Information and Software Technology 53 (2011) 666–681

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

# An approach to identifying causes of implied scenarios using unenforceable orders

In-Gwon Song *, Sang-Uk Jeon, Ah-Rim Han, Doo-Hwan Bae

*Department of Computer Science, College of Information Science and Technology, KAIST, Daejeon, Republic of Korea*

ARTICLE INFO

*Article history:*
Available online 24 November 2010

*Keywords:*
UML 2.0
Interaction overview diagram
Sequence diagram
Scenarios
Requirements verification
Implied scenarios

ABSTRACT

*Context:* The implied scenarios are unexpected behaviors in the scenario specifications. Detecting and handling them is essential for the correctness of the scenario specifications. To handle such implied scenarios, identifying the causes of implied scenarios is also essential. Most recent researches focus on detecting those implied scenarios, themselves or limited causes of implied scenarios.
*Objective:* The purpose of this research is to provide an approach to detecting the causes of implied scenarios.
*Method:* The scenario specification is a set of events and a set of relative orders between the events, and enforces them for its implementation. Among the orders, a set of orders that cannot be inherently enforced is the unenforceable orders. Obviously, existence of unenforceable orders leads the implied scenarios. To obtain the unenforceable orders, we first provide a method to represent each of the specification and its implementation as a set of orders between events, called the causal order graph. Then, the differences between them are the unenforceable orders.
*Results:* Because the unenforceable orders consist of events and their order relation that are specified in the scenario specification, they can point out which part of the scenario specification should be considered to handle the implied scenarios. In addition, our approach supports the synchronous, asynchronous, and FIFO communication styles without the state explosion or heavy computational overhead. To validate our approach, we provide two case studies.
*Conclusion:* This approach helps a designer to effectively correct the scenario specification by identifying where to be fixed, especially in large cases and under the various communication styles.


## 1. Introduction

Since the requirements specification determines the scope of subsequent development phases, its correctness is important. As scenarios become popular as a means of specifying the requirements [13], Message Sequence Charts (MSCs) [8] or Unified Modeling Language (UML) [12] interaction diagrams are widely used for the scenario descriptions because of their proper formality and ease of use. However, the scenario specification specified in the MSCs or UML interaction diagrams may cause differences between the specification and its implementation: the scenario specification can describe only partial behaviors of a system, while its implementation has full behaviors. Such differences are referred to as the implied scenarios [2,15]. More formally, the implied scenarios are differences between a scenario specification and its minimal implementation [15,11], which satisfies the scenario specification. In this paper, we refer to the minimal implementation as *implementation model*. Note that the implementation model is not the source code, but a smallest set of behaviors satisfying the specification. Therefore, the implied scenarios means inherent but unexpected behaviors, and they should be identified to obtain the correct scenario specification.

Several works have been proposed to detect the implied scenarios [9,11,15]. In these approaches, the model-checking technique is used through the synthesis of an automata-based implementation model. Such approaches have two limitations. First, the model-checking technique can only identify implied scenarios as a form of error traces. Such error traces do not indicate the locations in the scenarios specification where the designer should consider to handle the implied scenarios. We refer to such locations as the *causes of implied scenarios*. With the knowledge of the causes of implied scenarios, it certainly becomes more easier to handle the implied scenarios. Moreover, automatic identification of the causes enables (semi-)automatic treatment of the implied scenarios. Another limitation is that they assume only the synchronous communication style in the scenario specification. This is because handling the asynchronous communication style may lead to the state explosion in the synthesis of the automata-based implementation model. However, emerging software, such as web services and embedded software, requires the asynchronous communication style. Therefore, in the detection of the implied scenarios,

* Corresponding author. Tel.: +82 423507739; fax: +82 423508488.
*E-mail address:* igsong@gmail.com (I.-G. Song).

0950-5849/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2010.11.007

the consideration of the asynchronous communication style as well as the synchronous one is needed.

