Detecting overlapping use cases

H. Liu, W.Z. Shao, L. Zhang and Z.Y. Ma

Abstract: To deal with the complexity of large information systems, the divide-and-conquer policy is usually adopted to capture requirements from a large number of stakeholders: obtain requirements from different stakeholders, respectively, and then put them together to form a full requirement specification. One of the problems induced by the policy is overlapping requirements. A use case driven approach could not avoid overlapping requirements either: it produces overlapping use cases, which are even more harmful, because a use case describes not only inputs and outputs as traditional requirements do, but also the scenarios. Each of the overlapping use cases provides a message sequence to implement the common subgoal. Overlapping use cases not only decrease the maintainability of the requirement specification, but also result in a complicated, confusing and expensive system. To be worse, it is difficult to detect overlapping use cases with existing methods for requirement management. To find out overlapping use cases, a detection approach using sequence diagrams and statecharts is proposed. Evaluation results suggest that practical requirement models do contain overlapping use cases, and the proposed approach is effective in detecting them.

1 Introduction

Requirements engineering is very hard [1]. Surveys suggest that most of the requirement models are inconsistent, inadequate and confusing. And errors in requirements lower the quality and maintainability of the resulting software, increase the cost and defer the release. Furthermore, the later the requirement errors are fixed, the more expensive the software is. As a result, efficient approaches are urgently needed to improve the quality of software requirements.

Nowadays, information systems become more and more complicated and involve more and more stakeholders in capturing requirements. Since a large system may involve numerous stakeholders, there will be a large number of stakeholders who would like to bring forward their requirements or their expectations on the system to be built. The number is too large for an engineer to capture the requirements from all of them. So, the stakeholders are usually divided into small groups, and each group will be assigned to a requirements engineer (or a group of requirements engineers). The engineer communicates with stakeholders of his group, writes down their requirements and prepares them in a predefined format. In this way, the complexity of the systems is solved successfully. However, it also brings a new problem: overlapping requirements. Stakeholders divided into different groups may propose overlapping requirements. But unfortunately, these stakeholders do not have opportunities to exchange their ideas so as to find their common interest because they belong to different groups. On the other hand, different groups of stakeholders are assigned to different requirements engineers, which means requirements engineers cannot find out overlapping requirements either.

Since the concept of use cases was brought forward by Jacobson [2], it has been a popular tool for requirements engineering [3, 4]. A use case is a way to achieve the actor's goal, which may be divided into subgoals. The way is depicted by message sequences indicating interactions between the actor(s) and the system [3]. A single execution path of a use case is a scenario [5], which is presented as a message sequence. Sometimes other attributes, such as goals, pre-conditions and post-conditions are given in addition to the message sequences. Use case based approaches differ from traditional ones in that they describe ways (use cases) to achieve users' goals. Stakeholders can understand the description, as well as requirements engineers. So, use cases serve as the basic of their communications. However, they cannot avoid the problem of overlapping requirements. On the contrary, they worsen the problem by introducing overlapping use cases, which will be discussed in this paper. Overlapping use cases is a new form of overlapping requirements. So, they may cause inconsistency too, although it is not the focus of this paper. In this paper, we focus on different message sequences achieving the same goal (subgoal).

Since the concerns of different stakeholders may overlap with each other, it is very likely for different use cases, each of which typically contains several subgoals, to share some common subgoals. As a result, it is also inevitable for use cases to overlap with each other. In other words, there are many use cases providing message sequences to achieve a common subgoal. We call these message sequences overlapping message sequences, which distribute in different use cases but achieve the same goal. And the use cases containing the overlapping message sequences are called overlapping use cases. So, the problem of overlapping use cases is equal to that of overlapping message sequences. Overlapping use cases share certain common subgoals, but their overall goals may differ. It should be noted that overlapping message sequences are not necessarily identical. In the literature, a special situation of this problem (in

doi:10.1049/iet-sen:20060023

Paper first received 28th April and in revised form 6th December 2006

The authors are with the Software Institute, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, People's Republic of China

E-mail: liuhui2005@gmail.com

which the overlapping message sequences in different use cases are identical) has been discussed recently [6, 7]. For convenience, we call this special case ISDUC (identical sequences in different use cases). However, the problem in general has not been addressed. As stakeholders usually provide use cases in isolation, it is typical for them to use different message sequences to implement the same subgoal. When the overlapping message sequences are different, we call the problem DSSG (different sequences for the same goal).

In this paper, a statechart-based approach is proposed to detect overlapping use cases. Although the focus of the approach is DSSG, it can detect all overlapping message sequences, no matter they are identical or not. So, it detects not only DSSG but also ISDUC, that is it detects overlapping message sequences in general. However, how to deal with these overlapping message sequences is left to stakeholders and requirements engineers because of the following reasons. First, it cannot decide which message sequence is the best one without humans intervention. Second, sometimes different ways to do the same thing are really needed for fault-tolerance or users' convenience.

2 Problem of overlapping use cases

In this section, we illustrate the problem of overlapping use cases through an e-business example, discuss why overlapping use cases should be detected and analyse the difficulty of the detection.

