www.ietdl.org

Published in IET Software
Received on 27th March 2009
Revised on 20th September 2010
doi: 10.1049/iet-sen.2009.0029

ISSN 1751-8806

Suffix tree-based approach to detecting duplications in sequence diagrams

H. Liu[1,2,3] *Z. Niu*[1] *Z. Ma*[2] *W. Shao*[2]

[1]*School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, People's Republic of China*
[2]*Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, People's Republic of China*
[3]*Beijing Lab of Intelligent Information Technology, School of Computer Science, Beijing Institute of Technology, Beijing 100081, People's Republic of China*
E-mail: liuhui2005@gmail.com

Abstract: Models are core artefacts in software development and maintenance. Consequently, quality of models, especially maintainability and extensibility, becomes a big concern for most non-trivial applications. For some reasons, software models usually contain some duplications. These duplications had better be detected and removed because the duplications may reduce maintainability, extensibility and reusability of models. As an initial attempt to address the issue, the author propose an approach in this study to detecting duplications in sequence diagrams. With special preprocessing, the author convert 2-dimensional (2-D) sequence diagrams into an 1-D array. Then the author construct a suffix tree for the array. With the suffix tree, duplications are detected and reported. To ensure that every duplication detected with the suffix tree can be extracted as a separate reusable sequence diagram, the author revise the traditional construction algorithm of suffix trees by proposing a special algorithm to detect the longest common prefixes of suffixes. The author also probe approaches to removing duplications. The proposed approach has been implemented in DuplicationDetector. With the implementation, the author evaluated the proposed approach on six industrial applications. Evaluation results suggest that the approach is effective in detecting duplications in sequence diagrams. The main contribution of the study is an approach to detecting duplications in sequence diagrams, a prototype implementation and an initial evaluation.

1 Introduction

Models are becoming core artefacts of software development and maintenance. The Unified Modelling Language (UML) [1] was proposed by Object Management Group (OMG), and it was finally adopted by ISO as a standard modelling language (ISO/IEC 19501). Based on the success of UML, OMG brought forward the concept of model driven architecture (MDA) [2]. With the popularity of UML and MDA, system modelling becomes a key activity in software development. In the context of MDA, models are automatically transformed into source code, which is in turn automatically transformed into executable systems. Therefore, developers design models to which maintainers make changes whenever new requirements are added or existing requirements are deleted or changed. In other words, models become the core artefacts in both development and maintenance. As a result, the maintainability, readability and extensibility of models become more and more critical for system development and maintenance. Consequently, any progress on revealing or improving model quality, especially maintainability, readability and reusability, is valuable.

Model duplications reduce the maintainability and reusability of modes, and thus should be detected and removed. Model duplications are identical model fragments. They may reduce the maintainability and reusability of models [3, 4], and thus they are usually considered as bad smells. The most serious problem of model duplications is that once a change is made in one place, you have to make the same change in all of its duplications. It is also possible that the duplications are maintained by different people. As a result, it is very possible to miss some duplications, and thus the system become inconsistent. It is even worse if a bug is found out in a piece of model who has lots of duplications spread through the model. If one or more duplications are missed, bugs are still alive. Consequently, we had better detect and remove duplications to improve the maintainability of models. At least, we should detect and track them if it is hard or impossible to remove them in some special cases.

As an initial attempt to address the problem of model duplications, we propose an approach to detecting duplications in sequence diagrams. Sequence diagrams are intensively used in system modelling to describe behaviours of use cases, operations and collaborations. Sequence diagrams can illustrate the interactions between users and the system. They may also describe the interactions among a group of objects cooperating to fulfill a special function. Besides the detection of duplications in sequence diagrams,

© The Institution of Engineering and Technology 2011

we also probe the approaches to removing the detected duplications.

Duplications are introduced into sequence diagrams [3, 4] for the following reasons:

1. Software systems are usually very complex. To deal with complex systems, we often employ the divide-and-conquer policy. Unfortunately, duplications are produced as a byproduct of the policy.
2. Poor design, especially lack of abstract.
3. Designers' reluctancy to restructure their design for reuse.
4. The special way sequence diagrams are used. For a scenario (or a use case, an algorithm and so on), there is a main execution flow and several alternatives. These flows are similar to each other, and share some common parts [5]. Once each even flow is described by a sequence diagram [6], the common parts among the flows are turned into duplications in sequence diagrams.

Duplications in sequence diagrams would reduce maintainability and reusability of sequence diagrams [3, 4]. Detailed impact is discussed as following:

1. Duplications make it difficult to change existing sequence diagrams. Changes on a piece of a fragment have to be carried out on all duplications of the fragment. If some of them are not changed synchronously, it would cause inconsistency or even insecurity in the resulting systems.
2. Duplications increase the size of the models. It is one of the reasons why software engineers are reluctant to restructure existing models for reuse. Their workload is usually measured by the size of models, instead of the quality of the models.
3. Duplications may increase the workload of implementation. For a large system, hundreds of software engineers cooperate together, and each of them (or each small team) would take charge of a small part of the system. As a result, it is very possible that duplications may be built and implemented by different engineers. In other words, a single functionality is implemented more than once.

OMG has also realised the problem [3], and made a non-trivial revision on sequence diagram meta-model (UML2.0 [1]). What it provides is a mechanism (InteractionUse) to reuse existing fragments. But how to find out duplications is still left to developers and maintainers.

To address the problem, this paper proposes an approach to detecting duplications in sequence diagrams. First, 2-dimensional (2-D) sequence diagrams are converted into an 1-D array. Then based on the array, a suffix tree is constructed. With the suffix tree, duplications are detected and reported. In order to improve the precision of the approach, we also revise the traditional construction algorithm of suffix trees. The new algorithm ensures that every duplication detected with the suffix tree can be extracted as a separate reusable sequence diagram, that is, these duplications can be removed via InteractionUse.

The main ideal of detecting overlapping sequence diagrams has been discussed briefly in [7]. But for space limitation, details of the algorithms was not presented. In this paper, the following extensions have been made:

1. Further evaluations (four cases) have been done after the publication of [7]. Evaluation results are presented and discussed in this paper (in Section 5).
2. A new section is added to discuss the contributions and limitation of the proposed approach (in Section 6).
3. A new section is added on how to deal with duplications in sequence diagram once they are detected. The orders in which duplications should be dealt with are also discussed (in Section 4.2.4).
4. The detailed algorithm to compare CombinedFragments is presented (in Fig. 8).
5. A way to avoid false duplications reported by suffix trees is proposed (in Section 3.2).
6. Complexity analysis of the algorithms is presented (in Section 4).
7. Details of the implementation of the proposed approach is presented (in Section 5).
8. Related works are classified, and a thorough comparison is made between this paper and related works.

The rest of this paper is structured as follows. Section 2 discusses related work. Section 3 introduces sequence diagrams of UML2.0 and suffix trees. Section 4 presents the detection approach. Section 5 presents tool support and evaluation of the proposed approach. Section 6 discusses some issues involved in the proposed approach. Section 7 makes a conclusion.

2 Related work

2.1 Detection of duplications

People have discussed duplicate source code for a long time. The focus is how to detect clones automatically. People have proposed approaches and developed lots of tools to automatically (or semi-automatically) detect duplicate source code. Dup [8] detects line-by-line clones with suffix trees. It detects not only duplications, but also near-duplications because it replaces identifiers of functions, variables and types with a special identifier (parameter identifier). But the line-by-line method depends on the line-structure modification. Tairas and Gray [9] also detect clones based on suffix trees. Another suffix tree-based clone detection algorithm is proposed by Kamiya *et al.* [10]. DupLoc proposed by Ducasse *et al.* [11] is a language dependent clone detection tool. It removes white-spaces and comments, and then detect clones in dynamic pattern matching (DPM) algorithm. CloneDR proposed by Baxter *et al.* [12] detects duplications based on abstract syntax trees (AST) by comparing sub-trees of the AST.

However, the existing detection approaches mainly focus on source code. They do not take models into consideration. One of the possible reasons is that MDA and UML are quite new technologies, which were proposed recently by OMG.

