dg)$ duplicates, and each duplicate have $m(dg)$ actions (each action is a line of script). Then, *da* (duplicate actions) is defined as:

$$da(dg) = (n(dg) - 1) \times m(dg). \tag{1}$$

The definition of $da(dg)$ corresponds to the number of actions (or lines of script) that can be saved, when the entire *dg* group is replaced by calling a newly created keyword. For example, suppose *dg* has 3 duplicates and each duplicate has 6 actions. The savings is $(3-1) \times 6 = 12$ actions.

Taking all duplicate groups into consideration, the total savings of actions *tda* becomes:

$$tda = \sum_{dg \in dgs} (da(dg)). \tag{2}$$

Thus, we can define *dp* (duplicate percentage) as:

$$dp = \frac{tda}{total\ actions} \tag{3}$$

where *total actions* are the total number of actions (or lines of script) in *s*. Ideally, it would be best for a script to have a $dp = 0$ (*i.e.*, no duplicates).

As an example, suppose a script *s* contains 60 actions in total, and *s* has 2 duplicate groups. The first group has 3 duplicates, each with 6 actions; the second group has 2 duplicates, each with 5 actions. By extracting keywords, the first and the second groups can save $(3-1) \times 6 = 12$ and $(2-1) \times 5 = 5$ actions, respectively. Thus, $dp = (12+5) \div 60 = 28.3\%$. In Section 5, we will study the duplicate percentage for typical test scripts.

4. THE DESIGN AND IMPLEMENTATION OF DDT

Fig. 6 shows the system architecture of DDT. Particularly, DDT extends RIDE (Robot framework IDE) [19] for detecting the duplicate actions of KDT scripts. DDT consists of two modules, the Duplicate Detector and the Extract Keyword Helper.

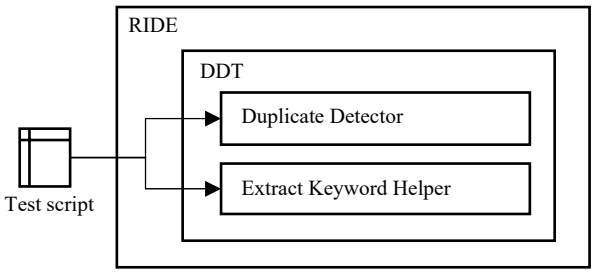


Fig. 6. The system architecture of DDT.

The Duplicate Detector obtains the KDT script from RIDE and is responsible to detect and display the duplicate actions using the proposed algorithms. The Extract Keyword Helper analyzes the duplicate actions selected by users, extracts those actions into a new user keyword, and specifies the values of arguments for the new keyword in the test cases so that the KDT script is refactored into a better structure with less duplicates.

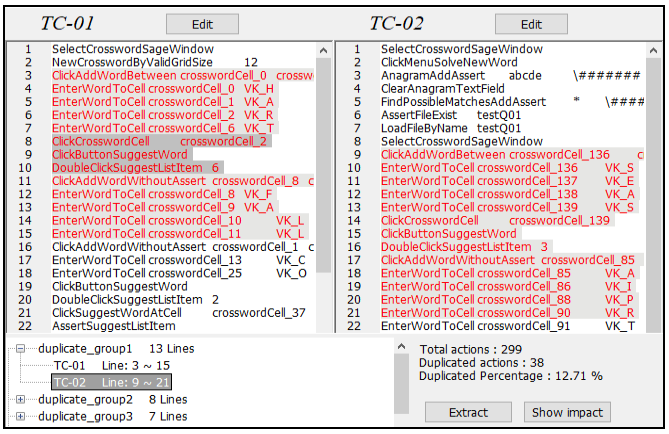


Fig. 7. Selecting the actions for keyword extraction.

Fig. 7 shows a screenshot of using DDT-LRS to detect the duplicates shown in Fig. 3. The detected duplicate groups (ordered in decreasing number of duplicate actions) are shown on the bottom left. Each group contains a list of modules (test cases or user keywords) that contain the detected duplicates. When the user selects a module in the list, the corresponding module is shown and the duplicate actions are highlighted (marked as red). By reviewing these duplicates, the user can determine whether refactoring is necessary. In case that the user modified (or enhanced) some of the modules, the user can request DDT to perform a new detection. DDT will report the most current duplicates. The user interface of DDT provides an option for the user to select the detection algorithm, either DDT-LRS or DDT-LCS. In addition, based on the user's personal preference, he/she can also change the value of MIN_DUPLICATE_LENGTH_THRESHOLD.

