Detecting Duplications in Sequence Diagrams Based on Suffix Trees

Hui Liu, Zhiyi Ma*, Lu Zhang, Weizhong Shao*
Software Institute, School of Electronics Engineering and Computer Science
Peking University, Beijing 100871, China
{liuhui04,mzy,zhangl, wzshao}@sei.pku.edu.cn

Abstract

With the popularity of UML and MDA, models are replacing source code as core artifacts of software development and maintenance. But duplications in models reduce models' maintainability and reusability. To address the problem, we should detect duplications first. As an initial step to address the problem, we propose an approach to detect duplications in sequence diagrams. With special preprocessing, we convert 2-dimensional sequence diagrams into a 1-dimensional array. Then we construct a suffix tree of the array. We revise the traditional construction algorithm of suffix trees by proposing a special algorithm to detect common prefixes of suffixes. The algorithm ensures that every duplication detected with the suffix tree can be extracted into a separate reusable sequence diagram. With the suffix tree, duplications are found as refactoring candidates. With tool support, the proposed approach has been applied to real industrial projects, and the evaluation results suggest that the approach is effective.

1 Introduction

As a result of the popularity of MDA (Model Driven Architecture) [9] and UML (Unified Modeling Language) [10], models are replacing source code as core artifacts of software development and maintenance. UML was proposed by OMG, and became a de facto standard modeling language. Based on the success of UML, OMG brought forward the concept of MDA. In the context of MDA, models are automatically transformed into source code, which in turn is automatically transformed into executable. 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 main artifacts that developers and maintainers deal with. As a result, the quality (especially maintainability) of models becomes a big concern for most nontrivial projects, and progresses on revealing and improving the quality of models are desirable.

Model duplications are identical copies of the same model fragments. Duplications reduce models' maintainability and reusability [12] [11], and thus are usually considered as bad smells. Therefore, we had better detect and remove duplications to improve models' quality.

As an initial step to address the problem of model duplications, we propose an approach to detect duplications in sequence diagrams. Sequence diagrams are heavily used in system modeling to describe behaviors of use cases, operations and collaborations.

There are lots of duplications in sequence diagrams [12] [11]. The first reason (objective) comes from the complexity of systems. We often use the divide-and-conquer policy to deal with complex systems. Unfortunately, duplications appear as a byproduct of the policy. The second reason (subjective) is poor design, especially lack of *abstract*. The third reason comes from designers' reluctancy to restructure their design. The fourth reason is closely related to the way sequence diagrams are used. For a scenario (or a use case, an algorithm and so on), there is often a main execution flow and several alternative flows, and these flows are similar to each other (in other words, they share common parts) [4]. In order to make sequence diagrams as clear as possible, one sequence diagram usually describes only one execution flow [7]. As a result, similar flows are described by different sequence diagrams, and the common parts of the flows are turned into duplications in the resulting sequence diagrams.

Duplications in sequence diagrams, just as duplications in other diagrams, reduce maintainability and reusability of sequence diagrams [12] [11]. OMG has also realized the problem of duplications in sequence diagrams, and made a nontrivial revision on sequence diagram metamodel so as to avoid or remove duplications in sequence diagrams [12] [10]. Some impacts of duplications in sequence diagrams are listed below:

1. Duplications make it difficult to modify existing se-

*Corresponding authors

XIII ASIA PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC'06)
0-7695-2685-3/06 $20.00 © 2006 IEEE

quence diagrams. Changes to a piece of a diagram have to be carried out in all the duplications of the piece. If some of them are not changed synchronously, these changes may lead to inconsistency or even insecurity in the resulting system.

2. Duplications increase the size of the models.

3. Duplications may increase the workload of implementation.

Although UML2.0 provides an effective mechanism to remove duplications in sequence diagrams, it is still left to developers and maintainers to find out duplications.

This paper proposes an algorithm to automatically detect duplications in sequence diagrams based on suffix trees. 2-dimensional sequence diagrams are converted into a 1-dimensional array, and a suffix tree of the array is constructed. We also revise the traditional construction algorithm of suffix trees. The algorithm ensures that every duplication detected with the suffix tree can be extracted into a separate reusable sequence diagram.