2.2 Software refactoring

Software refactoring is an effective means to improve software quality, especially maintainability, extensibility and reusability. Software refactoring was first proposed by Opdyke in his PhD thesis [13]. Ever since that, people have made great advances in research and application of software refactoring. Software refactoring has been accepted as an important activity in software development and maintenance. In eXtreme Programming (XP) [14] and other agile software development processes, refactoring is one of the core activities [15, 16]. With the popularity of refactoring, more and more CASEs and IDEs, such as Eclipse and Visual Studio, begin to provide support for

© The Institution of Engineering and Technology 2011

doi: 10.1049/iet-sen.2009.0029

software refactoring. Professional refactoring tools, such as Refactoring Brower [17], are also developed.

Fowler *et al.* [18] discuss bad smells in programs, and propose refactoring rules to restructure them. Philipps and Rumpe [19], Tokuda and Batory [20], and Ivkovic and Kontogiannis [21] discuss refactorings of architectures. Russo *et al.* [22] apply refactorings to requirements specifications.

In order to maximise the effect of software refactoring activities, Liu *et al.* [23–25] try to schedule conflicting refactorings. With the schedule, more effective refactorings are carried out first, and more destructive refactoring may be delayed. Evaluations suggest that the schedule of conflicting refactorings helps to improve refactoring effect.

In recent years, the quality of models becomes more and more important. As a response to it, model refactoring [26] is proposed to improve models' quality while preserving models' external behaviours. Sunyé *et al.* [26] introduce refactoring into UML models and explain what is model refactoring. Astels [27] investigates how UML supports model refactoring. Correa and Werner [28] make a step further and apply refactoring technology to OCL (Object Constraint Language) [29]. Ivkovic and Kontogiannis [21] investigate refactorings on software architectures which are described in UML profiles. How can model refactoring be performed without changing the behaviour of models is explained by Straeten *et al.* [30] and Liu *et al.* [31].

A comprehensive survey of software refactoring is given by Mens and Tourwé [32].

2.3 Refactorings on use case diagrams and sequence diagrams

Refactoring on use case diagrams and sequence diagrams is also discussed. Use case diagrams and sequence diagram are intensively used in describing software requirements. Yu *et al.* [33] and Xu *et al.* [34] propose and discuss some refactorings on use cases. Ren *et al.* [4] propose refactorings to remove duplications in sequence diagrams. Both papers [4, 34] mention the refactoring of removing duplicate message sequences in use case episodes (sequence diagrams).

Although refactorings to remove duplicate message sequences have been proposed, how to detect duplicate message sequences is rarely discussed. Model refactoring is usually performed in the following way. Experts define some bad smells first, and then propose refactorings to remove them [18, 32]. Duplication is a typical bad smell, and it deserves refactoring of course. Ren *et al.* [4] have proposed refactorings to remove duplications in sequence diagrams. However, in order to apply model refactorings, bad smells should be found out first. But within our knowledge, how to detect duplication in sequence diagrams is rarely discussed whereas there are rich detection algorithms and tools for the detection of duplicate source code. In [35], we focus on the detection of overlapping use cases. Message sequences detected in [35] fulfill the same goal, but they are not necessarily identical. However, the detection approach proposed in this paper detects duplications in sequence diagrams.

3 Basic sequence diagrams and suffix trees

3.1 Basic sequence diagrams and duplicate fragments

Basic sequence diagrams of UML2.0 [1] are special cases where only basic elements of UML2.0 sequence diagrams, such as lifelines, OccurrenceSpecifications, ExecutionSpecifications and synchronous messages, appear. Other UML2.0 elements, such as Gates, InteractionUses, CombinedFragments and asynchronous messages (concurrency), are considered as advance features, which are discussed in Section 4.3.

Definition 1 (Basic sequence diagram): A basic sequence diagram is a tuple $(L, O, E, M, <, R_{o,l}, R_{o,e}, R_{o,m})$ where L is a set of lifelines, O is a set of OccurrenceSpecifications, E is a set of ExecutionSpecifications and M is a set of messages, $<$ is a total ordering on O, $R_{o,l}$ is a relationship between O and L indicating lifelines covered by OccurrenceSpecifications, $R_{o,e}$ is a relationship between O and E indicating initial and terminal OccurrenceSpecifications of every ExecutionSpecification, $R_{o,m}$ is a relationship between O and M indicating endpoints of every message.

There is a total ordering among OccurrenceSpeciations of a basic sequence diagram because concurrency, branches and loops do not appear in basic sequence diagrams. In order to define the portion of a sequence diagram that can be extracted as a separate reusable interaction diagram, we give the following definitions.

Definition 2 (Fragment of a sequence diagram): A fragment of a sequence diagram is a rectangular area in a sequence diagram whose edges are parallel to the axes of the sequence diagram.

A fragment itself can be considered as a basic sequence diagram in fact. A fragment can be recorded as $(L, O, E, M, <, R_{o,l}, R_{o,e}, R_{o,m})$ as defined in Definition 1.

Definition 3 (Extractable fragment): An extractable fragment is a fragment of a sequence diagram which satisfies the following constraints:

1. No edge of the fragment intersects any Execution Specification.
2. Any OccurrenceSpecification outside the fragment appears above the top or below the bottom of the fragment.

An extractable fragment is a portion that can be extracted as a separate interaction diagram. The first constraint indicates that every ExecutionSpecification contained in an extractable fragment should be complete. In other words, if an extractable fragment contains any part of an ExecutionSpecification, the extractable fragment should contains the initial and terminal points of the ExecutionSpecification. An example is show in Fig. 1. Because the smaller rectangle (in dashed lines) intersects an ExecutionSpecification, the rectangle has to be extended to the very beginning of the cut ExecutionSpecification as indicated by the larger rectangle (in solid lines) to cover the ExecutionSpecification completely.

The second constraint of extractable fragments is presented to ensure that no events outside the extractable fragment are executed concurrently with the fragment.

Definition 4 (Duplicate fragments): Two extractable fragments are duplicate fragments if: (i) They contain the same elements, and (ii) Elements have the same relationships in the two fragments.

In a sequence diagram, the horizontal positions of elements are not important as far as semantics are concerned. What is important is the relative order on the vertical axis. Therefore, the relationship among elements (specially, OccurrenceSpecifications) of a sequence diagram can be

© The Institution of Engineering and Technology 2011

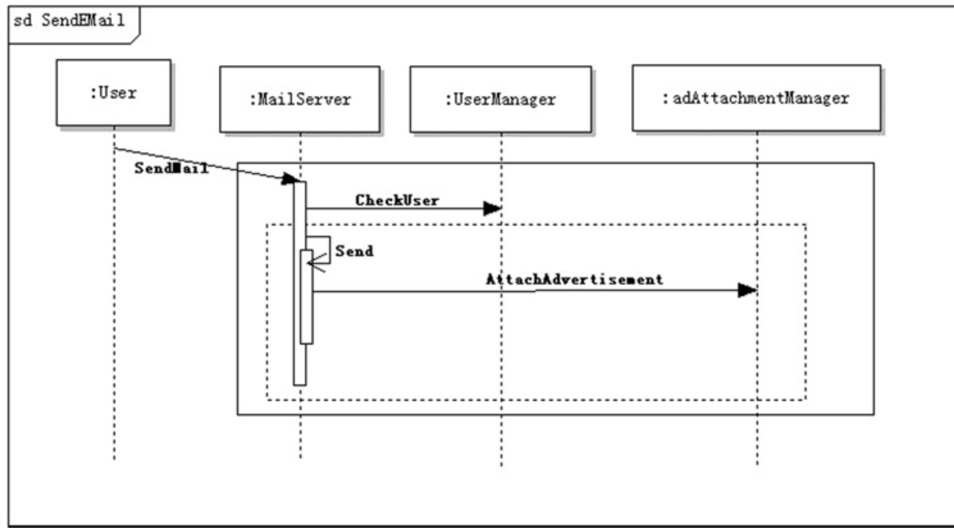

Fig. 1 *Extractable fragments*

simplified as the relative order on the vertical axis. This is the meaning of '$<$' in Definition 1.

Messages entering or leaving duplicate fragments deserve special attention. The initial points of the entering messages are replaced with Gates when the extractable fragments are extracted as separate interactions. The sequence diagram OriginalSD shown in Fig. 2 is a good example. An extractable fragment of OriginalSD is extracted as a separate diagram ExtractedSD, and the original diagram is restructured as NewSD referring ExtractedSD. The initial point of the entering message message1# is replaced with a gate when the extractable fragment is extracted as a separate interaction ExtractedSD. It suggests that the exact initial points of entering messages of extractable fragments are not important, and they can be ignored when duplicate fragments are compared. We can also ignore terminal positions of leaving messages in the same way.