To illustrate how a user keyword is extracted automatically with DDT, consider the duplicate actions (marked as red) in the two test cases, TC-01 and TC-02, shown in Fig.

7. Suppose that the user decides to extract three actions (lines 8~10) into a new user keyword. The user first selects these three actions and then clicks on the "Show impact" button in Fig. 7 to review the test cases/user keywords that also have the same duplicate actions. The results are shown in Fig. 8 where three test cases/user keywords have these three actions. The user can click each one of them to review the details of the extraction, including the argument values of each action. The user then clicks the "Extract" button in Fig. 7 and a dialog box will be displayed (Fig. 9). In the dialog box, the user provides the name of the new keyword, say UseSuggestWord, and chooses the test cases/user keywords to be extracted (or simply Extract all). Once the user presses the OK button, DDT will extract these three actions as a new user keyword and refactor the KDT script automatically (Fig. 10). Fig. 11 shows the setting of UseSuggestWord keyword where the arguments are generated by DDT automatically.

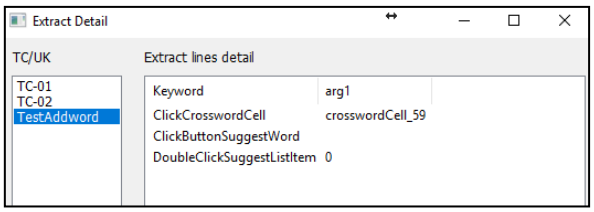


Fig. 8. The test cases (TCs) or user keywords (UKs) that contain the selected actions.

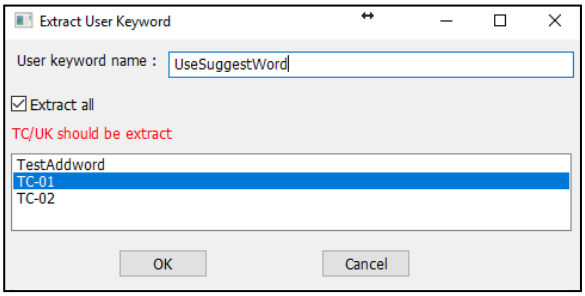
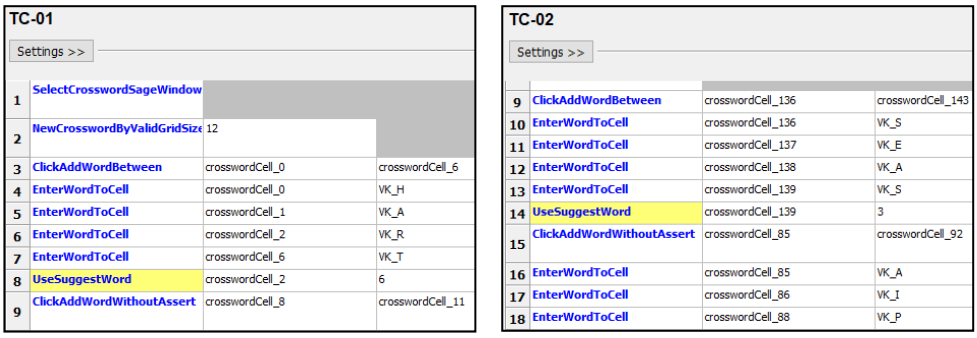


Fig. 9. Extracting a new user keyword.



(a) Refactored TC-01. (b) Refactored TC-02.

Fig. 10. The refactored test cases of Fig. 7 after extracting the keyword by DDT.

Note that RIDE allows users to select and extract a sequence of actions from a test case into a new user keyword. Nevertheless, the users have to manually identify and define the arguments (*i.e.*, parameters) of the new keyword, and also specify the argument values for the caller in the test case. In addition, given a group of n duplicates, RIDE can only extract the first duplicate, not the rest of the $n-1$ duplicates. Thus, refactoring duplicates usually requires non-trivial efforts and can be time-consuming and error-prone. To reduce such efforts, DDT integrates duplicate detection and keyword extraction together, and also automates the process of keyword extraction. The automation includes computing the needed arguments for the new keyword extracted from two (or more) duplicates, setting the arguments for each action within the new keyword, and specifying the values of arguments for the callers of the extracted keywords in each refactored test case.