2.1 Example

We take an e-business website as an example. As depicted in Fig. 1 (for clarity, we omit interactions irrelevant to registration for both use cases), there are two use cases Consultation and Purchase, which are proposed independently by $Customer_1$ and $Customer_2$, respectively. If customers are unregistered, registration will be needed before consultation. So, registration is included in Consultation. During purchase, a customer selects goods by pulling them into a shopping cart and then makes an order. Just at this moment, customer ID is required which should be got by registration. As a result, registration should also be included in Purchase.

We suppose that $Customer_1$ and $Customer_2$ are divided into different groups, which are assigned to different requirements engineers. Furthermore, $Customer_1$ has no idea of purchase, whereas $Customer_2$ never considers consultation. So, when preparing requirements, the requirements engineer gathering requirements from $Customer_1$ would write a long use case combining interactions of Consultation and Registration because there is no need for him to extract registration actions out as a stand-alone use case. The same is true for $Customer_2$ and the requirements engineer who gathers requirements from him. Obviously, corresponding message sequences (as shown in Fig. 1) of the two use cases complement the same subgoal: registration. In other words, use case Consultation and use case Purchase overlap with each other. But as shown in Fig. 1, the overlapping use cases achieve different overall goals (Consultation and Purchase). We also note that the overlapping message sequences, which achieves the same goal (registration), differ from each other. In Section 2.4, we will discuss how the difference between overlapping message sequences (and that between overall goals of overlapping use cases) complicates the detection of overlapping use cases.

Use Case: Consultation
1. Customer: Input customer ID
2. System: Validate that it is unique
3. Customer: Input country name
4. System: Validate country name
5. Customer: Input zip code
6. System: Validate the zip code
7. System: Get city name by zip code
8. Customer: Input customer name
9. System: Validate that it is a valid name
10. .....

Use Case: Purchase
1. Customer: Pull goods into a shopping cart
2. System: Ask unregistered customer to register first.
3. Customer: Input customer's name
4. System: Check spelling mistake
5. Customer: Input customer ID
6. System: Validate that it is unique
7. Customer: Select country
8. Customer: Input city name
9. System: Validate the city name against the country
10. System: Success confirm

Fig. 1 *Rough use cases*

2.2 Impacts of overlapping use cases

Overlapping use cases can cause harmful impacts on the whole life cycle of the system, although it is possible to develop a system implementing all of them.

Overlapping requirements may cause inconsistency. As different requirements are proposed by different stakeholders, it is usually unavoidable for the following situation to occur. Each individual requirement is well described and satisfiable, but it is impossible to satisfy all of them together. In the literature, this problem has already been intensively discussed [8–11]. In this paper, we are mainly concerned with overlapping use cases, which can be satisfied simultaneously by a single system.

Other impacts of overlapping use cases are briefly discussed below: First, overlapping use cases may mean additional efforts on system development. When multiple message sequences share a common subgoal, they lead to redundancy in the resulting system: multiple modules are created fulfilling a single goal. Sometimes, these different modules are necessary for fault-tolerance or users' convenience. But in most circumstances, one of them is enough.

Second, overlapping use cases may make the system difficult to maintain. When the system contains multiple implementations of the same goal, more efforts are obviously needed to modify these implementations whenever the requirements change. Furthermore, it is incredibly hard to find (prerequisite to modification) all the related message sequences manually (see Section 2.4 for details). If we fail to modify some of the implementations, it may cause a security hole in the modified system.

Third, overlapping use cases make the system behaviour confusing. When the system provides different modules for the same goal, what the system also provides is confusion for users who switch among these overlapping use cases.

Finally, overlapping use cases may frustrate use case proposers. When they propose a message sequence to achieve certain subgoal in a use case, they may believe that they have brought forward a unique way to do that whether within the use case or anywhere else. Forced to do that in other ways while executing other use cases, they may complain that their requirements are not satisfied.

2.3 Actions on overlapping use cases

Sometimes, we had better learn to live with overlapping use cases. Different users may insist on different ways of interacting with the system. In these cases, what we should do is to detect and track the overlapping use cases.

In other times, however, we may remove overlapping use cases by model refactoring. For example, interactions of registration in Fig. 1 should be extracted out as a stand-alone use case as described in Fig. 2.

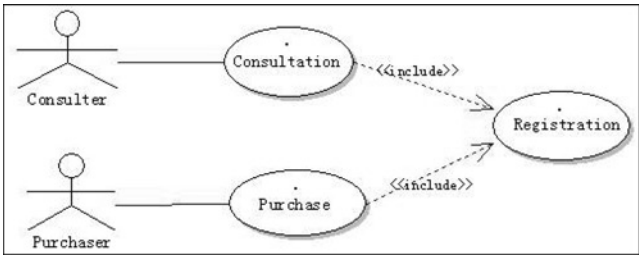

Fig. 2 *Refactored use cases*

Obviously, this kind of refactoring cannot be achieved during the preparation of requirements because these use cases are proposed by different groups who usually do not give enough attention to others' work. In fact, this kind of use case refactoring can only be based on successful detection of overlapping message sequences.

Note that even if refactoring is applicable, designers can decide to track and tolerate the overlaps. The decision usually depends on the policy of inconsistency and overlap management.