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

2 Basic Sequence Diagrams and Suffix Trees

2.1 Basic Sequence Diagrams and Duplicate Fragments

Basic sequence diagrams of UML2.0 [10] 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 3.3.

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.*

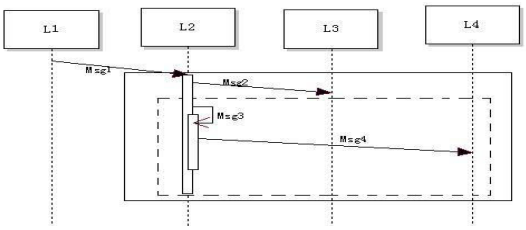

Figure 1. Extractable Fragments

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 into 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 ExecutionSpecification.*

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 into 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: ① They contain*

0-7695-2685-3/06 $20.00

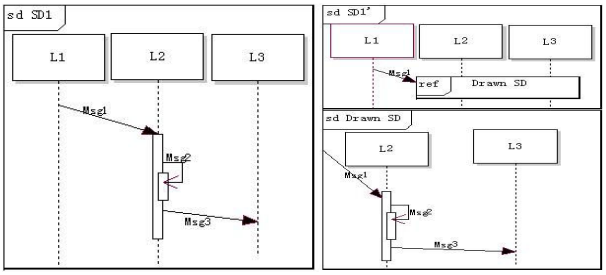

Figure 2. Cut Messages

the same elements, and ② 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 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 into separate interactions. The sequence diagram *SD1* shown in Fig 2 is a good example. An extractable fragment of *SD1* is extracted into a separate diagram *Drawn SD*, and the original diagram is restructured as *SD1'* referring *Drawn SD*. The initial point of the entering message Msg_1 is replaced with a gate when the extractable fragment is extracted into a separate interaction *Drawn SD*. 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.

2.2 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 [6].

Let S_0 be an array of $(n-1)$ elements from Σ, and \$ is an element matching no element of Σ. We attach \$ to the end of S_0 and get 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 $S = abcbc\$$.

A suffix trees T has two properties [6]:

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.

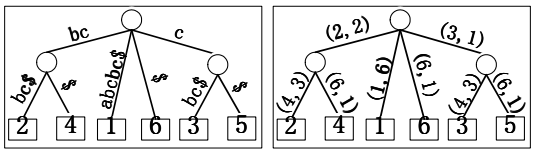

Figure 3. Suffix Tree and Its Compact Representation

These properties tell us how to detect duplications: Every internal node of T is a duplication shared by all the leaves (suffixes) beneath it. For example, from Fig. 3, we know that "bc" appears twice in the original string "$abcbc\$$" because its corresponding node has two leaves in the tree (the second suffix "$bcbc\$$" and the fourth suffix "$bc\$$").

The basic construction algorithm of suffix trees requires $O(n^2)$ time complexity. But advanced algorithms, such as those given by McCreight [8], Chen and Seiferas [3], and Ukkonen [15] 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 prefixes 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. In the next section, the proposed approach is discussed in detail.

3 Detection Approach

3.1 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 3.2, and then the approach is generalized to deal with advance features of UML2.0 sequence diagrams in Section 3.3.

3.2 Basic Sequence Diagrams

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

0-7695-2685-3/06 $20.00 © 2006 IEEE

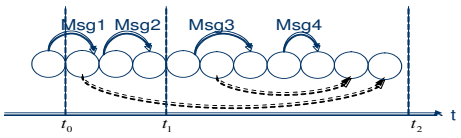

Figure 4. Compressed Sequence Diagram

3.2.1 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 Equation 1.

$$L = \{\ell \mid \exists o \in O \cdot \ell = o.Lifeline\} \quad (1)$$

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 2.1, 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-dimensional basic sequence diagram can be mapped into a 1-dimensional 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 as shown in Fig.4 is briefly identified as a CSD (Compressed Sequence Diagram).