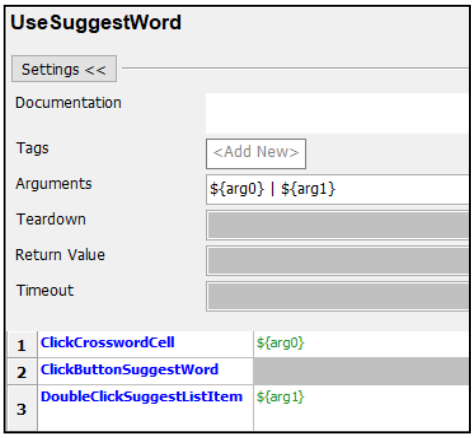


Fig. 11. The setting of extracted keyword generated by DDT.

The arguments of a keyword extracted from any two duplicates x and y can be obtained by examining each parameter of every action in x and y. Suppose that x_{ij} and y_{ij} are the jth parameter of the ith action in x and y respectively, where $0 \le i \le m$ and $0 \le j \le n$. If both x_{ij} and y_{ij} are variable-type parameters, then the extracted keyword will require a variable-type argument to represent x_{ij} and y_{ij}. If both x_{ij} and y_{ij} are value-type parameters and $x_{ij} \neq y_{ij}$, the extracted keyword will also require a variable-type argument to represent x_{ij} and y_{ij}. If, however, both x_{ij} and y_{ij} are value-type parameters and $x_{ij} = y_{ij}$, the extracted keyword can simply use x_{ij} (or y_{ij}) in its corresponding actions and does not need an argument. If x_{ij} is a variable-type parameter and y_{ij} is a value-type parameter, or vice versa, the extracted keyword will need a variable-type argument in order to represent the variable-type parameter x_{ij} (or y_{ij}).

For illustration, consider the test cases TC1 and TC2 that have four duplicate actions shown in Fig. 12. Suppose that the user decides to extract these duplicates into a new user keyword called `ExtractedUK`. Let `A1`…`A4` be the duplicate actions in TC1 and TC2, respectively. Since both `A1` actions in TC1 and TC2 have a value-type parameter with the same value 1, `ExtractedUK` can simply use this value as the parameter of `A1` and does not need an argument. On the contrary, both `A2` actions in TC1 and TC2 have a

value-type parameter with different values 2 and 3, respectively. Thus, a variable-type argument, say *${arg0}*, is needed for A2 in ExtractedUK. The parameters of A3 in TC1 and TC2 are *${a}* and *${b}*, respectively. Both parameters are variable-types and, hence, a variable-type argument, say *${arg1}*, is therefore required in ExtractedUK. Finally, A4 in TC1 has a value-type parameter with a value 4 and A4 in TC2 has a variable-type parameter *${c}*. Thus, a variable-type argument, say *${arg2}*, is needed in ExtractedUK.

Fig. 13 shows the refactored TC1 and TC2 after extracting ExtractedUK. The values of *${arg0}* are 2 and 3 for A2 in TC1 and TC2, respectively. The values of *${arg1}* are *${a}* and {*$b}* for A3 in TC1 and TC2, respectively. Similarly, the values of *${arg2}* are 4 and {*$c}* for A4 in TC1 and TC2, respectively.

TC1		TC2	
…		…	
Action1	1	Action1	1
Action2	2	Action2	3
Action3	*${a}*	Action3	*${b}*
Action4	4	Action4	*${c}*
…		…	

Fig. 12. An example of two duplicate KDT test cases.

TC1	TC2	ExtractedUK	
…	…	Action1	1
ExtractedUK 2 *${a}* 4	ExtractedUK 3 *${b}* *${c}*	Action2	*${arg0}*
…	…	Action3	*${arg1}*
		Action4	*${arg2}*

Fig. 13. The refactored test cases and the setting of extracted keyword.

5. EVALUATION

We conduct experiments to study whether DDT is useful. The following four research questions are addressed:

RQ1 Does a typical KDT script contain duplicate actions? If yes, what is the duplicate percentage?
RQ2 Is using DDT more efficient than finding duplicate actions manually?
RQ3 Should the duplicate actions detected by DDT-LRS and DDT-LCS be extracted as keywords?
RQ4 Is the extract keyword feature of DDT more efficient than that of RIDE?