3.2 Suffix trees

A suffix tree is a compacted trie (digital search tree) containing all suffixes of a given array. Suffix trees have been used widely in pattern matching and clone detection [36].

Let S_0 be an array of $(n-1)$ elements from $\sum$, and \$ is an element matching no element of $\sum$. We attach \$ to the end of S_0 and obtain a new array $S = S_0\$$. We denote S as $S := S[1{:}n]$. For any $i \in [1,n]$, we call $S[i{:}n]$ the ith suffix of S. If a compacted trie T contains all the suffixes of S, we call it a suffix tree of S. Fig. 3 is a suffix tree of array $S = \text{ababa\$}$.

A suffix trees T has two properties [36]:

1. *Node existence:* There is a leaf for each suffix of S.
2. *Common prefix:* If two suffixes of S share a prefix, say y, then they must share the path in T leading to the extended locus of y.

These properties tell us how to detect duplicate subarrays: any internal node of T is a duplication shared by all the leaf nodes (suffixes) beneath it. For example, from Fig. 3, we know that substring '*ba*' appears twice in the original string 'ababa\$' because its corresponding node has two leaf nodes in the tree (the second suffix 'baba\$' and the fourth suffix 'ba\$').

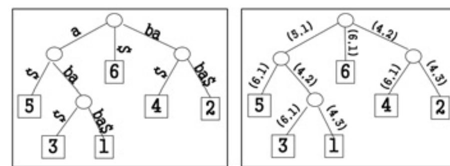

Fig. 3 *Suffix tree and its compact representation*

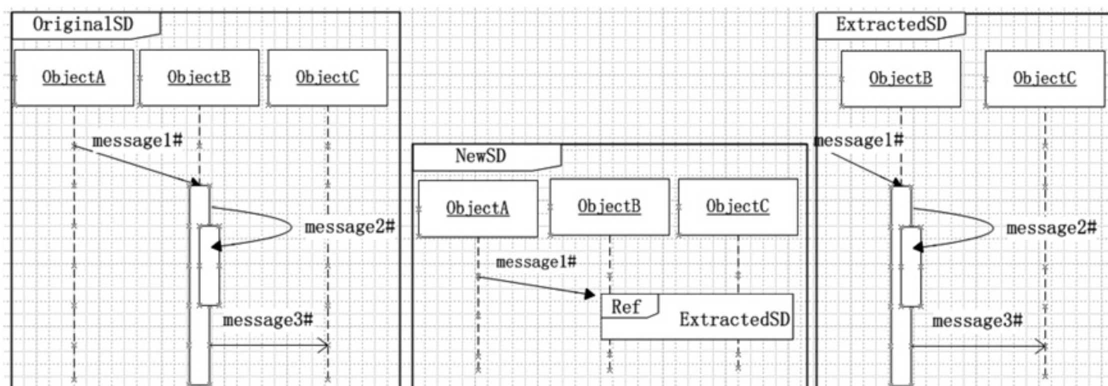

Fig. 2 *Cut messages*

© The Institution of Engineering and Technology 2011
IET Softw., 2011, Vol. 5, Iss. 4, pp. 385–397
doi: 10.1049/iet-sen.2009.0029

However, sometimes it may report false duplications. In Fig. 3, it says that 'aba' has appeared twice (in the firth suffix and the third suffix), but in fact we can not extract two 'aba' from 'ababa$' at the same time. In order to avoid such false duplications, we have to do some further detection before report any duplication: if a subarray Sub has two leaf nodes, ith suffix and jth suffix, then the common prefix ComPrefix (duplicate subarray) of the two suffixes is the first k elements of Sub where

$$k = \min(|j - i|, \text{Sub.Length}) \quad (1)$$

For the above example, the process is shown below

$$\begin{cases} i = 1 \\ j = 3 \\ \text{Sub} = \text{aba} \end{cases} \Rightarrow \begin{cases} k = 2 \\ \text{Com Prefix} = \text{ab} \end{cases}$$

The basic construction algorithm of suffix trees requires $O(n^2)$ time complexity. But advanced algorithms, such as those given by McCreight [37], Chen and Seiferas [38], and Ukkonen [39] require only linear time and space.

The focus of suffix tree-based approaches is the construction of the suffix trees. The core of the construction is an algorithm to detect the longest common prefix of any pair of suffixes. In this paper, we first map sequence diagrams into an array, and then propose an algorithm to detect the longest common prefixes of its suffixes. The proposed algorithm guarantees that duplications found out with the suffix tree are extractable duplicate fragments of the sequence diagrams. In the next section, the proposed approach is discussed in detail.

4 Detection approach

4.1 Overview

A sequence diagram is a planar diagram where every element has a position and a planar shape. We try to arrange all elements of a sequence diagram into an array. Then these arrays (one array for each sequence diagram) are concatenated into a new array for which a suffix tree is built. We revise the traditional construction algorithm of suffix trees by proposing a special algorithm to detect common prefixes of suffixes. Every duplication detected with the algorithm corresponds to an extractable fragment. With the suffix trees, duplications are detected for refactoring. We first propose the detection approach for basic sequence diagrams in Section 4.2, and then the approach is generalised to deal with advance features of UML2.0 sequence diagrams in Section 4.3.

4.2 Basic sequence diagrams

In this subsection, we confine ourselves to basic UML2.0 sequence diagrams.

4.2.1 Mapping basic sequence diagrams into arrays: If we add an attribute Lifeline to OccurrenceSpecifications to present the lifelines they cover, the set of lifelines (L) of a sequence diagram can be collected from its OccurrenceSpecifications (O) as shown in (2).

$$L = \{\ell | \exists o \in O \cdot \ell = o.\text{Lifeline}\} \quad (2)$$

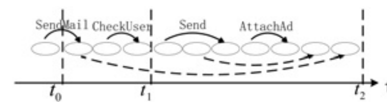

Fig. 4 *CSD of SendMail in Fig. 1*

The set of lifelines gathered from OccurrenceSpecifications is equal to L in Definition 1 because every meaningful lifeline must be covered by one or more OccurrenceSpecifications. If a lifeline is not covered by any OccurrenceSpecification, nothing meaningful happens to the lifeline. As a result, the lifeline can be safely removed without changing the semantics of the enclosing diagram.

With the attachment, a basic sequence diagram can be recorded as (O, E, M, $<$, $R_{o,e}$, $R_{o,m}$) where L and $R_{o,l}$ of Definition 1 are embedded in O.Lifeline. In Section 3.1, we know that there is a total ordering among Occurrencespecifications of a basic sequence diagram which reflects their relative vertical positions. Therefore, with lifelines removed, a 2-D basic sequence diagram can be mapped into a 1-D array by projecting elements on its vertical axis. An example is shown in Fig. 4 which is mapped from the basic sequence diagram shown in Fig. 1. In Fig. 4, circles are OccurrenceSpecifications, dashed directed lines are ExecutionSpecifications and solid directed lines are messages.

For convenience, a compressed sequence diagram (CSD) as shown in Fig. 4 is briefly identified as a CSD.

If messages and ExecutionSpecifications of a CSD are recorded as attributes of OccurrenceSpecifications (such as OccurrenceSpecification.InMsg), a CSD can be treated as an array of OccurrenceSpecifications with attributes from which lifelines, messages and ExecutionSpecifications can be recovered.

4.2.2 Detecting the longest common prefix of suffixes: We gather arrays mapped from a set of sequence diagrams, and then concatenate the arrays into a long array (LA). As discussed in Section 3.2, if we can define an algorithm to detect the longest common prefix of two suffixes of LA, we can construct a suffix tree for LA, and find out duplicate subarrays (fragments) with the suffix tree. In order to guarantee that duplicate subarrays found by the suffix tree would correspond to extractable fragments of the original sequence diagrams, the algorithm should promise that the longest common prefix corresponds to an extractable fragments of the original sequence diagrams.