No matter what you would like to do with overlapping use cases (remove them by refactoring, or just record and track them), you have to detect them first. However, the detection of overlapping use cases is quite difficult.

2.4 Difficulty in detecting overlapping use cases

It is not easy to detect overlapping use cases from complicated use case models. Different proposers of use cases may use different message sequences to implement the same subgoal in different use cases (i.e. DSSG). If we take a close look at the use cases in Fig. 1, we can find obvious difference in the ways of registration in the two use cases. It makes the automation of this detection very difficult. Matching algorithms for identical message sequences fail to detect these message sequences because these message sequences are not identical (though they achieve the same subgoal). Furthermore, it is impractical to detect them by comparing goals explicitly attached to them because if the number of messages contained in a message sequence is N, the number of subsequences is $((N+1) \times N/2)$. It demands too much to explicitly attach goals to all of them.

3 Approach to detecting overlapping use cases

3.1 Overview

As discussed in Section 2.4, the major difficulty in detecting overlapping use cases (message sequences) is how to identify and compare goals of different message sequences. Users communicate with the system in order to obtain some meaningful outputs and/or transform the system from one state into another. So, goals may include system outputs and transition of system state. Outputs are specified explicitly in message sequences, but transitions of system state are not.

A scenario can be described with a sequence diagram, which in turn can be transformed into a statechart. There are quite a few approaches to transform sequence diagrams into statecharts [12–14]. Since a scenario is a single execution path, the corresponding statechart can be expressed as a transition path. After the description and transformation, the problem of overlapping use cases turns into the problem of overlapping transition paths (subpaths). Overlapping transition (sub-) paths transform the system from the same initial state to identical terminal states, and they produce the same meaningful outputs for actors. In other words, overlapping transition (sub-) paths achieve the same goals for users.

First, we bring forward a way to identify the goal of a transition path by its initial state, terminal state and critical output actions. Then we compare the goals of the resulting transition paths transformed from sequence diagrams and find out transition paths (and subpaths) that implement the same goal (or subgoal), that is overlapping transition paths. Tracing back from these paths (and subpaths), we find out overlapping message sequences (and thus overlapping use cases).

3.2 Goal identification and comparison

In order to detect overlapping use cases (overlapping message sequences), the most urgent task is to identify and compare goals of message sequences. A sequence diagram may contain a large number of subsequences, which implement subgoals. So, given a sequence diagram, goal identification methods should be able to identify goals for any subsequence of it.

Interacting with a system, users aim to obtain outputs from the system or/and transform the system from one state to another. For a transition path, state transition of the system could be identified by its initial and terminal states. Outputs produced by the system by performing output actions could be determined by the output actions and their environment (the system states). In a statechart, an output action is associated with a transition, and its environment is the initial state of the transition. We also note that only a small part of the outputs are directly related to users' goal or subgoal. Such kind of outputs are called critical outputs. Of course, users know what they expect from the system, and it is exactly the critical outputs. So, it is not difficult for users to tell which output is critical. Actions that produce critical outputs are identified as critical actions, and transitions associated with critical actions are called critical transitions.

For every critical action, we gather the tuple

$$\langle \textit{Critical action}, \textit{Initial state} \rangle$$

For a transition path, these tuples are gathered as a bag. It is a 'bag', not a 'set', because some tuples may be identical to each other. For example, if a critical action a_1 appears in two transitions from the same state s_1, the tuple $\langle a_1, s_1$ will appear twice in the resulting bag if both of the transitions are included in the transition path. Since the bag is enough to identify outputs of the transition path, we identify it as output bag. As a result, the goal of a transition path can be expressed with its initial state, terminal state and output bag. Subpaths of a transition path are transition paths too. So, their goals can be identified in the same way.

A scenario can be transformed into a transition path and share the same goal. A scenario is a single execution path of a use case, and it is usually described with a sequence diagram. With additional semantic information, a sequence diagram can be automatically transformed into a statechart [12–14]. The resulting statechart can be expressed as a transition path because a scenario is a single execution path. As a result, a scenario can be successfully transformed into a transition path. Furthermore, subsequences of a scenario (sequence diagram) can find their corresponding subpaths in the resulting transition path, and thus their goals can also be identified in the same way.

Besides goal identification, there is another problem to be solved: goal comparison. As discussed earlier, the goal of a transition path is identified with its initial state, terminate state and output bag. As a result, we have the following definition.

Definition 1 (equal transition paths) Transition Paths: Transition paths, tp_1 and tp_2, implement the same goal if all the following conditions are satisfied:

1. Initial state of tp_1 is identical to that of tp_2
2. Terminal state of tp_1 is identical to that of tp_2
3. The output bag of tp_1 is equal to that of tp_2.

Comparison of states is intuitive, and comparison of two bags (output bags) can be done with existing algorithms.

If two transition paths implement the same goal, we say that they are equal (as far as goals are concerned). It should be noted that identical paths are equal, but equal paths are not necessarily identical.