We select three open-source applications, ezScrum (EzS) [20], Cloud Testing Platform (CTP) [21], and Crossword Sage (CS) [22], as the subjects of our study. EzS is a web-based project management tool (Fig. 14) supporting the agile software process Scrum. CTP is a web-based testing platform supporting Android app compatibility testing in the cloud. Both EzS and CTP have a continuous integration system that performs automated acceptance testing by running a KDT script developed in Robot framework. These KDT scripts perform comprehensive testing for the user interface of EzS and CTP,

and have 3820 and 2348 actions, respectively. Thus, they are suitable targets for our study on the detection of duplicate actions.

CS is a Java-based rich client application for crossword puzzle creation/resolution, which has been studied by many GUI testing researches [18, 23-26]. However, CS does not have a built-in KDT script in its code base. We take the KDT scripts that were developed for CS previously [23]. As described in [23], five different KDT scripts (called CS1-CS5) were developed by five different testers, following the same test plan which contains 9 different test cases and executes a total of 468 actions at run time. CS1-CS5 are good candidates for the study of duplicates, because they perform exactly the same actions and thus can be directly compared.

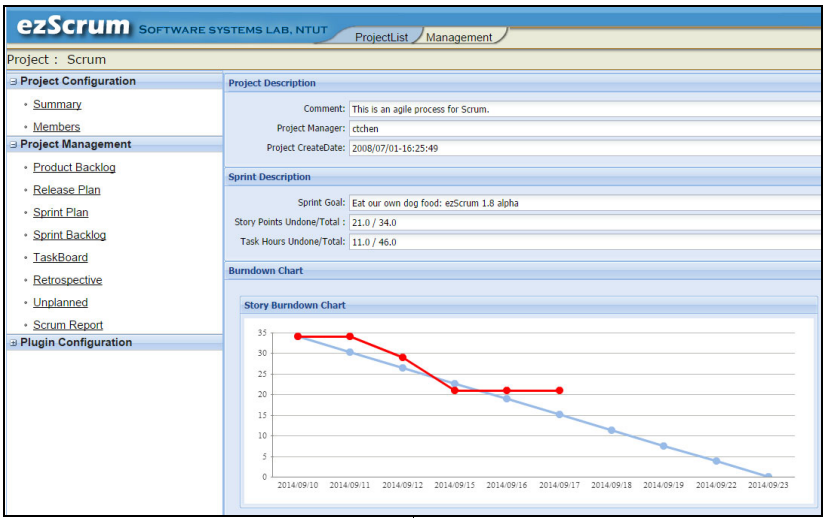


Fig. 14. A screenshot of ezScrum.

5.1 Experiment I

The first experiment addresses RQ1. For the evaluation of duplicate percentage, we use DDT-LRS to detect duplicates. Such duplicates can always be refactored into keywords and thus reflect the range of improvements that can be achieved. Note that, in this experiment, we do not report the results of DDT-LCS. This is because the Type 3 duplicates detected by DDT-LCS are not always extractable (explained in Section 3.3) and thus reporting the results of DDT-LCS can give an overestimated duplicate percentage.

When the threshold is 3 (*i.e.*, MIN_DUPLICATE_LENGTH_THRESHOLD is 3 and all duplicates of 3 or more actions are reported as duplicates), the results are shown in Table 3. The script of CTP (called simply CTP hereafter) had 145 groups of duplicates, a total of 1357 duplicate actions, and a duplicate percentage of 42.1%. In other words, if all the duplicates are refactored, CTP could be 42.1% smaller. Note that, although CS1-CS5 all performed exactly the same sequence of actions at run time, they had different implementations which gave different degrees of duplicates. While CS4 had a duplicate percentage of 39.3%, the duplicate percentage of CS2 was 21.8%, indicating that CS2 was better than CS4 in terms of having less duplicates.

Table 3. The duplicate percentage of CS1-CS5.