In the mapping from a sequence diagram into a CSD, we know that the axis (t) of a CSD (as shown in Fig. 4) is the vertical axis of the sequence diagram in fact. Therefore, a section of a CSD corresponds to a rectangular area in the enclosing diagram. As depicted in Fig. 4, the section within [t_1, t_2] corresponds to the smaller rectangle in Fig. 1 whereas that within [t_0, t_2] corresponds to the lager rectangle in Fig. 1. Furthermore, no OccurrenceSpecification outside a region (such as [t_1, t_2]) occurs within the duration of the region because all OccurrenceSpecifications are ordered on the time axis. In other words, every section of a CSD satisfies the second constraint of extractable fragments in Definition 3.

In order to satisfy the first constraint of extractable fragments (extractable fragments should not intersect with any executionSepcification), every OccurrenceSpecification has to be checked to make sure that every ExecutionSpecification beneath it is completely included in the section. If a section

© The Institution of Engineering and Technology 2011

of a CSD corresponds to an extractable fragment, we call it extractable section.

Extractable fragments should not cross the boundaries of sequence diagrams. In order to isolate diagrams from each other, a DiagramEndPoint is inserted into the end of every sequence diagram.

The special algorithm to detect the longest common prefix of two suffixes of LA is presented in Fig. 5. For every OccurrenceSpecification, its attributes Lifeline, InMsg, OutMsg and Exs represent the lifeline it covers, incoming message, outgoing message and the sequence of ExecutionSpecifications beneath it, respectively. For InMsge, we also record the distance from its initial OccurrenceSpecification to its terminal Occurrence Specification (the current point). Fig. 6 presents two sequence diagram fragments, *F1* and *F2*, where numbered circles represent OccurrenceSpecifications. In both fragments, OccurrenceSpecification 1 and 2 send message MessageA whereas OccurrenceSpecification 3 and 4 receive MessageA. Therefore, if attributes InMsg and OutMsg are compared only by the message names, the algorithm in Fig. 5 would report the two fragments as duplicate (in fact, they are not). Comparing InMsg.Distance helps in avoiding such kind of false positives. With the distance, we can also tell whether it is an entering message or internal message by

$$S_m(i).I\text{nMsg.Distance} < i \qquad (3)$$

where $S_m(i)$ is the ith element of the suffix S_m. As discussed in Section 3.1, the distinction is necessary because original positions of incoming messages are not important whereas those of internal message are important. Therefore, InMsg.Distance is compared if and only if InMsg. Distance $< i$ holds.

```
//Input: Suffix S_m, S_n (m > n), MinimalLength
//Output: the length of the longest common prefix of S_m and S_n
For(i=1; i ≤ S_m.Length; i++)
  //Loop to compare every pair of elements
   If (S_m(i).Type ≠ S_n(i).Type)
      Break;
   End If
   If (S_m(i).Type = OccurrenceSpecification)
     If (S_m(i) ≠ S_n(i))
      //by comparing attributes: Lifeline, InMsg, OutMsg and Exs.
       Break;
     End If
     G(i) = Max(G(i − 1), i);//Default value
     For(j=1; i ≤ S_m(i).Exs.Length; j++)
       If (i < S_m(i).Exs(j).SPoint)
        //Some ExecutionSpecification beneath is cut
          Goto Maxj;
       End If
       G(i) = Max(G(i), S_m(i).Exs(j).EPoint + i);
     End For
   End If
   IF (S_m(i).Type = DiagramEndPoint)
    //An extractable fragment should not cross the boundary of a diagram.
     Break;
  End If
End For
Maxj: Return the max j which satisfies:
      (G(j) ≤ j) and (i > j ≥ MinimalLength)
```

Fig. 5 *Algorithm to detect the longest common prefix of two suffixes*

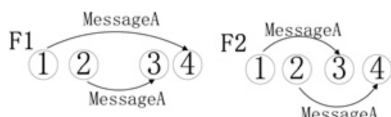

Fig. 6 *Comparison of messages*

For every OccurrenceSpecification, the Execution-Specifications beneath it are recorded as a sequence Exs. We also record the initial and terminal positions of every ExecutionSpecifications by SPoint and EPoint (which are distances to the current OccurrenceSpecification instead of the actual positions). With SPoint, we can determine whether the initial point of an ExecutionSpecification is contained in the prefix section by

$$S_m(i).\text{Exs}(j).S\,\text{Point} < i \qquad (4)$$

Where $S_m(i)$.Exs(j) is the jth ExecutionSpecification of $S_m(i)$. With EPoint, we can obtain the shortest length $G(i)$ of the common prefix, which includes the current OccurrenceSpecification completely: Extractable fragments should contain both initial and terminal points of every ExecutionSpecification involved (the first constraint of extractable fragments in Definition 3)

$$\text{for}\,(j = 1; j \leq S_m(i).\text{Exs.Length}; j++)$$
$$G(i) = \text{Max}(G(i), S_m(i).\text{Exs}(j).E\,\text{Point} + i)$$

However, since suffixes are compared from left to right, comparing the current points, we cannot determine whether the common prefix will be extended to $G(i)$ or not. Therefore, we just record $G(i)$, and go on to compare the following OccurrenceSpecifications until mismatch is encountered. Then, we turn back, and find the max j, which holds the following constraint

$$(G(j) < j) \wedge (i > j \geq \text{Minima Length}) \qquad (5)$$

Constraint 5 ensures that if the common prefix is terminated at position j, the common prefix will contain all ExecutionSpecifications involved.

A DiagramEndPoint is unequal to any other points, even if the point is also a DiagramEndPoint. In this way, duplicate fragments are confined within diagrams.

Since the elements in suffixes S_m and S_n are compared pair by pair, the time complexity of the algorithm is $O(\text{Min}(\text{Length}(S_m),\text{Length}(S_n)))$.

4.2.3 Constructing suffix trees and detecting duplications: With the algorithm to detect the longest common prefix of suffixes, we can construct suffix trees for basic sequence diagrams. Because the common prefixes detected with the algorithm are insured to be extractable fragments, all the internal nodes of the suffix trees are extractable fragments, too. So, once the suffix tree is constructed, duplicate fragments are found in the same way as duplicate source code is found with suffix trees: Every internal node of the suffix tree represents a duplication shared by all the leaf nodes (suffixes) beneath it.

4.2.4 Actions on duplications: Once duplications are detected, designers should decide how to deal with them. Reported duplications happen to be duplicate. In other words, they are currently identical, but they are very likely to evolve in different ways in future. In this case, extracting them as a common message sequence is non-sense.

If designers are sure that the duplications represent the same interaction, and there is little chance for them to

© The Institution of Engineering and Technology 2011

doi: 10.1049/iet-sen.2009.0029

evolve in different ways, they had better remove the duplications. Duplications in sequence diagrams can be removed with InteractionUses [1, 3]. A duplicate fragment is extracted as a separate reusable sequence diagram, and every occurrence of the fragment is replaced with an InteractionUses referring the separate reusable interaction.

The orders in which duplications are dealt with are also important. Duplications may overlap. As are result, the refactorings applied on duplications may conflict with each other [40], that is, applying a refactoring may disable others. Consequently, strategies are important in deciding which refactorings should be carried out first.

A feasible strategy is to prioritise duplications according to the following formula

$$\text{duplication length} \times (\text{amount of occurrences} - 1) \quad (6)$$

Formula (6) calculates the amount of messages that can be removed by extracting the duplication as a reusable sequence diagram. The more messages a refactoring can remove, the higher priority it would be assigned.

Changes in sequence diagrams may require (or suggest) corresponding changes in other diagrams of the system. For example, an extracted reusable sequence diagram may indicate a chance to extract an common use case or a reusable operation.

4.3 Advanced sequence diagrams

In Section 4.2, we confine ourselves to basic sequence diagrams. However, for complex systems, basic sequence diagrams are not powerful enough. We need advanced sequence diagrams to describe concurrency, iterations and alternatives. UML2.0 has introduced advanced concepts and mechanisms to deal with such situations. In order to make the proposed approach more practical, this subsection extends the approach to deal with advanced features of UML2.0 sequence diagrams: CombinedFragment, concurrency, Formal Gate and InteractionUse.

4.3.1 CombinedFragment: The main constraint of basic sequence diagrams is that a basic sequence diagrams can only contain one trace. In order to contain more than one trace in a diagram, UML2.0 introduces CombinedFragments: Through the use of CombinedFragments the user will be able to describe a number of traces in a compact and concise manner [1, pp. 507].