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

3.2.2 Constructing Suffix Trees for Basic Sequence Diagrams

We gather arrays mapped from a set of sequence diagrams, and then concatenate the arrays into a long array (*LA*). As discussed in Section 2.2, 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 of a CSD corresponds to an extractable fragment, we call it an 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 OccurrenceSpecification (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 Msg whereas OccurrenceSpecification 3 and 4 receive Msg. 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

$$Sm(i).InMsg.Distance < i \quad (2)$$

where $Sm(i)$ is the ith element of the suffix Sm. As

0-7695-2685-3/06 $20.00

```
//Input: suffix Sm, Sn (m>n), MinimalLength
For(i=1;i<=Sm.L;i++)
 //Loop to compare every pair of elements
  If (Sm(i).Type != Sn(i).Type)
       Break;
  End If
  If (Sm(i).Type = OccurrenceSpecification)
    If (Sm(i) is unequal to Sn(i))
      //by comparing attributes: Lifelie,
      //  InMsg, OutMsg and Exs.
         Break;
    End If
    G(i)=Max(G(i-1),i);//Default value
    For(j=1;i<=Sm(i).Exs.Length;j++)
      If (i < Sm(i).Exs(j).SPoint)
       //Some ExecutionSpecification
       //beneath is cut
         Goto Maxj;
      End If
      G(i)=Max(G(i),Sm(i).Exs(j).EPoint+i);
    End For
  End If
  IF (Sm(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
```

Figure 5. Algorithm to Detect Longest Common Prefix of Two Suffixes

discussed in Section 2.1, 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.

For every OccurrenceSpecification, the ExecutionSpecifications 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