Script	Total actions	Duplicate groups (|*dgs*|)	Duplicate actions (*tda*)	Duplicate percentage (*dp*)
EzS	3820	145	1357	35.5%
CTP	2348	100	988	42.1%
CS1	299	15	72	24.1%
CS2	294	12	64	21.8%
CS3	402	20	135	33.6%
CS4	468	21	184	39.3%
CS5	315	15	79	25.1%

As the threshold was increased, the duplicate percentage dropped significantly (Fig. 15). This was because short duplicates were no longer reported as duplicates. When the threshold is 12, the duplicate percentage was around 4-13%. Note that one can perform quite a non-trivial task with 12 consecutive GUI actions. Thus, having duplicates of such a length does not make sense at all. We will study whether such actions should indeed be refactored later in RQ3. Note that when the threshold is set too small (*e.g.*, 1 or 2), the gain of refactoring a detected duplicate group does not necessarily out weight the overhead of maintaining an extra keyword. Thus, we use 3 as our default threshold. Such a threshold reports short duplicates by default. The user can however change the default threshold (by using the user interface of DDT) based on his/her personal preference. For example, if the user would like to consider refactoring only duplicates of greater than or equal to 10 actions, the user can simply set the threshold as 10.

Overall, the answer to RQ1 is "yes, typical KDT scripts such as EzS, CTP, and CS1-CS5 did contain a lot of duplicate actions and the duplicate percentage was around 21.8-42.1%."

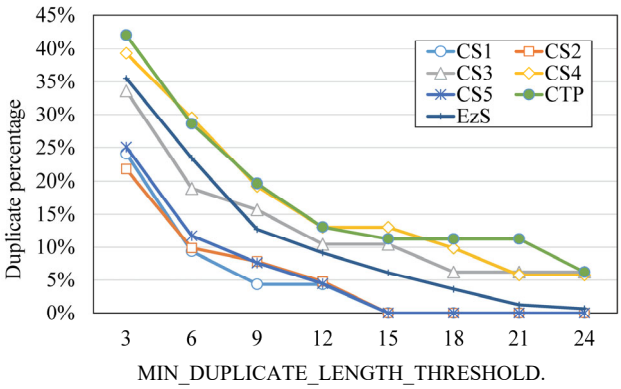


Fig. 15. The duplicate percentages corresponding to different values of MIN_DUPLICATE_LENGTH_THRESHOLD.

5.2 Experiment II

The second experiment addresses RQ2. We evaluate the time required for a human tester to find the duplicates in a test script manually. We invite 10 participants (graduate students of our department), called P1-P10, to serve as testers. We instruct the partici-

pants so that they become familiar with the operation of RIDE. We then request the participants to use RIDE to identify the longest (Type 2) duplicates from CS1. CS1 has 16 user keywords and contains a total of 299 actions (or lines of script). The longest duplicate has 13 actions. In addition to RIDE, the participants are also allowed to use any other tools that can facilitate finding duplicates (*e.g.*, copy the text script into a text editor that supports text search or text comparison). We record the time the participants take. After that, we instruct the participants of the use of DDT-LRS. We then request the participants to use DDT-LRS to find duplicates and record the time the participants spent finding them. As this experiment concerns only with the required detection efforts, the participants are not requested to refactor the longest duplicate.

Table 4. The time needed to find the longest duplicate (unit mm:ss).

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	Avg.
Manual	21:34	08:55	04:23	06:12	15:06	14:00	08:57	06:28	18:34	26:26	14:04
DDT-LRS	00:04	00:03	00:03	00:03	00:03	00:05	00:05	00:04	00:07	00:05	00:04

The results are shown in Table 4. On average, it took a participant 14 minutes (14: 04) to find the longest duplicate manually. On the other hand, by using DDT-LRS, only 4 seconds were needed (finding the longest as well as the rest of the duplicate groups). The huge difference points out that having a tool like DDT can help a tester when he/she needs to maintain a test script in the long run. After all, finding a single duplicate manually is already time-consuming, let alone finding all the rest of the duplicates. Therefore, when eliminating duplicates is desirable, with DDT, the tester can save a lot of time on identifying and locating duplicates. Note that, we did not request the participants to find the longest Type 3 duplicate, because finding such a duplicate by hand would definitely require much more time than that reported in Table 4. Since the gap between manual and DDT-LRS was already huge, further widening the gap was unnecessary.

Overall, the answer to RQ2 is "yes, using DDT was much more efficient than finding duplicates manually."

5.3 Experiment III

The third experiment addresses RQ3. Note that having duplicate actions in a script is a smell, indicating a potential maintenance problem. It does not, however, imply that all duplicate actions should be refactored into keywords. This experiment attempts to answer whether the duplicate actions detected by DDT-LRS and DDT-LCS should be removed.