A CombinedFragment is composed of an operator (such as Loop, Assertion and Option) and a sequence of operands which are isolated from each other by horizontal bars as shown in Fig. 7. A CombinedFragment can be involved in an extractable fragment in the following two ways: The CombinedFragment is completely included in the extractable fragment, or the extractable fragment is confined to be an inter region of an operand of the CombinedFragment.

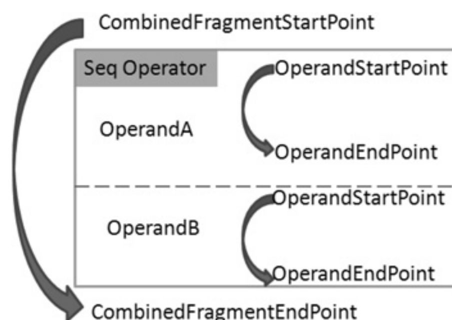

Fig. 7 *CombinedFragment*

To make sure that CombinedFragments are correctly involved, four kinds of vertices are inserted: CombinedFragmentStartPoint, CombinedFragmentEndPoint, Operand StartPoint and OperandEndPoint as shown in Fig. 7. Curves indicate what the Distance attribute means for the vertices.

If a CombinedFragmentStartPoint is involved, the prefix should be extended to its corresponding CombinedFragmentEndPoint. Therefore for every CombinedFragmentStartPoint C_i

$$G(i) = \max(G(i), C_i.\text{Distance} + i) \quad (7)$$

C_i.Distance is the distance from the current point (C_i) to its corresponding CombinedFragmentEndPoint. OperandStartPoints are dealt with in similar way: If a OperandStartPoint is involved, the corresponding OperandStartPoint should also be contained. The detailed algorithm to compare CombinedFragments is presented in Fig. 8

4.3.2 InteractionUse: As mentions in Section 1, different sequence diagrams may share certain common portions (duplications). Recently released UML2.0 realises the problem and introduces a new element InteractionUse to remove such kind of duplications. An InteractionUse refers to a common Interaction. Inserting an InteractionUse is semantically equivalent to copying the contents of the referred Interaction where the InteractionUse is inserted.

An InteractionUse is composed of four parts: a set of actual gates, referred interaction, parameters and return value. Parameters and return values can be compared as character strings. Referred interactions had better be compared by their contents. However, it may increase computation load dramatically. We have to balance between cost and benefits. As a result of the balance, we compare the names instead of the content of referred interactions here in order to make the algorithm more practical.

```
If (S_m(i).Type = CombinedFragmentStartPoint)
   If (S_m(i).interactionOperator ≠ S_n(i).interactionOperator)
      break;
   End If
   // Make sure to include the corresponding CombinedFragmentEndPoint
   G(i) = max(G(i − 1), S_m(i).Distance + i);
End If
If (S_m(i).Type = OperandStartPoint)
   // Make sure to include the corresponding OperandEndPoint
      G(i) = max(G(i − 1), S_m(i).Distance + i);
End If
If (S_m(i).Type = OperandEndPoint)
   If (S_m(i).Distance ≠ S_n(i).Distance)
      break;
   End If
   If (S_m(i).Distance ≤ i)
     // Make sure to include the corresponding OperandStartPoint
      break;
   End If
End If
If (S_m(i).Type = CombinedFragmentEndPoint)
   If (S_m(i).Distance ≠ S_n(i).Distance)
      break;
   End If
   If (S_m(i).Distance ≤ i)
     // Make sure to include the corresponding CombinedFragmentStartPoint
      break;
   End If
End If
```

Fig. 8 *Algorithm to compare CombinedFragments*

© The Institution of Engineering and Technology 2011

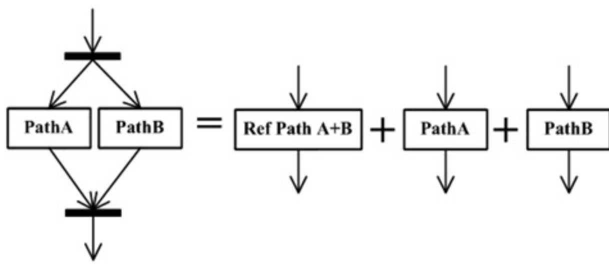

Fig. 9 *Concurrency*

Actual gates relay messages to formalGates of the interaction referred by the InteractionUse to which the actual gates are attached. The connection is usually established by matching the names of messages instead of the positions of gates [1, pp.523]. Therefore, we can remove actual gates, and connect messages directly to the InteractionUse. An InteractionUse may contain more than one message entering it. In order to compare entering messages, we sort the their entering messages into a sequence InMsgs by messages' names before comparison. Leaving messages are compared in similar way.

4.3.3 Concurrency: As discussed in Section 4.2.1, sequence diagrams can be converted into arrays only if they contain no concurrency. Therefore it is necessary to find some ways to decompose concurrency.

Fig. 9 illustrates how to decompose concurrency by extracting concurrent execution paths as separate diagrams. The left part of Fig. 9 is the original sequence diagrams where paths A and B are executed concurrently. In the right part, execution path A and B are extracted as separate diagram A and B, respectively. The concurrent part of the original diagram is replaced with a reference to $A + B$ indicating that A and B would be executed concurrently here. As a result, none of the three diagrams contains concurrency if Ref $A + B$ is considered as a black box, and the detection algorithm proposed in Section 4.2 can be applied to them now. We call the process concurrency decomposition.

Concurrent flows may communicate by asynchronous messages. During the concurrency decomposition, every asynchronous message is cut into an leaving message of the sponsor flow (for example, flow A), and a entering message of the receiver (B). Since the two messages can be related by their identical names [1, pp. 523], no information is lost.

4.3.4 Formal gate: A gate is a connection point for relaying a message outside an interactionFragment with a message inside the interactionFragment [1]. Since a formal gate is actually a representation of an OccurrenceSpecification that is not in the same scope as the gate, it is nature to deal with a formal gate as a special kind of OccurrenceSpecification: It has no lifeline or ExecutionSpecification. Just set its lifeline and ExecutionSpecifications to null.

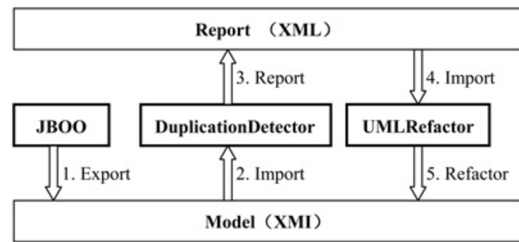

Fig. 10 *Interactions among JBOO, UMLRefactor and DuplicationDetector*

5 Tool support and evaluation

5.1 Tool support

In order to provide tool support for duplication detection, we have implemented the proposed approach as a plug-in DuplicationDetector in model refactoring tool UMLRefactor. UMLRefactor is a part of CASE JBOO 4.0 [41]. DuplicationDetector reports and locates duplications whereas UMLRefactor removes duplications by model refactoring. The cooperation of these tools is illustrated in Fig. 10.

DuplicationDetector exchanges data with JBOO via XMI files. When engineers call DuplicationDetector to detect duplications, the modelling tool JBOO will automatically generate an XMI file and pass it to the DuplicationDetector. Since XMI is the standard format of UML models, DuplicationDetector can also handle models built with other modelling tools. Although the previous versions of JBOO were released as products, the current version of JBOO is just a research project yet. DuplicationDetector is developed by our research group recently, and has been checked and accepted as a part of JBOO by the sponsor of the project.

5.2 Subject applications

With the tool support, we applied the proposed approach to six real industrial applications for evaluation. The applications are briefly introduced in Table 1.

The first application is an e-business website selling office supplies from copy paper, folders to personal computers. It is similar to New Egg (http://www.newegg.com). The application was implemented in C#, Javascript, HTML and SQL, costing more than 100 man-years. The second application is an office supplies management system. The system manages delivery and procurement of office supplies as well as meeting room reservation. It is also a web-based application: Users order office supplies with web browsers.