To address those limitations, in this paper, we provide an approach to detecting unenforceable orders to identify causes of implied scenarios. Essentially, the scenario specification enforces relative orders between events. If some orders among them cannot be enforced in an implementation, such orders cause implied scenarios. We call such orders the *unenforceable orders*. (We will give more detailed explanation on them in Section 5.1.) To detect the unenforceable orders, our approach creates a graph that represents the orders enforced by the scenario specification. Based on the graph, we also construct another graph representing the orders that are enforceable in the implementation model. Then, we calculate differences between those graphs. These represent the unenforceable orders.

Our approach has the following advantages:

- Since the unenforceable orders correspond to the events of the scenario specification, they can be used as the causes of implied scenarios, which indicate which part of the scenario specification should be considered to handle the implied scenarios.
- As our approach does not synthesize any automata-based model, it can handle the asynchronous communication without producing the state explosion. Based on the asynchronous communication, our approach can also handle the various communication styles, including the synchronous and FIFO communication style.
- Since the two graphs are based on the scenario specification and its implementation model, not only are the implied scenarios caused by the non-local choice detected, but also the input–output implied scenarios.
- Our approach provides the fine-grained detection because we separately use the sending and receiving events unlike previous approaches [9,11,15].
- Our approach is applicable for the large-scale scenario specification because the complexity of our algorithm is $O(|V_{spec}|^3)$ where $V_{spec}$ is a set of events in the scenario specification.

The rest of this paper is organized as follows: In Section 2, we discuss related works. Section 3 defines several terms and concepts. In Section 4, we explain the loop unrolling to deal with loops in the scenario specification. Section 5 defines the unenforceable orders, and presents the algorithms. In addition, the complexity of the algorithms is shown in order to show the efficiency of our approach. In Section 6, we present a technique to support loops and other communication styles. After presenting two case studies in Section 7, we conclude our approach with the discussion of future work in Section 8.

## 2. Related work

The term "implied scenario" was first introduced by Alur et al. [2]. They provided a framework to verify that a given scenario specification is realizable with some implementation. Their realizability is classified on two levels: weak and strong. The weak realizability is satisfied if a scenario specification has all the combinations of the local behavior of each process, while the strong realizability is satisfied if a scenario specification satisfies the weak realizability and if it is deadlock-free. If those realizabilities are not satisfied, the scenario specification has the implied scenarios. Their work focuses on determining whether a scenario specification has implied scenarios or not, and is limited to the specifications that specify finite system behaviors, which means that no loop is allowed in the specification.

Uchitel et al. provided the method and tool for detecting implied scenarios [15]. Their method can deal with infinite system behaviors which are expressed by loops of an high-level message sequence chart (hMSC). They presented an algorithm that builds the Labeled Transition System (LTS) behavior model as the implementation model. They also presented the way of obtaining differences between the MSC specification and its implementation model. Since they use the model-checking technique to obtain the differences, their algorithm can produce the implied scenarios in the form of error traces, so that their work cannot identify the causes of implied scenarios. In addition, their work is applicable only to the synchronous communication style and assumes the synchrony hypothesis, which means that there are no events between sending and receiving events of a message.

Letier et al. have extended Uchitel et al.'s work by considering an observation that the reception of a message cannot be controlled, but is monitored [9]. They refer to the implied scenarios, resulting from their approach, as the input–output implied scenarios. Their work still assumes the synchronous communication and synchrony hypothesis.

Muccini proposed another approach to detecting implied scenarios, not involving the use of the model-checking technique. Instead, it starts from the detection of the non-local choices. Then, by investigating the events occurring after the non-local choices, it detects the implied scenarios. Although this approach is not aimed at detecting the causes of implied scenarios, it can be used to identify which choices are problematic through the intermediate results. However, this approach does not touch upon the input–output implied scenarios and assume the synchronous communication and synchrony hypothesis.