We choose CS1 and CS4 as the target of our study. The two scripts perform exactly the same actions and represent two extremes, one with a higher and the other with a lower duplicate percentage. Intuitively, a bad script (one with a high duplicate percentage) should have a lot of room for improvement. But, what about a good script (CS1)? Can it be further improved? Our choice of studying CS1 and CS4 allows us to directly compare the detection results of the two scripts, and study the change of detection precision under different levels of duplicate percentage.

We invite 10 participants (the same participants described in the previous experi-

ment) to study the longest 10 duplicate groups (called DG1-DG10) detected by DDT-LRS. The participants are requested to answer whether DG1-DG10 (or a part of DG1-DG10) should be refactored by extracting a new keyword. A yes answer must satisfy the condition $C_{EXTRACT}$, where $C_{EXTRACT}$ is defined as "*the duplicate actions represent a meaningful user operation and are worthy of being extracted as a new keyword so that the new keyword can be reused later and the resulting test script becomes easier to read*."

The results of CS4 are shown in Table 5. The participants did not always give the same answer to the same duplicate group. This is reasonable because while one may consider that extracting a new keyword for a certain duplicate group gives a better overall readability and/or reusability, the other may not. In other words, the decision of whether to fix a particularly smell is personal. Indeed, this is also the reason that a smell is called a smell (a potential problem), rather than a problem. In Table 5, while P1 considered DG5 should be refactored, P4 did not; while all participants considered DG2 should be refactored, only 60% of the participants considered DG5 should be refactored. Considering that not everyone gives the same answer and there is not a definite right or wrong to the answer, we choose to take the average as the indication of whether DG1-DG10 should be refactored. In this way, the overall precision of DDT-LRS was 81%. The results indicate that DDT-LRS was highly reliable in detecting duplicate actions for CS4.

The story of DDT-LCS is however quite different. DDT-LCS can detect Type 3 duplicates. For example, though the two sequences of actions, ABXC and AYBC, are not exact the same, their common subsequences ABC are considered duplicates. Thus, we are most interested in whether the Type 3 duplicates detected by DDT-LCS should be extracted as keywords. We request the participants to study the longest 10 duplicate groups detected by DDT-LCS, and then answer whether each of the groups should be refactored. A yes answer must satisfy the condition $C_{EXTRACT}$ described previously and additionally the newly extracted keyword must contain a part of the non-consecutive actions (*e.g.*, the new keyword contains the actions BC from the duplicate actions ABXC). The additional condition stresses the advantage of DDT-LCS over DDT-LRS (without this condition, DDT-LCS is not any more useful than DDT-LRS).

Table 5. The precision of the 10 longest duplicates of CS4 detected by DDT-LRS.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	
DG1	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	90%
DG2	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	100%
DG3	N	Y	Y	Y	Y	Y	Y	N	Y	Y	80%
DG4	Y	Y	Y	Y	Y	Y	Y	N	N	Y	80%
DG5	Y	Y	Y	N	Y	N	Y	N	N	Y	60%
DG6	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	90%
DG7	Y	Y	Y	N	N	Y	Y	Y	Y	Y	80%
DG8	N	Y	Y	N	Y	Y	Y	N	Y	Y	70%
DG9	Y	Y	Y	N	Y	Y	Y	N	Y	Y	80%
DG10	Y	Y	Y	N	N	Y	Y	Y	Y	Y	80%
	80%	100%	100%	40%	80%	90%	100%	40%	80%	100%	81%

Table 6. The precision of DDT-LRS and DDT-LCS.

	CS1	CS4
DDT-LRS	58%	81%
DDT-LCS	10%	23%

The results are shown in Table 6. For CS4, DDT-LCS had a precision of 23% (*i.e.*, on average, the participants considered 23% of the longest 10 duplicate groups detected by DDT-LCS should be extracted as keywords). The results indicate that DDT-LCS was able to find Type 3 duplicate actions, which was not possible with DDT-LRS. However, DDT-LCS also reported a high percentage of false positives (77%).

A deeper analysis revealed that the reason behind these false positives was related to the nature of the test script. As an example, suppose ABXC and AYBC are Type 3 duplicate actions. Note that ABXC is a sequence of user interface actions where X directly follows B. That is, X is likely strongly dependent on the fact that B is being executed first, and thus it is likely that ABXC cannot be reordered into some other sequence like ABCX. In other words, it is unlikely that Type 3 duplicates such as ABXC and AYBC can be extracted into a new keyword ABC. Therefore, Type 3 duplicate actions contained a lot of false positives.