3.3 Generating statecharts

We describe each scenario with a sequence diagram so that the sequence diagram can be transformed into a transition path (see Section 3.2). In order to synthesise statecharts transformed from sequence diagrams, the system should be treated as a single object in the sequence diagrams. In other words, inner objects of the system will be ignored.

Many approaches have been proposed to transform sequence diagrams into statecharts [12–14]. In order to compare transitions of system state (part of goals), the transformation should be able to detect identical states among the resulting statecharts. In Whittle's method [12], global variables are introduced to identify identical states among statecharts, whereas the other two approaches leave the task to developers. So, we base our approach on Whittle's method. In the following paragraphs, we will give a brief introduction to the approach.

The first step of the method is to equip messages with pre- and post-conditions. Object Constrain Language (OCL), which is proposed by OMG to describe constraints in UML models, is suitable for this job. First, we should declare some global variables to present important aspects of the system. System states will be determined by the global variable, which are arrayed as a vector called state vector. And then, we use these global variables to describe pre- and post-conditions of messages. Suppose that a sequence diagram is represented as follows

$$\boldsymbol{S}_1 \overset{M_1}{\Longrightarrow} \boldsymbol{S}'_1, \ldots, \boldsymbol{S}_j \overset{M_j}{\Longrightarrow} \boldsymbol{S}'_j \tag{1}$$

where $\boldsymbol{S}_j$ and $\boldsymbol{S}_j'$ are states, M_j is a message and $\boldsymbol{S}_j \overset{M_j}{\Longrightarrow} \boldsymbol{S}'_j$ means that message M_j transforms the system from state $\boldsymbol{S}_j$ into another state $\boldsymbol{S}_j'$. As introduced above, states are depicted with state vectors, and we use $S_{j,i}$ to represent the value of the ith variable of the vector $\boldsymbol{S}_j$. v_i is the ith variable of the state vector.

The second step of the method is to extend state definition. First, we initialise state vectors from pre- and post-conditions in the following ways

1. If the precondition of message M_j says that $v_k = y$, then $S_{j,k} = y$.
2. If the postcondition of message M_j says that $v_k = y$, then $S'_{j,k} = y$.
3. If neither the pre-condition nor the post-condition of message M_j mentions v_k, $S'_{j,k} = S_{j,k} = ?$

where ? means undetermined. And then, these undermined vectors are determined in the following ways

1. Unification: if state vectors $\boldsymbol{S}_j$ and $\boldsymbol{S}_k$ ($j \neq k$) are unifiable, they are considered the same.
2. The frame axiom: if $\boldsymbol{S}_j[k] = ?$ and $j > 0$, then $\boldsymbol{S}_j[k] := \boldsymbol{S}'_{j-1}[k]$. If $\boldsymbol{S}'_j[k] = ?$, $\boldsymbol{S}'_j[k] := \boldsymbol{S}_j[k]$.

The third step is to generate statecharts. Messages directed towards an object are considered events in the statechart for the object. Messages directed away from it are considered actions.

The last step is to merge identical states. Two states are identical if they share the same state vectors and have at least one incoming transition with the same label.

For convenience, we express a sequence diagram as SD

$$\text{SD} = \{M, \text{MS}, E, A, \text{CA}, C, \text{PRE}, \text{POST}\} \tag{2}$$

where M is messages, MS is a sequence of messages in order, E is events (messages sent by users), A is actions performed by the system, CA is critical actions, which produce critical outputs, C is conditions which are referred as messages' pre- and post-conditions, PRE and POST are relationships between C and M indicating pre-conditions and post-conditions of messages, respectively.

With Whittle's method, which is introduced earlier, SD is transformed into a statechart ST

$$\begin{aligned}\text{ST} = \{&T, \text{CT}, S, E, A, \text{CA}, S_\text{i}, S_\text{t}\\ &\text{INI}, \text{TER}, \text{TE}, \text{TA}, \text{T\_PATH}\}\end{aligned} \tag{3}$$

where T is transitions, CT is critical transitions, S is states, E, A and CA are identical with those in SD, S_i is the initial state, S_t is the terminal state, INI and TER are relationships between transitions and states indicating initial states and terminal states of transitions, TE is relationships between T and E, TA is relationship between T and A. TE(t) is the event triggering the transition t, and TA(t) is the set of actions associated with t, T_PATH is a transition path corresponding to MS of its source sequence diagram SD.

$$\begin{aligned}&\text{STATE\_OR\_TRANSITION} ::= S | T\\ &\text{T\_PATH} : \text{seqSTATE\_OR\_TRANSITION}\end{aligned} \tag{4}$$

T_PATH is the set $\langle s_0, t_1, s_1, t_2, s_2, \ldots, s_n\rangle$. States and transitions appear alternately representing that transition t_i transforms the system state from s_i into s_{i+1} T_PATH of every statechart is gathered into STP (set of transition paths). CT is critical transitions, which can be obtained by formula (5)

$$\text{CT} = \{t \in T \mid \exists a \in A \bullet a \in \text{CA} \wedge a \in \text{TA}(t)\} \tag{5}$$

Identical states are collected as SISS (set of identical state sets)

$$\text{SISS} = \mathbb{P}S \tag{6}$$

In (6), every element of SISS is a set of identical states among statecharts.