Figure 6. Comparison of Messages

the prefix section by

$$Sm(i).Exs(j).SPoint < i \tag{3}$$

Where $Sm(i).Exs(j)$ is the jth ExecutionSpecification of $Sm(i)$. With $EPoint$, we can get 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):

```
for(j=1;j<=Sm(i).Exs.Length;j++)
  G(i)= Max(G(i),Sm(i).Exs(j).EPoint+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:

$$(G(j) < j) \wedge (i > j \geq MinimaLength) \tag{4}$$

Constraint 4 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.

3.2.3 Duplication Detection and Model Refactoring

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.

Duplications in sequence diagrams can be removed with InteractionUses [12] [10]. A duplicate fragment is extracted into a separate reusable sequence diagram, and every occurrence of the fragment is replaced with an InteractionUse referring the separate reusable interaction.

3.3 Advanced Sequence Diagrams

In Section 3.2, 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.

0-7695-2685-3/06 $20.00 © 2006 IEEE

3.3.1 CombinedFragment

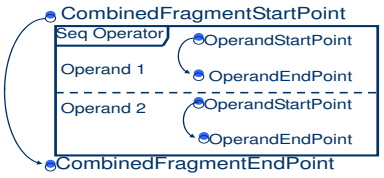

Figure 7. CombinedFragment

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* [10, 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.

To make sure that CombinedFragments are correctly involved, four kinds of vertices are inserted: CombinedFragmentStartPoint, CombinedFragmentEndPoint, OperandStartPoint, 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:

$$G(i) = \max(G(i), C_i.Distance + i) \qquad (5)$$

$C_i.Distance$ is the distance from the current point (C_i) to its corresponding CombinedFragmentEndPoint. OperandEndPoints are dealt with in similar way: If a OperandEndPoint is involved, the corresponding OperandStartPoint should also be contained.

3.3.2 Concurrency

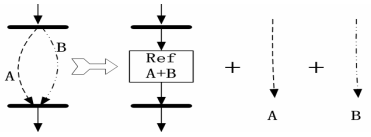

Figure 8. Concurrency

Fig. 8 illustrates how to decompose concurrency by extracting concurrent execution paths into separate diagrams. The left part of Fig. 8 is the original sequence diagrams with paths ***A*** and ***B*** executed concurrently. In the right part, execution path ***A*** and ***B*** are extracted into 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 3.2 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 [10, pp.523], no information is lost.

3.3.3 Formal Gate

A gate is a connection point for relaying a message outside an interactionFragment with a message inside the interactionFragment [10]. 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*.

3.3.4 InteractionUse

An InteractionUse is composed of four parts: a set of actual gates, referred interaction, parameters and return value. Parameters and return value 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.

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 [10, 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.

0-7695-2685-3/06 $20.00

Table 1. Industrial Projects and Evaluation Results

Industrial Projects	Target Users	Modeling Languages	No. of Sequence Diagrams	Duplication Percentage
E-business Website	External Clients	UML1.x	35	14.8%
Resource Management System	Internal Staff	UML2	15	8.3%

4 Tool Support and Evaluation

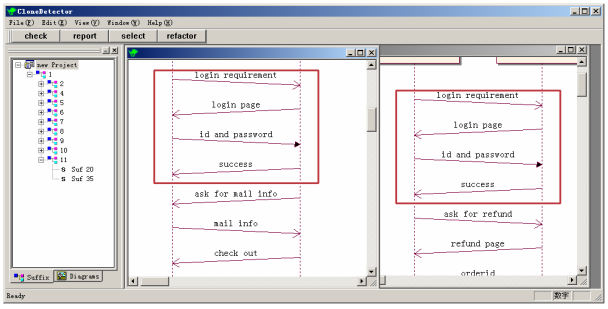

Figure 9. The Main Window of *DuplicationDetector*

In order to provide tool support for duplication detection, we implement the proposed approach as a plug-in *DuplicationDetector* for our model refactoring tool *UMLRefactor*, which in turn is a part of our CASE *JBOO* 4.0 [16]. *DuplicationDetector* reports and locates duplications whereas *UMLRefactor* removes duplications by model refactoring.

With the tool support, we apply the proposed approach to two real industrial projects for evaluation. The projects are shown in Table. 1. The first is an e-business website selling office supplies from copy paper, folders to personal computers. The second is an office supplies management system. The system manages delivery and procurement of office supplies as well as meeting room reservation. For the development team, the e-business website is the first project they modelled in UML. Therefore we cannot expect too much on the model quality. On the contrary, when the resource management system was developed, they became familiar with UML and model-based development.

To remove accidental duplications, the size threshold ℓ is experientially set to 4. The size of a fragment is identified by the amount of the messages it contains directly. The duplication percentage of a sequence diagram is contained by dividing the amount of messages involved in duplicate fragments by the total length of the diagram. Duplication percentages of projects are computed in the same way.

Evaluation results are presented in Table 1, and details are shown in Fig.10 and Fig.11. As suggested by the evaluation results, project1 (e-business website) has a higher duplication percentage than project2. The results consist with

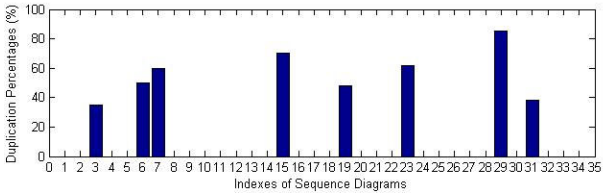

Figure 10. Duplication Percentage per Diagrams of Project1

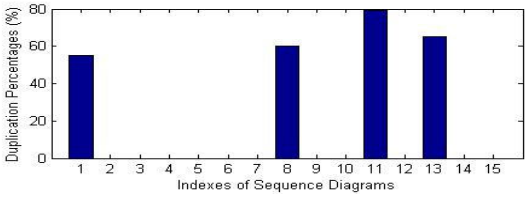

Figure 11. Duplication Percentage per Diagrams of Project2

our expectations: If a project is complex, requirements are instable, and the developers are beginners in modeling, duplications have more chances to be created. It also suggests us where and when to apply the *DuplicationDetector*.