For CS1, the results of both DDT-LRS and DDT-LCS are shown in Table 6. Although, CS1 had less duplicates (*i.e.*, less room for improvements), DDT-LRS and DDT-LCS were able to find duplicate actions that should be refactored. The results also reflected that the precisions were not as high as those of CS4. Overall speaking, both DDT-LRS and DDT-LCS were able to detect duplicate actions that should be refactored. If the tester does not care about false positives and would like to thoroughly examine the duplicate actions of a test script, DDT-LCS can be used. On the other hand, if the tester desires a higher precision of duplicate detection, DDT-LRS can be used. The answer to RQ3 is "yes, 58-81% of duplicate groups detected by DDT-LRS should be extracted as keywords, and 10-23% of duplicate groups detected by DDT-LCS should be extracted as keywords."

5.4 Experiment IV

The fourth experiment addresses RQ4. We compare the extract keyword feature of DDT with that of RIDE. With RIDE, the user can select a sequence of consecutive actions and request RIDE to automatically extract these actions into a new user keyword. This is fine. But, argument extraction is not supported and when refactoring a group of n duplicates, RIDE can only help extract the first duplicate into a keyword. No assistance is offered for the extraction of the rest of the $n-1$ duplicates. In contrast, when a duplicate group is detected, the user can request DDT to automatically extract all duplicates of the same group into a keyword at the same time.

We invite 12 participants (also graduate students of our department) to serve as testers. We evaluate the time needed for a participant to perform keyword extraction with different tools. We select 5 groups of duplicates from CS1. Each duplicate in the groups has 3-5 actions to be extracted as a new keyword. For example, the three actions, ClickCrosswordCell, ClickSuggestWordButton, and DoubleClickSuggestListItem, are extracted into a new keyword called UseSuggestWord. We request the participants to extract all duplicates of the 5 duplicate groups by using 3 different methods: (i) hand, (ii) RIDE extract keyword, and (iii) DDT extract keyword. To have a fair comparison, we divide

the participants into 3 equal-sized groups. The first group performs extraction in the order of (i), (ii), and (iii); the second group in the order of (ii), (iii), and (i); and the third group in the order of (iii), (i), and (ii). The results are shown in Table 7. On average, a participant needed about 36 minutes to extract keywords fully manually. With the help of RIDE, the keyword extraction time was reduced to around 34 minutes. With DDT, the extraction required less than 3 minutes, significantly better than RIDE. Thus, we can conclude that the answer to RQ4 is "the extract keyword feature of DDT is significantly more efficient than that of RIDE – DDT can improve the efficiency of refactoring duplicates by 11 times."

Table 7. The time needed to perform keyword extraction.

	Manual (mm:ss)	RIDE (mm:ss)	DDT (mm:ss)
Average	36:10	34:07	2:52

6. CONCLUSIONS AND FUTURE WORK

This paper proposed an approach to detect duplicate actions in KDT scripts. Two algorithms DDT-LRS and DDT-LCS that detect Type 2 and Type 3 duplicate actions are reported. To evaluate the degree of duplicates in a KDT script, a measure called duplicate percentage has been presented. Moreover, a tool called DDT has been developed to support the proposed approach. In addition, DDT can also automate the keyword extraction for the specified actions among the identified duplicates. The experimental results suggest that there are 21-42% of duplicate actions in a typical KDT script. Further, DDT can detect these duplicate actions in 3-5 seconds and up to 58-81% of these duplicate actions should be refactored.

The current implementation of DDT does not allow the same keyword name to be used in two (or more) different test suites. In the future, we plan to extend DDT so that keywords stored in different test suites can be properly distinguished. We also plan to enhance DDT-LCS for improving the precision of Type 3 duplicate detection for KDT scripts.

ACKNOWLEDGMENT

This research was partially supported by the Ministry of Science and Technology, Taiwan, under contract numbers MOST 107-2221-E-027-032 and MOST 107-2221-E-027-029, which is gratefully acknowledged.

REFERENCES

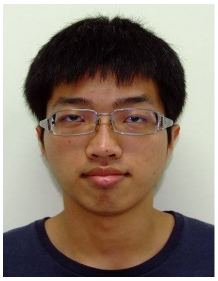
1. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, Boston, 1999.
2. Robot framework, http://robotframework.org/.
3. T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, Vol. 28, 2002, pp. 654-670.