Applying the approach to the example message sequences in Section 2.1, we generate the statecharts as shown in Fig. 3. The left statechart is generated from message sequence of Consulation, whereas the right one is generated from message sequence of Purchase. In a normal statechart, there should be an initial state. But the initial state Sb0 of the right statechart has been merged with Sb1 because as far as global variables (ID, Country, City and Name) are concerned, the two states are identical.

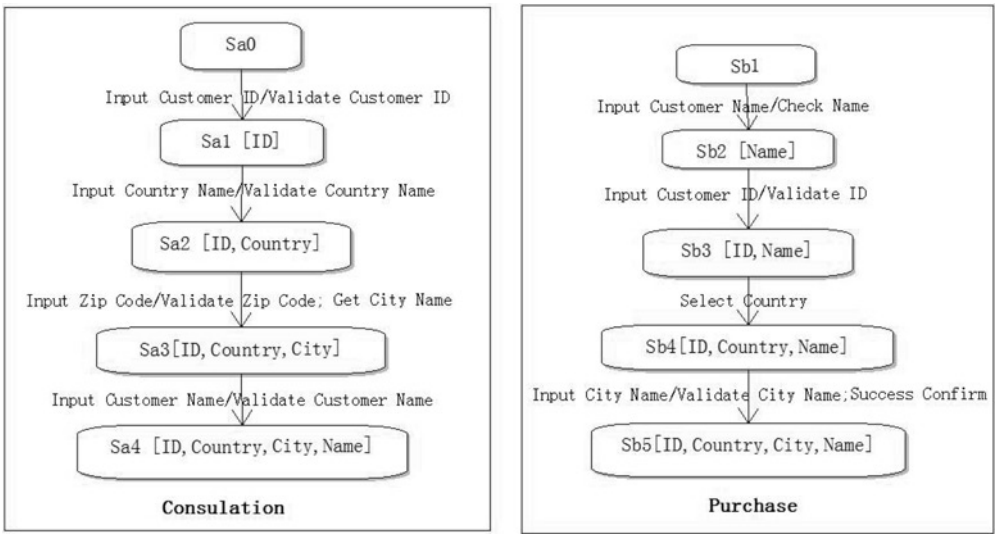

Fig. 3 *Statecharts generated from message sequences in Fig. 1*

Whittle's method [12] can automatically merge such kind of states. By comparing the states in the two statecharts, we also obtain SISS = {{Sa0, Sb1}, {Sa4, Sb5}}, which suggests that state Sa0 is identical to Sb1, and Sa4 is equal to Sb5.

3.4 Detecting overlapping transition paths

For each pair of elements ($IStates_1$, $IStates_2$) of SISS, we try to find out transition paths whose initial states and terminal states belong to $IStates_1$ and $IStates_2$, respectively. These transition paths share the same transition of system state because they have identical initial states and identical terminal states. We call these transition paths candidate transition paths. The following algorithm is presented to detect candidate transition paths and its explanation is given in the following paragraphs:

Algorithm for Detecting Candidate Transition Paths
Input: SISS, STP

Output: SCTP
SCTP = { }
for each pair of elements of SISS: ⟨$IStates_1$, $IStates_2$⟩
  set Current_Set empty
  for each element of STP: t_path
    set Current_Path empty
    for each element of t_path (in order): et
      if Current_Path is empty and et ∈ $IStates_1$
        add et to the tail of Current_Path
      end if
      if Current_Path is not empty and et ∈ $IStates_2$
        add et to the tail of Current_Path
        add Currrent_Path to Current_Set
        clean Currrent_Path
      end if
      if Current_Path is not empty and et ∉ $IStates_2$
        add et to the tail of Current_Path
      end if
    end for
    add Current_Set to SCTP
  end for
end for

We search every transition path (t_path) in STP (set of transition paths) in order, for the first state which belongs to $IStates_1$, record it and all the following elements in t_path until the first state which belongs to $IStates_2$. The resulting sequence of the recorded elements is a candidate transition path. After recovering the first candidate transition path, we try to detect other candidate transition paths in the remainder of t_path. All the candidate transition paths found for a pair of elements of SISS are gathered into a set which is added to SCTP (set of candidate transition paths) as an element. SCTP can be formalised as

$$\begin{aligned} SCTP = \{stp : set\ \text{STATE\_OR\_TRANSITION} \mid \\ (\forall tp \in stp \bullet (\exists IState_1, IState_2 \in SISS \\ \bullet tp(1) \in IState_1 \wedge tp(|tp|) \in IState_2 \\ \wedge (\exists t\_path \in T\_PATH \bullet tp \in t\_path))) \\ \wedge (\forall tp_1, tp_2 \in stp \bullet tp_1(1) = tp_2(1) \\ \wedge tp_1(|tp_1|) = tp_2(|tp_2|))\} \end{aligned} \quad (7)$$

An element of SCTP is a set of candidate transition paths with identical initial states and identical terminal states. So, Rule 1 and rule 2 of equal transition paths (see Definition1 for details) are automatically satisfied if goal comparison is applied to two candidate transition paths of the same element of SCTP. In order to check the third condition of Definition1, we gather output bags for these transition paths from the same element of SCTP, respectively, and compare the bags to see whether these bags are equal to each other or not. If some of them are equal, their corresponding transition paths are equal too. Transition paths achieving the same goal are identified as overlapping transition paths. Overlapping transition paths are gathered into SOTP (set of overlapping transition paths).