Precisions on the two projects are 100%, i.e. no false duplication is reported. The recall on the second project is also 100%, i.e. all duplications found manually are reported by the approach. But unfortunately, two (and only two) equivalent fragments of the first project are not found by the approach. The two fragments are equivalent, but not identical: Optional ExecutionSpecifications are explicitly depicted in one fragment whereas they are omitted in the other fragment. The different ways to deal with optional elements cause the failure of the detection approach. Dealing with optional elements is planned in our future work.

Duplications are presented to developers as refactoring candidates. If duplications are considered as bad smells, developers carry out refactorings to remove them. Then we applied the detection approach to the refactored models, and the duplication percentages of the two projects are reduced to 3.9% and 0%, respectively. The results suggest that most of the reported duplications are considered by developers as bad smells and removed by model refactorings.

5 Related Work

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, such as Dup [1] and CloneDR [2], to automatically detect duplicate source code.

However, the existing detection approaches have not taken models into consideration. One of the possible reasons is that MDA and UML are quite new technologies, which were proposed recently by OMG.

In recent years, the quality of models deserves more and more concern. As a response to it, model refactoring [13] is proposed to improve models' quality while preserving models' external behaviors. Model refactoring is usually performed in the following way. Experts define some *bad smells* first which are indicators of bad design, and then propose certain refactorings to remove the bad smells [14] [5]. Of course, duplications are bad smells which deserve refactoring, and Ren et al. [11] 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 bad smells in models is rarely discussed whereas there are rich detection algorithms and tools for bad smells (especially duplications) in source code. The detection approach proposed in this paper is an attempt to address this issue.

6 Conclusion and Future Work

With the popularity of UML and MDA, we encounter a new problem : duplications in models. We analyze the reasons for duplications in sequence diagrams, analyze their impact in development and maintenance, and reach a conclusion that duplications are causing serious problems for the development and maintenance of sequence diagrams. Then, we propose an algorithm to detect duplications in sequence diagrams. With tool support, the approach is applied to real industrial projects for evaluation. The evaluation results suggest that duplications do exist in sequence diagrams and the proposed algorithm is effective in detecting them.

Further work is needed to automatically delete (or attach) optional ExecutionSpecifications. If an ExecutionSpecification is not involved in a recursion, it is usually (not always) omitted in the diagrams. So, we had better remove (or attach) optional ExecutionSpecifications before detection. How to deal with synonyms is another problem to be solved. Different developers (or even the same developer) may describe identical messages, objects or classes with different words which are synonymous in some way. We also plan to detect duplications in other diagrams, such as use case diagrams and state charts.

7 Acknowledgements

The work is funded by the National Grand Fundamental Research 973 Program of China No. 2005CB321805,the National Natural Science Foundation of China No. 60473064, and the Key Technologies R&D Programme No. 2003BA904B02.

References

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *the Second IEEE Working Conference on Reverse Engineering*, pages 86–95, July 1995.

[2] I. Baxter, A. Yahin, L. Moura, S. Anna, M, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance (ICSM '98)*, pages 368–377, 1998.

[3] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, pages 97–107. Springer Verlag, Berlin, 1985.

[4] H.-E. Eriksson, M. Penker, B. Lyons, and D. Fado. *UML2 Toolkit*. Wiley Publishing, Inc., Indianapolis, Indiana, 2004.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[6] R. Grossi and G. Italiano. Suffix trees and their applications in string algorithms. In *the First South American Workshop on String Processing (WSP1993)*, pages 57–76, September 1993.

[7] I. Jacobson, M. Christerson, P. Jonsson, and G. Oevergaard. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison Wesley, Massachusetts, 1992.

[8] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, April 1976.

[9] Object Management Goup. MDA guide version 1.0.1. Technical Report OMG/03-06-01, 2003.

[10] Object Management Goup. UML 2.0 superstructure specification. Technical Report ptc/04-10-02, 2004.

[11] S. Ren, K. Rui, and G. Butler. Refactoring the scenario specification: A message sequence chart approach. In *9th International Conference on Object-Oriented Information System, LNCS 2817*, pages 294–298, 1995.

[12] B. Selic. What's new in UML 2.0? Technical report, IBM Rational Software, April 2005.

[13] G. Sunye, D. Pollet, Y. L. Traon, and J.-M. Jezequel. Refactoring UML models. In *the Fourth International Conference on the Unified Modeling Language*, pages 134–148, 2001.

[14] M. Tom and T. Tom. A survey of software refactoring. *IEEE Transition on Software Engineering*, 30(2):126–139, 2004.

[15] E. Ukkonen. On-line construction of suffix trees. Technical Report A-1993-1, Department of Computer Science, University of Helsinki, Finland, 1993.

[16] M. Zhiyi, Z. Junfeng, M. Xiangwen, and Z. Wenjun. Research and implementation of jade bird object-oriented software modeling tool. *Journal of Software*, 14(1):97–102, 2003. in Chinese.