4. S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, Vol. 33, 2007, pp. 577-591.
5. C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Technical Report No. 2007-541, School of Computing, Queen's University at Kingston, Ontario, Canada, 2007.
6. C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, Vol. 74, 2009, pp. 470-495.
7. R. Sedgewick and K. Wayne, *Algorithms*, 4th ed., Addison-Wesley Professional, 2011.
8. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithm*, 3rd ed., MIT Press, 2009.
9. A. van Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, 2001, pp. 92-95.
10. G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison Wesley, NJ, 2007.
11. G. Bavota, A. Qusef, R. Oliveto, A. de Lucia, and D. Binkley, "Are test smells really harmful? An empirical study," *Empirical Software Engineering*, Vol. 20, 2015, pp. 1052-1094.
12. F. Palomba, D. di Nucci, A. Panichella, R. Oliveto, and A. de Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *Proceedings of IEEE/ACM 9th International Workshop on Search-Based Software Testing*, 2016, pp. 5-14.
13. G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 416-419.
14. T. Lavoie, M. Mérineau, E. Merlo, and P. Potvin, "A case study of TTCN-3 test scripts clone analysis in an industrial telecommunication setting," *Information and Software Technology*, Vol. 87, 2017, pp. 32-45.
15. S. W. Suan, "An automated assistant for reducing duplication in living documentation," Master's Thesis, School of Computer Science, University of Manchester, United Kingdom, 2015.
16. Behavior-driven development, https://en.wikipedia.org/wiki/Behavior-driven_development.
17. L. P. Binamungu, S. M. Embury, and N. Konstantinou, "Maintaining behaviour driven development specifications: Challenges and opportunities," in *Proceedings of IEEE 25th International Conference on Software Analysis*, *Evolution and Reengineering*, 2018, pp. 175-184.
18. W.-K. Chen and J. C. Wang, "Bad smells and refactoring methods for GUI test scripts," in *Proceedings of the 13th ACIS International Conference on Software Engineering*, *Artificial Intelligence*, *Networking and Parallel Distributed Computing*, 2012, pp. 289-294.
19. RIDE, https://github.com/robotframework/RIDE.
20. ezScrum, https://github.com/ezScrum, 2018.
21. Cloud Testing Platform (CTP), https://www.openfoundry.org/of/projects/2193, 2018.
22. B. Westgarth, Crossword sage, http://crosswordsage.sourceforge.net, 2018.

23. W.-K. Chen, C.-H. Liu, P.-H. Chen, and Y. Wang, "Is low coupling an important design principle to KDT scripts?" in *Proceedings of the 5th International Conference on Frontier Computing*, LNEE No. 422, 2018, pp. 45-56.
24. A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing," *ACM Transactions on Software Engineering and Methodology*, Vol. 18, 2008.
25. Q. Xie and A. M. Memon, "Using a pilot study to derive a GUI model for automated testing," *ACM Transactions on Software Engineering and Methodology*, Vol. 18, 2008.
26. X. Yuan and A. M. Memon, "Generating event sequence-based test cases using GUI runtime state feedback," *IEEE Transactions on Software Engineering*, Vol. 36, 2010, pp. 81-95.

Chien-Hung Liu (劉建宏) received his Ph.D. degree in Computer Science and Engineering from the University of Texas at Arlington in 2002. He is currently an Associate Professor of Computer Science and Information Engineering Department at National Taipei University of Technology, Taiwan. His research interests include software testing, software engineering, service engineering, and cloud computing.

Woei-Kae Chen (陳偉凱) received M.S. and Ph.D. degrees in Computer Engineering from North Carolina State University in 1988 and 1991, respectively. He is currently a Professor at Department of Computer Science and Information Engineering and the Director of Software Development Research Center of National Taipei University of Technology, Taiwan. His research interests include software testing, software engineering, visual programming, and cloud computing.



Chen-Yan Liao (廖振諺) received B.S. and M.S. degrees in Computer Science and Information Engineering Department from National Taipei University of Technology, Taiwan in 2015 and 2017, respectively. He is currently a Software Engineer of Computer and Network Center in National Taipei University of Technology. His research interests include software engineering and software testing.