The generated statecharts shown in Fig. 3 have two transition paths: one begins at Sa0 and ends at Sa4; the other from Sb1 to Sb5. Since {Sa0, Sb1} ∈ SISS and {Sa4, Sb5} ∈ SISS, the two transition paths are candidate transition paths. Then, we compare the critical out bags of the two transition paths, discovering that neither of them has any critical output. So, we can determine that the two transition paths are overlapping transition paths.

3.5 Detecting overlapping message sequences

In the definition of T_PATH, relationship has been established between transition paths and message sequences. For a transition, its corresponding message sequence is ⟨triggering event, associated actions⟩. So, we collect such

message sequences for all transitions of an overlapping transition path, connect them together in the same order as transitions appear in the transition path, and we obtain the message sequence for the overlapping transition path. The algorithm is presented as follows:

Algorithm for Detecting Overlapping Message Sequences

```
Input: SOTP
Output: SOMS
SOMS = { }
for each element of SOTP: ctps1
    set Current_Set empty
    for each overlapping transitions path of ctps1: t_path
      set Current_MS empty
      for each element of t_path (in order): et
        if et ∈T
          add TE(et) to the tail of Current_MS
          add TA(et) to the tail of Current_MS
        end if
      end for
      add Current_MS to Current_Set
    end for
    add Current_MS to SOMS
  end for
```

With the algorithm, all message sequences achieving the same goal (i.e. overlapping message sequences defined in Section 1) are pulled together as an element of SOMS (set of overlapping message sequences).

3.6 Dealing with overlapping use cases

How to deal with the overlapping use cases is up to requirements engineers and designers. They may ask providers of these overlapping message sequences to negotiate with each other. If a unique message sequence is reached, replace all the overlapping message sequences with it by use case refactoring (remove overlapping use cases by refactoring) [6, 7]. If none of them will give up their expectations, requirements engineers should record the overlapping message sequences and their providers for possible requirements change (detect and track overlapping use cases).

The detection may have to be repeated for several times if model refactoring is adopted. Changing a scenario (an execution path of a use case expressed as a message sequence) may influence other scenarios of the use case. So, once a scenario is changed, we have to make corresponding change to other scenarios of the use case and mark them unchangeable. After solving other overlapping use cases, we will remove marks of unchangeable scenarios and apply detecting approach again to all scenarios. If no overlapping use case is found, detection is completed.

The solution to the overlapping message sequences in Fig.1 has been discussed in Section 2. Stakeholders reach an agreement on how to register. So, we can remove the overlapping use cases by model refactoring, and the resulting use case diagram is shown in Fig. 2.

4 Evaluation

The proposed approach has been supported by Detector, a plug-in of our modelling tool JBOO (Jade Bird Object-Oriented software modelling tool) [15]. Requirements engineers can draw use cases and sequence diagrams with modelling tool JBOO. And then, they can call the plug-in Detector to detect overlapping use cases. Fig. 4 illustrates the main interface of Detector. Each child of the project node in the left part of Fig. 4 is a group of overlapping use cases. The message fragments indicated by boxes in each diagram are overlapping sequences. Detector exchanges data with JBOO via XMI files. When designers call Detector to detect overlapping use cases, the modelling tool JBOO will automatically generate an XMI file and pass it to Detector. Since XMI is the standard format of UML models, Detector can also handle models built with other modeling tools. Though the previous versions of JBOO were released as products, the current version of JBOO is just a research project yet. Detector is developed by our research group recently, and has been checked and accepted as a part of JBOO by the sponsor of the project.

With the support of this tool; the proposed approach has been successfully applied to two genuine industrial projects: an office supplies a management system and an e-business system. The projects were developed by an e-business company, and the first author of the paper was a member of the development team as a senior software engineer. An overview of the evaluation projects is presented in Table 1. The evaluation is carried out by the authors and their research group. Four undergraduates and the first author play the roles of stakeholders of the evaluation