Baker et al. proposed an approach to detecting and resolving semantic pathologies in UML sequence diagrams [5]. Based on a graph, which is generated from the UML sequence diagram and referred to as the causal order graph, they proposed the conditions for detecting the pathologies. The approach supports various communication styles. However, their work has three limitations. First, the approach cannot be used when the scenario specification has loops. Second, since the approach does not consider the state-merging effects, the detected pathologies may omit some implied scenarios. Third, the synchrony hypothesis is partially assumed.

Recently, in the service-oriented architecture area, several researches for detecting the implied scenarios were presented as a name of "check of local enforceability" [7,19]. However, those works only check whether the implied scenarios exist or not, like Alur et al.'s work [2].

## 3. Background

### 3.1. UML scenario specification

Our approach uses the scenario specification that is represented as UML interaction diagrams. We refer to it as the *UML scenario specification*. Since it has similar structures with the MSC scenario specification in [15], their formal definitions are nearly the same. In spite of that, we reformulate our UML scenario specification to clearly describe our approach. Our UML scenario specification consists of basic sequence diagrams (bSDs) and a basic interaction overview diagram (bIOD) which are simplified from original sequence diagrams and an interaction overview diagram in UML 2.0, respectively.

**Definition 1.** A basic sequence diagram $B$ is a structure $(O, M, L, loc, <)$, where:

- $O$ is a set of event occurrences. It consists of a set of sending occurrences $\mathcal{S}$ and a set of receiving occurrences $\mathcal{R}$.
- $M$ is a set of messages. A message $m$ is a structure $(s,r,n)$ such that $s \in \mathcal{S}, r \in \mathcal{R}$ and $n$ is the label of the message. In addition, we denote a labeling function $lbl$ such that $lbl(s) = lbl(r) = n$ for $(s,r,n) \in M$.
- $L$ is a set of lifelines.
- $loc : O \to L$ maps an event occurrence to a lifeline. $loc^{-1}(l)$ is a set of event occurrences on a lifeline $l$.
- $<$ is a set of total orders $<_l \subseteq loc^{-1}(l) \times loc^{-1}(l)$ between event occurrences. The orders depict an order relationship between two adjacent event occurrences.

**Definition 2.** The basic interaction overview diagram $I$ is a graph $(E,V)$, where $V$ is a set of vertices that consist of control vertices and vertices referencing bSDs, and $E \subseteq (V \times V)$ is a set of directed edges that represent control flows. The control vertices are categorized into *initial*, *final*, *decision*, and *merge*.

Sequence diagrams in UML 2.0 have combined fragments to describe control structures or provide restricted views. We omit the combined fragments in the definition of the bSD because such control structures can also be represented by control vertices of the bIOD, and the restricted views do not affect behaviors presented by the bSD.

A bIOD provides a means of describing control flows between bSDs. It is a graph that consists of the vertices referencing bSDs, control vertices, and edges representing the control flows between the vertices. The control vertices consists of *initial*, *final*, *decision*, and *merge*. In this paper, we do not use the fork and merge nodes. While the different lifelines may concurrently behave, those nodes may introduce another parallelism. There are no exact definitions of its semantics. To deal with those parallelisms, we should first define their formal semantics. Since that task is not within the scope of this paper, we assume that the fork and merge nodes are not used in our scenario specification. We also assume that there is only one initial vertex for simplicity.

Now we define the UML scenario specification. It consists of a set of bSDs, a bIOD, and reference relationships between them. The definition of a UML scenario specification is as follows:

**Definition 3.** A UML scenario specification is a structure $(\mathcal{B},\mathcal{I},ref)$, where $\mathcal{B}$ is a set of bSDs, $\mathcal{I}$ is a bIOD, and $ref$ is a mapping function from a referencing vertex in $\mathcal{I}$ to a referenced bSD in $\mathcal{B}$.

### 3.2. Causal order graph

The UML sequence diagram describes the sending and receiving events of the messages, and the orders between them. Those events and orders can be formalized as a partially ordered set. To represent the partially ordered set, many prior works use a directed graph. The graph is referred to as the *causal order graph*. In the causal order graph, each vertex represents the sending or receiving event in the sequence diagram, and each edge indicates that the event that corresponds to its target vertex should occur after the occurrence of the event that corresponds to its source vertex.