Table 1 Evaluation applications

Application ID	Application name	Modelling language	Amount of sequence diagrams
1	E-business Website	UML 1.x	35
2	Resource Management System	UML 2.0	15
3	Staff Information System	UML 1.x	20
4	Transportation Application	UML1.x	13
5	Sport Entries and Qualification Application	UML 1.x	9
6	Arrivals and Departures Application	UML 1.x	14

© The Institution of Engineering and Technology 2011

doi: 10.1049/iet-sen.2009.0029

The application was implemented in C# and SQL, costing 24 man-months. The first and the second applications were developed in the same company, and the first author of the paper was a member of the development team as a senior software engineer. For the development team, the e-business website was the first application they modelled in UML. On the contrary, when the resource management system (the second application) was developed, the development team became familiar with UML and model-based development. Furthermore, the first application is more complicated than the second one. Therefore we expected the first application to contain more duplications than the second one.

Other evaluation applications (Staff Information System, Transportation Application, Sport Entries and Qualification Application, and Arrivals and Departures Application) are sub-applications of a complicated Beijing Olympic Information System. Staff Information System records information of every person working for Beijing Olympic Games. Once needed, the information can be retrieved, classified and edited. Transportation Application manages the plan to transport athletes from the Olympic Village to gymnasiums. Sport Entries and Qualification Application manages the official entry of qualified and eligible athletes into the Olympic Games. Arrivals and Departures Application manages the arrivals and departures of athletes. These system had not yet been implemented when the evaluation was done. But models of these applications had been built, and they were ready to be evaluated.

The six applications were selected for the evaluation for the following reasons:

1. First, all of them have complete design models. It is easy to download source code of open source applications from internet, but it is really hard to find complete design models of applications, especially sequence diagrams which are needed in our evaluation.
2. Second, the models of these applications were described in UML. It is the right format our evaluation tools expect.
3. Finally, these application were designed by the authors and their colleagues who would carry out the evaluation. It might minimise the misunderstanding of the models during the evaluation.

5.3 Process

The evaluation was carried out by seven graduate students by applying the proposed approach to the subject applications. The students major in computer science in our institute. All of them are familiar with UML, and each of them has at least three years of experiments with object-oriented design and implementation. They have also participated in the develop of a UML modelling tool, and have taken a training course of UML before the development. Before the evaluation, they had been informed what duplications in sequence diagrams are, and how to deal with them. Backgrounds of the subject applications were also introduced to the evaluators to facilitate the evaluation.

For each evaluation application, the evaluation followed the following process:

1. First, engineers applied the detection tool on the evaluation application. It would report some potential duplications.
2. Second, engineers manually checked the potential duplications reported by the detection tool, and distinguished false-positive potential duplications.
3. Third, engineers manually checked the evaluation application to find out duplications missed by the detection tool.
4. Finally, precision rate and recall rate were computed. The precision rate and recall rate are defined as

$$\text{precision} = 100\% - \frac{\#\text{false} - \text{positive fragments}}{\#\text{reported duplicate fragments}} \quad (8)$$

$$\text{recall} = \frac{\#\text{deteted duplicate fragments}}{\#\text{all duplicate fragments}} \quad (9)$$

To remove accidental duplications, the size threshold 1 was experientially set to 4. The size of a fragment was measured by the amount of the messages it contains directly. The duplication percentage of a sequence diagram was computed by dividing the amount of messages involved in duplicate fragments by the total length of the diagram. Duplication percentages of applications were computed in the same way.

5.4 Results and analysis

5.4.1 Raw result: Evaluation results are presented in Table 2. In the 1st–6th evaluation applications, 8, 4, 5, 2, 2, 3 duplicate fragments in sequence diagrams were detected, respectively. Their duplication percentage were 14.8, 8.3, 7.1, 5.6, 4.8, 6.3%, respectively. The precision was 100% for all evaluation applications, that is no false duplication was reported. The recall rates were 80, 100, 100, 50, 100, 100%, respectively.

5.4.2 Analysis: First, the detection algorithm achieved a high precision rate. As suggested by the results, the precision rate was 100% for all evaluation applications.

Second, the detection algorithm could find out automatically most duplications in sequence diagrams. As suggested by the results, $24 = (8 + 4 + 5 + 2 + 2 + 3)$ duplicate fragments were found out, and only four fragments were missed. The total recall rate was $24/(24 + 2) = 92.3\%$.

Table 2 Evaluation result

Application ID	Amount of duplicate fragments	Duplication percentage, %	Recall, %	Precision, %
1	8	14.8	80	100
2	4	8.3	100	100
3	5	7.1	100	100
4	2	5.6	50	100
5	2	4.8	100	100
6	3	6.3	100	100

© The Institution of Engineering and Technology 2011

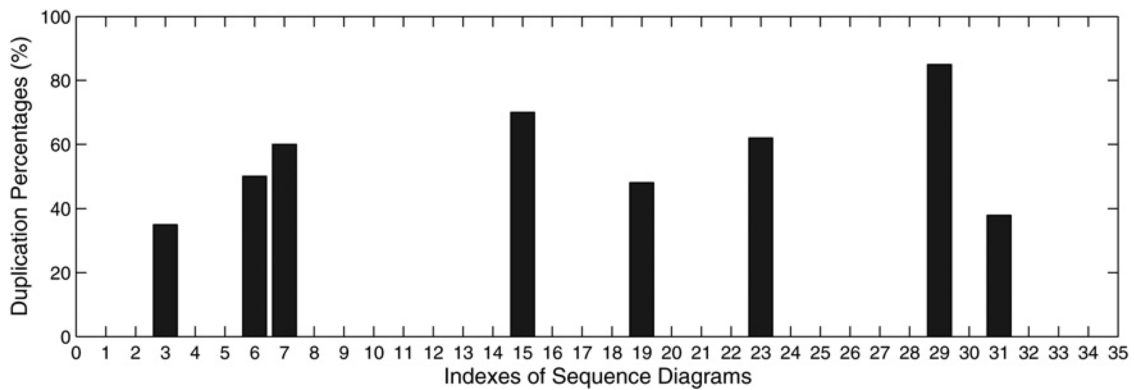

Fig. 11 *Duplication percentage per diagrams of the first application*

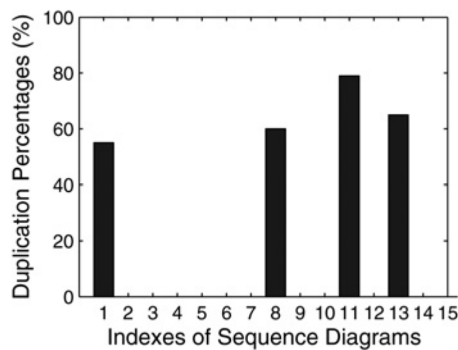

Fig. 12 *Duplication percentage per diagrams of the second application*

For the miss of the four duplicate fragments, we explain as follows:

1. The first two missed fragments came from the first evaluation application. The two fragments were equivalent, but not identical: Optional ExecutionSpecifications were explicitly depicted in one fragment whereas they were omitted in the other fragment. The different ways to deal with optional elements cause the failure of the detection approach. How to dealing with optional elements is discussed in Section 6.
2. The other two missed duplicate fragments came from the fourth evaliation application (Transportation Application). The two duplications contained the same object (lifeline), but entitled it with different names. How to deal with such kind of duplications is discussed in Section 6.

Third, duplication percentage might vary from application to application. As suggested by the evaluation results, the duplication percentage varied from 4.8% to 14.8%. The maximal duplication percentage is more than three times of the minimal (14.8% = 4.8%*3.0833). Its standard deviation is 0.0363, and its coefficient of variation ((standard deviation/mean) $\times$ 100%) is 46.42%.

Fourth, complex applications designed by beginners of UML are more likely to result in higher duplication percentage. As suggested by the evaluation results, the first application (e-business website) has a higher duplication percentage more than the second one. It consists with our expectation: If an application is complex, and the developers are unskilled in modelling, more duplications would be introduced. It also implies where and when to apply the proposed approach.

Finally, duplication percentage of a single sequence diagram is high, although the overall duplication percentage of an application is low. The distribution of duplicate fragments are presented in Figs. 11–16. For the 23 sequence diagrams containing duplications, the duplication percentage ranged from 30 to 85%, with a mean of 52.02%.