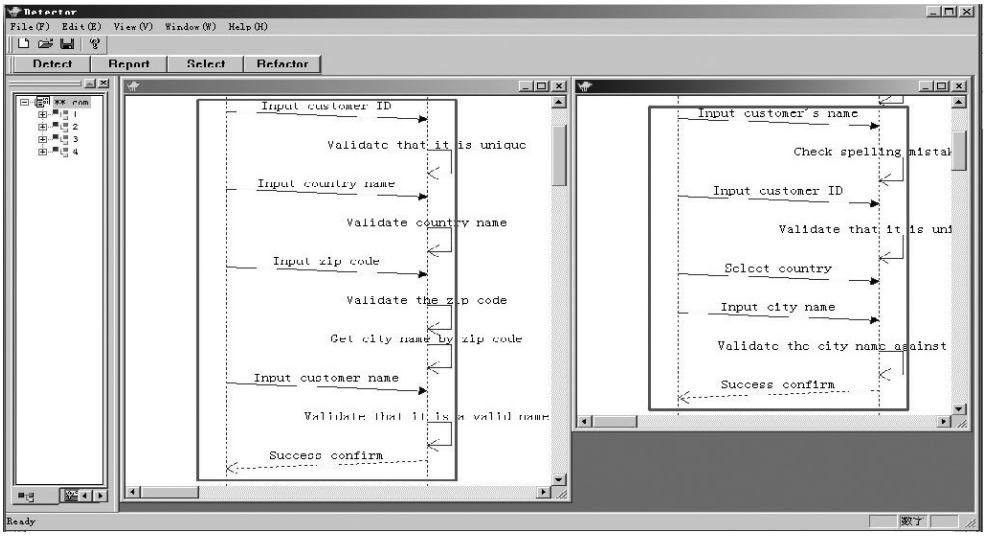

Fig. 4 *Detector*

Table 1: Evaluation projects

Project	Number of use cases	Number of events	Size of use case models, kb
Office supplies management system	18	225	2.14
E-business system	27	452	53.2

projects, whereas other seven graduates develop the evaluation tool and apply the proposed approach to the evaluation projects.

The office supplies management system (project one) is to manage delivery and procurement of office supplies as well as meeting room reservation. The system includes 18 use cases and all of them are specified with sequence diagrams. We apply the proposed detecting approach to these use cases and the results are presented in Table 2. Each row of the table is a group of message sequences which achieve the same goal or subgoal (i.e. a group of overlapping message sequences defined in Section 1). Column 'Number of overlapping message sequences' indicates how many message sequences each group has. 'Number of involved messages' is the sum of lengths of each message sequences in this group. For example, as shown in Table 2, the first group consists of 2 message sequences which achieve the same goal, and these 2 messages sequences are composed of 10 messages. In other words, the average length of each message sequence is five.

As shown in Table 2, two groups of overlapping message sequences are found out from project one: the first group consists of three message sequences which are made up of 10 messages. These message sequences achieve the same subgoal: request for procurement. One of the sequences comes from use case 'request for procurement', and the other is a part of use case 'apply for office supplies'. The second group has 2 message sequences whose average length (number of messages) is 8.5, and these 2 sequences share the same goal: registration. The messages are very similar to those in Fig. 1.

The e-business project (Project two) is to develop a website selling office supplies from copy paper, folders to personal computers and digital cameras. As a requirements model, the development team (one of the author is a member of the team) built a set of use cases which were described with sequence diagrams. We select 27 use cases, each containing at least one sequence diagram, for evaluation. In these sequence diagrams, we have done two experiments to find overlapping message sequences.

In the first experiment, we apply the approach directly to the selected use cases. The results are presented in Table 3 (its columns and rows have the the same meaning as those of Table 2). As shown in the table, we have found four groups of overlapping message sequences. And then, these message sequences are presented to stakeholders to make sure that they really implement the same goal (subgoal). After discussion, it is affirmed that all message sequences in each of the four groups implement the same goal.

Table 2: Evaluation results (project one)

	Number of overlapping message sequences	Number of involved messages	Average length of message sequences
1	2	10	5
2	2	17	8.5

Table 3: Evaluation results (project two)

	Number of overlapping message sequences	Number of involved messages	Average length of message sequences
1	2	18	9
2	6	27	4.5
3	2	16	8
4	2	29	14.5

With the detection results of the first experiment, overlapping message sequences are removed by use case refactoring. And then, the second experiment is done on the refactored use cases. First, we insert four different message sequences which implement the same goal (i.e. DSSG) into the message sequences manually and apply the approach to the revised message sequences again. The result is that all of the inserted sequences are detected (and nothing else is reported except the manually inserted sequences).

The results of the evaluation suggest that overlapping message sequences do exist in genuine projects, and the proposed approach can help to some extent in detecting them. It also suggests that it is possible that the approach can find out most of the overlapping message sequences, although it is not rigorously proved.

5 Related work

Yu *et al.* [6] brought forward the idea of use case refactoring as a part of model refactoring. One important use case refactoring they proposed is to remove duplications by reorganising use cases with 'include', 'extend' and 'generalisation'. However, in order to apply this kind of refactorings, overlapping use cases should be found out first. Unfortunately, they say nothing about how to detect overlapping use cases although they do have a detailed analysis on the mechanism for this refactoring.

Robinson and Woo [16] tried to retrieve reusable sequence diagrams from a library by matching action sequences. They did it by converting sequence diagrams into graphs, and applied SUBDUE [17] algorithm to these graphs to retrieve similar graphs or subgraphs. But unfortunately, it is not a good solution for overlapping use cases because of the following reasons. First, it does not take care of similar scenarios within the specification itself. Second, concept and relationship matching algorithms [16] aim to detect similar message sequences that have a similar structure. As a result, different sequences implementing the same goal would not be detected if they do not share similar structures. Third, the approach is not powerful and accurate enough because it does not take care of the order of messages. If there is more than one message between two lifelines, one cannot tell the relative order between them in the graph. As a result, 'false' matches would be reported.