Removing duplications in sequence diagram has two benefits. First, it improves the maintainability of models, making them easier to change. Second, it may help to reduce duplicate code in implementation, and save implementation effort. The first benefit has been investigated in Section 5.4.1. In order to investigate the second benefit, we would investigate how many source lines could be saved if duplications in sequence diagrams are removed. Because the evaluated applications are commercial and protected by copyright, we are authorised to accessed the source code of

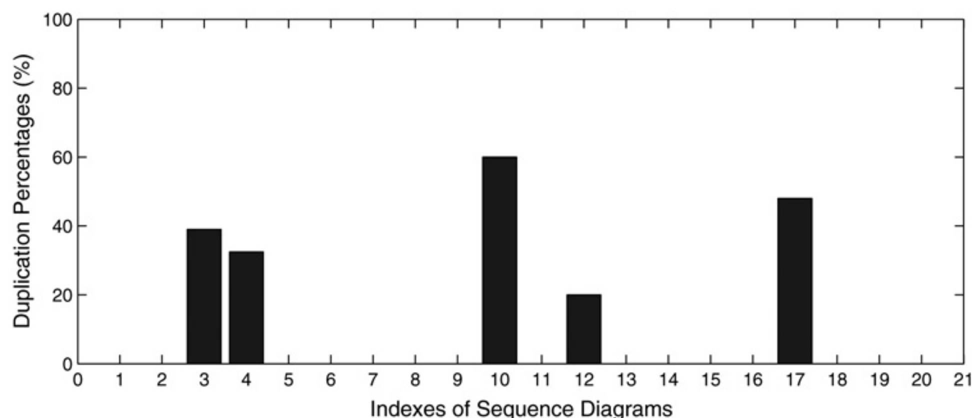

Fig. 13 *Duplication percentage per diagrams of the third application*

© The Institution of Engineering and Technology 2011
IET Softw., 2011, Vol. 5, Iss. 4, pp. 385–397
doi: 10.1049/iet-sen.2009.0029

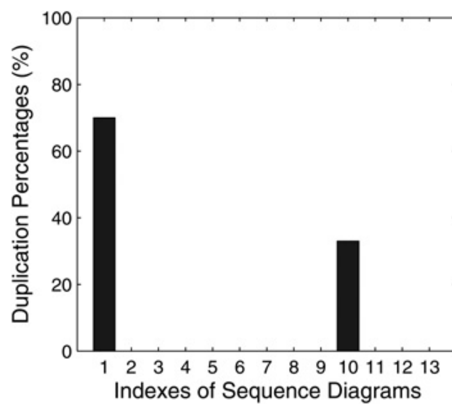

Fig. 14 *Duplication percentage per diagrams of the fourth application*

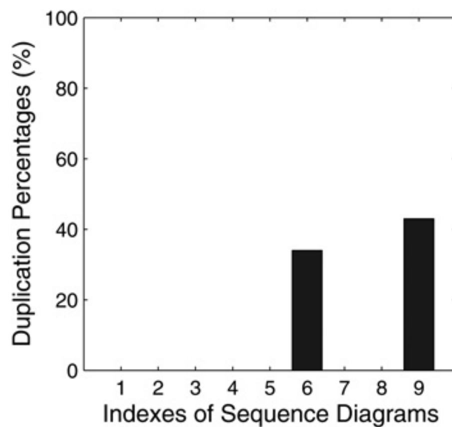

Fig. 15 *Duplication percentage per diagrams of the fifth application*

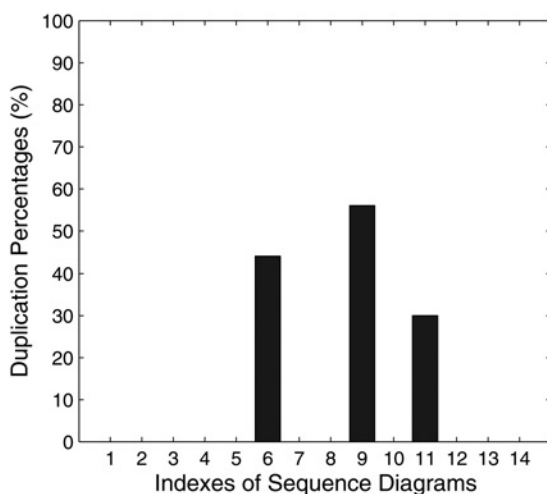

Fig. 16 *Duplication percentage per diagrams of the sixth application*

the second application only. The application has been implemented according to the original sequence diagrams, and it contains nearly 30 000 source lines. To find source code that could be removed as a result of removing duplications in sequence diagrams, we correlate duplicate fragments in sequence diagram to source code lines. After that, we investigate which parts of the source code could be reused. We find about 1000 lines of source code could have been saved if the application was implemented according to the refactored sequence diagrams. The duplication percentage ($1000/30\,000 = 3.3\%$) in source code is much lower than that in sequence diagrams (8.3%). The main reason is that not all source code lines can be correlated to sequence diagrams. Sequence diagrams model only important behaviours, letting others alone. Therefore, some duplications in source code could not be found in sequence diagrams.

5.5 Threats to validity

A threat to the external validity is the subjects of the evaluation. It is possible that some specific characteristics of the subjects led to our evaluation results. In order to reduce the threat, we carried out evaluation on six applications from different domains. It helps to generalise the conclusions.

A threat to the construct validity is that manual checking on evaluation applications might not find all duplicate fragments in sequence diagrams. As a result, the computation of recall rates might be inaccurate.

6 Discussion

6.1 Near-duplicate fragments

Currently, DuplicationDetector detects duplications only. It can not yet detect near-duplicate fragments. Sometimes, near-duplicate fragments can be unified and reused. For example, If the order of two method invocations is immaterial, different people may draw them in different orders. Another kind of near-duplications arise from synonyms. Different developers (or even the same developer) may describe identical objects or classes with different words which are synonymous in some way. In a certain context, two words are synonymous, but they may have different meaning in other contexts. So, thesaurus alone is not a satisfying solution.

To detect near-duplications, approximate matching is necessary. But approximate matching may be much more expensive. Further more, approximate matching may lead to great reduction in precision which may overtake the benefit of the improvement in recall.

6.2 Optional elements

Further work is needed to automatically delete (or attach) optional ExecutionSpecifications. If an Execution Specifications is not involved in a recursion, it is usually (not always) omitted in the diagrams. So, we can remove (or attach) optional ExecutionSpecifications before detection.

6.3 Evaluation criterion

The evaluation of the proposed approach focuses on its ability in detecting duplications in sequence diagrams, that is, its recall rate and precision rate. But the ultimate criterion to the approach should be how much load can be saved for the involved developers and maintainers. However, the measurement of the saved load is difficult for the following reasons:

1. First, in order to evaluate the effect of removing duplicate sequence diagrams on development and maintenance, we have to assign two teams to develop and maintain the same

© The Institution of Engineering and Technology 2011

application: The first team implements and maintains the application according to the original sequence diagrams whereas the other team restructures (refactoring) the diagrams by removing duplicate fragments from sequence diagrams before implementation and maintenance. However, it is hard, if not impossible, to apply such evaluation on real industrial applications because it would double the budget. It is unacceptable for any commercial company.

2. Second, the source code of the evaluation applications are not published. The first and second evaluation applications were developed in a commercial company. The company just kindly gave us the sequence diagrams for evaluation, but the source code and other artefacts or information (such as maintenance effort) are not publicly available. The other four applications (sub-applications of Beijing Olympic Information System) were designed by the researchers, but implemented and maintained by others.

7 Conclusion and future work

With the popularity of UML and MDA, we encounter a new problem: duplications in models. We analyse the reasons for duplications in sequence diagrams, analyse their impact in software development and maintenance, and reach a conclusion that duplications are causing serious problems for the development and maintenance of sequence diagrams. Then, it focuses on sequence diagrams of UML2.0, and propose an algorithm to detect duplications in sequence diagrams. We first map diagrams into an array, and then proposes a special algorithm to construct a suffix tree for the array. With the suffix tree, duplications are detected in the same way as duplicate source code is detected. With tool support, the approach is applied to real industrial applications for evaluation. The evaluation results suggest that duplications do exist in sequence diagrams and the proposed algorithm is effective in detecting them.

We also plan to detect duplications in other diagrams, such as use case diagrams and state charts. In order to facilitate model refactoring, algorithms to detect other bad smells in models are also necessary. We believe that more and more detection algorithms and tools will appear in the near future.

8 Acknowledgments

The work is funded by the National Natural Science Foundation of China (No. 61003065 and 60773152), Specialized Research Fund for the Doctoral Program of Higher Education (No. 20101101120027) and Excellent Young Scholars Research Fund of Beijing Institute of Technology (No. 2010Y0711).

9 References

1 Object Management Goup: 'UML 2.0 superstructure specification'. Technical Report ptc/04-10-02, 2004

2 Object Management Goup: 'MDA guide version 1.0.1'. Technical Report OMG/03-06-01, 2003

3 Selic, B.: 'What's new in UML 2.0?'. Technical report, IBM Rational Software, April 2005

4 Ren, S., Rui, K., Butler, G.: 'Refactoring the scenario specification: a message sequence chart approach'. Ninth Int. Conf. on Object-Oriented Information System, 1995, (*LNCS*, **2817**), pp. 294–298

5 Eriksson, H.-E., Penker, M., Lyons, B., Fado, D.: 'UML2 toolkit' (Wiley Publishing, Inc., Indianapolis, Indiana, 2004)

6 Jacobson, I., Christerson, M., Jonsson, P., Oevergaard, G.: 'Object-oriented software engineering – a use case driven approach' (Addison Wesley, Massachusetts, 1992)

7 Liu, H., Ma, Z., Zhang, L., Shao, W.: 'Detecting duplications in sequence diagrams based on suffix trees'. Proc. XIII Asia Pacific Software Engineering Conf. (APSEC'06), IEEE Computer Society, Bangalore, India, 6–8 December 2006, pp. 269–276

8 Baker, B.S.: 'On finding duplication and near-duplication in large software systems'. Second IEEE Working Conf. on Reverse Engineering, July 1995, pp. 86–95

9 Tairas, R., Gray, J.: 'Phoenix-based clone detection using suffix trees'. Proc. 44th Annual ACM Southeast Regional Conf., Melbourne, Florida, 2006, pp. 679–684

10 Kamiya, T., Kusumoto, S., Inoue, K.: 'CCFinder: a multi-linguistic token based code clone detection system for large scale source code', *IEEE Trans. Softw. Eng.*, 2002, **28**, (6), pp. 654–670

11 Ducasse, S., Rieger, M., Demeyer, S.: 'A language independent approach for detecting duplicated code'. Int. Conf. on Software Maintenance (ICSM '99), 1999, pp. 109–118

12 Baxter, I., Yahin, A., Moura, L., Anna, S.M., Bier, L.: 'Clone detection using abstract syntax trees'. Int. Conf. on Software Maintenance (ICSM '98), 1998, pp. 368–377

13 Opdyke, W.F.: 'Refactoring object-oriented frameworks'. PhD thesis, University of Illinois at Urbana-Champaign, 1992

14 Beck, K.: 'Extreme programming explained: embrace change' (Addison Wesley, 2000)

15 Pipka, J.U.: 'Refactoring in a 'test first'-world'. Third Int. Conf. on Extreme Programming and Flexible Processes in Software Engineering', May 2002, pp. 178–181

16 van Deursen, A., Moonen, L.: 'The video store revisited – thoughts on refactorings and testing'. Third Int. Conf. on eXtreme Programming and Flexible Processes in Software Engineering (XP2002), May 2002, pp. 71–76

17 Roberts, D., Brant, J., Johnson, R.: 'A refactoring tool for smalltalk', *Theory Pract. Object Syst. Spec. Issue Object-Oriented Softw. Evol. Re-Eng.*, 1997, **3**, (4), pp. 253–263

18 Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: 'Refactoring: improving the design of existing code' (Addison Wesley Professional, 1999)

19 Philipps, J., Rumpe, B.: 'Refinement of information flow architectures'. Proc. First IEEE Int. Conf. on Format Engineering Methods 1997, Hiroshima, Japan, 1997, pp. 203–212

20 Tokuda, L., Batory, D.: 'Evolving object-oriented designs with refactorings', *Autom. Softw. Eng.*, 2001, **8**, (11), pp. 89–120

21 Ivkovic, I., Kontogiannis, K.: 'A framework for software architecture refactoring using model transformations and semantic annotations'. Proc. Tenth European Conf. on Software Maintenance and Reengineering 2006 (CSMR 2006), 22–24 March 2006

22 Russo, A., Nuseibeh, B., Kramer, J.: 'Restructuring requirements specifications', *IEE Proc., Softw.*, 1999, **146**, (1), pp. 44–53

23 Liu, H., Yang, L., Niu, Z., Ma, Z., Shao, W.: 'Facilitating software refactoring with appropriate resolution order of bad smells'. Seventh joint meeting of the European Software Engineering Conf. (ESEC) and the ACM SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE), 2009, pp. 265–268

24 Liu, H., Li, G., Ma, Z., Shao, W.: 'Conflict aware scheduling of software refactorings', *IET Softw.*, 2008, **2**, (5), pp. 446–460

25 Liu, H., Li, G., Ma, Z., Shao, W.: 'Scheduling of conflicting refactorings to promote quality improvement'. Twenty-Second IEEE/ACM Int. Conf. on Automated Software Engineering, 2007, pp. 489–492

26 Sunyé, G., Pollet, D., Traon, Y.L., Jezequel, J.-M.: 'Refactoring UML models'. Fourth Int. Conf. Unified Modeling Language, 2001, pp. 134–148

27 Astels, D.: 'Refactoring with UML'. Third Int. Conf. on eXtreme Programming and Flexible Processes in Software Engineering, 2002, pp. 67–70

28 Correa, A., Werner, C.: 'Applying refactoring techniques to UML/OCL models'. Seventh Int. Conf. Unified Modeling Language (UML 2004), 2004, pp. 173–187

29 Object Management Group: 'UML 2.0 OCL specification'. Technical Report ptc/03-10-14, 2003

30 Straeten, R.V.D., Jonckers, V., Mens, T.: 'Supporting model refactorings through behaviour inheritance consistencies'. Proc. UML 2004 – the Unified Modelling Language: Modelling Languages and Applications. 7th International Conference, Lisbon, Portugal, 11–15 October 2004, pp. 305–319

31 Liu, H., Ma, Z., Shao, W.: 'Description and proof of property preservation of model transformations', *J. Softw.*, 2007, **18**, (10), pp. 2369–2379 (In Chinese with English abstract)

© The Institution of Engineering and Technology 2011

doi: 10.1049/iet-sen.2009.0029

32 Mens, T., Tourwé, T.: 'A survey of software refactoring', *IEEE Trans. Softw. Eng.*, 2004, **30**, (2), pp. 126–139

33 Yu, W., Li, J., Butler, G.: 'Refactoring use case models on episodes'. Proc. 19th Int. Conf. on Automated Software Engineering 2004, 20–24 September 2004, pp. 328–335

34 Xu, J., Yu, W., Rui, K., Butler, G.: 'Use case refactoring: a tool and a case study'. 11th Asia-Pacific Software Engineering Conf. 2004, Busan, Korea, 30 November to 3 December 2004, pp. 484–491

35 Liu, H., Shao, W.Z., Zhang, L., Ma, Z.Y.: 'Detecting overlapping use cases', *IET Softw. (formerly IEE Proc. Softw.)*, 2007, **1**, (1), pp. 29–36

36 Grossi, R., Italiano, G.: 'Suffix trees and their applications in string algorithms'. First South American Workshop on String Processing (WSP1993), September 1993, pp. 57–76

37 McCreight, E.M.: 'A space-economical suffix tree construction algorithm', *J. Assoc. Comput. Mach.*, 1976, **23**, (2), pp. 262–272

38 Chen, M.T., Seiferas, J.: 'Efficient and elegant subword tree construction', Apostolico, A., Galil, Z. (Eds.): 'Combinatorial algorithms on words' (Springer Verlag, Berlin, 1985), pp. 97–107

39 Ukkonen, E.: 'On-line construction of suffix trees'. Technical Report A-1993-1, Department of Computer Science, University of Helsinki, 1993

40 Mens, T., Taentzer, G., Runge, O.: 'Detecting structural refactoring conflicts using critical pair analysis', *Electron. Notes Theor. Comput. Sci.*, 2005, **127**, (3), pp. 113–128

41 Zhiyi, M., Junfeng, Z., Xiangwen, M., Wenjun, Z.: 'Research and implementation of jade bird object-oriented software modeling tool', *J. Softw.*, 2003, **14**, (1), pp. 97–102 (in Chinese)

© The Institution of Engineering and Technology 2011