6 Conclusions

The complexity of systems and scenario-based requirements engineering give rise to the problem of overlapping use cases, which is analysed in detail in this paper.

A special situation of overlapping use cases is introduced for the first time: different use cases provide different message sequences to achieve the same subgoal (DSSG). Existing inconsistency checking methods, refactoring of use case diagrams and scenario reuse do not take DSSG into consideration. Thus, they are incapable of solving the problem. It also explains the necessity to detect overlapping use cases.

A statechart-based approach is proposed to detect overlapping use cases. It describes use cases with sequence diagrams which are transformed into statecharts with additional semantic information. Goals are identified by transitions of system state and critical outputs which are produced by critical actions. By identifying identical states among statecharts, overlapping transition paths are gathered. They share the same transition of system state. If they produce the same output, they will have the same effect for users. Rules are provided to compare goals of two transition paths. From those transition paths that implement the same goals, overlapping message sequences are found.

Little additional work is needed for the detection. In many modern development processes (such as RUP [18]), the result of requirement capture workflow is usually a domain model and a set of use cases described with sequence diagrams. So, describing use cases in sequence diagrams is not a special need of detection, but a common task of requirements engineering. Transformation from sequence diagrams into statecharts is not a burden either [12]. It releases the work to draw statecharts manually which play a key role in system design. So, the transform is taken as a usual step in many modern development processes.

7 Acknowledgments

The authors thank the anonymous reviewers for their valuable comments and suggestions.

The work is funded by the National High-Tech Research and Development Plan of China, No. 2004AA112070, the National Grand Fundamental Research 973 Program of China, No. 2002CB312003, and the National Science Foundation of China, No. 60473064.

8 References

1 Brooks, E.P.: 'No silver bullet: essence and accidents of software engineering', *IEEE Computer*, 1987, **20**, (4), pp. 10–19
2 Jacobson, I., Christerson, M., Jonsson, P., and Oevergaard, G.: 'Object-oriented software engineering – a use case driven approach' (Addison-Wesley, 1992)
3 Cockburn, A.: 'Writing effective use cases' (Addison-Wesley, 2001)
4 Armour, F., and Miller, G.: 'Advanced use case modelling' (Addison-Wesley, 2001)
5 ptc/03-08-02: 'UML2 superstructure final adopted specification', 2003
6 Yu, W., Li, J., and Butler, G.: 'Refactoring use case models on episodes'. Proc. 19th Int. Conf. on Automated Software Engineering, Linz, Austria, September 2004, pp. 328–331
7 Xu, J., Yu, W., Rui, K., and Butler, G.: 'Use case refactoring: a tool and a case study'. Proc. 11th Asia-Pacific Software Engineering Conf., Busan, Korea, 2004, pp. 484–491
8 Lamsweerde, A.V., Darimont, R., and Letier, E.: 'Managing conflicts in goal-driven requirements engineering', *IEEE Trans. Softw. Eng.*, 1998, **24**, (11), pp. 908–926
9 Finkelstein, A.C.W., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B.: 'Inconsistency handling in multiperspective specifications', *IEEE Trans. Softw. Eng.*, 1994, **20**, (8), pp. 569–578
10 Grundy, J., Hosking, J., and Mugridge, A.W.: 'Inconsistency management for multiple-view software development environments', *IEEE Trans. Softw. Eng.*, 1998, **24**, (11), pp. 960–981
11 Richards, D., and Boettger, K.: 'Assisting decision making in requirements reconciliation'. Int. Conf. on Computer Supported Cooperative Work in Design (CSCWD 2002), Rio de Janeiro, Brazil, September 2002, pp. 25–27
12 Whittle, J., and Schumann, J.: 'Generating statechart designs from scenarios'. Proc. 22nd Int. Conf. on Software Engineering (ICSE 00), Limerick, Ireland, 2000, pp. 314–323
13 Ziadi, T., Helouet, L., and Jezequel, J.M.: 'Revisiting statechart synthesis with an algebraic approach'. Proc. 26th Int. Conf. on Software Engineering (ICSE 04), Edinburgh, United Kingdom, May 2004, pp. 242–251
14 Khriss, I., Elkoutbi, M., and Keller, R.: 'Automating the synthesis of UML statechart diagrams from multiple collaboration diagrams'. 1st Int. Workshop on The Unified Modeling Language (UML'98), Mulhouse, France, 1998, pp. 132–147
15 Zhi-Yi, M., Jun-Feng, Z., XiangWen, M., and WenJuan, Z.: 'Research and implementation of jade bird object-oriented software modeling tool', *J. Softw.*, 2003, **14**, (1), pp. 97–102 (in Chinese)
16 Robinson, W.N., and Woo, H.G.: 'Finding reusable UML sequence diagrams automatically', *IEEE Softw.*, 2004, **21**, (5), pp. 60–67
17 Jonyer, I., Cook, D.J., and Holder, B.L.: 'Graph-based hierarchical conceptual clustering', *J. Mach. Learn. Res.*, October 2001, **10**, (2), pp. 19–43
18 Kruchten, P.: 'The Rational Unified Process, an introduction' (Addison Wesley Longman, 2000, 2nd edn.)