Fig. 1 shows an example bSD and its causal order graph. For a clear explanation, we denote the sending event and receiving event of a message as $s_n$ and $r_n$, respectively, where $n$ is an identifier for each message. In Fig. 1a, the events $s_1$ and $r_1$ are sending and receiving events for the message "questionnaire", respectively. Since a sending event should occur before its corresponding receiving event, the causal order graph has an edge from $s_1$ to $r_1$ as shown in Fig. 1b. The event $s_1$ is located in a place above the event $s_2$ in Fig. 1a. According to the semantics of the sequence diagram, this

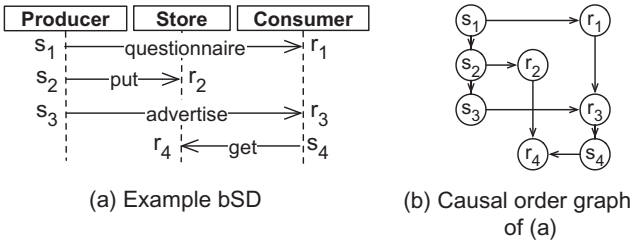


**Fig. 1.** Example of the causal order graph.

means that the event $s_2$ should arise after the event $s_1$. Hence, the causal order graph has an edge from the event $s_1$ to the event $s_2$. In this way, a causal order graph for a sequence diagram can be obtained. Its formal definition is as follows:

**Definition 4.** A causal order graph is defined as a directed graph $\mathcal{G} = (E,V)$. $V$ represents the set of vertices that represent events, while $E \subseteq V \times V$ represents a set of edges which denote orders between vertices.

## 4. Loop unrolling

When a system is described by the causal order graph and it has loops, the causal order graph also has loops. However, since the loops in the causal order graph represent concurrent events, the causal order graph is not appropriate for representing the UML scenario specification with loops. Therefore, we devised a loop-unrolling technique.

The basic idea for the loop-unrolling technique is as follows. Let's assume that the bSD $b$ has an unenforceable order and $b$ is in a loop. After the loop is unrolled, the bSD $b$ still has the unenforceable order.

In addition to the basic idea, we need to consider the concatenations between bSDs since the concatenations introduce another causal order relationship. For example, in Fig. 2, an event $s_4$ in the bSD "payStock" should happen after an event $r_1$ of the bSD "provide" because the bSD "payStock" is one of the next bSD of the bSD "provide" and $s_4$ and $r_1$ are on the same lifeline: the order "$r_1$ to $s_4$" is enforced. In order to prevent loss of such orders, we need to conserve them after the loop unrolling.

With this idea, we devise transformation templates of a generic loop as shown in Fig. 3. Fig. 3a shows the generic loop of the bIOD. Each circle means a subgraph of the bIOD. To make it as a simpler form, it is re-arranged as shown in Fig. 3b. Then, the re-arranged loop is unrolled as shown in Fig. 3c.

Before justifying our unrolling technique, let's define the causality relation $e \prec e'$ which means that the event $e$ should happen

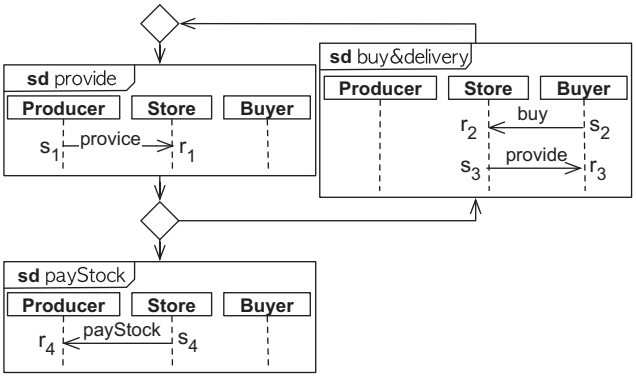


**Fig. 2.** Example for loop unrolling.

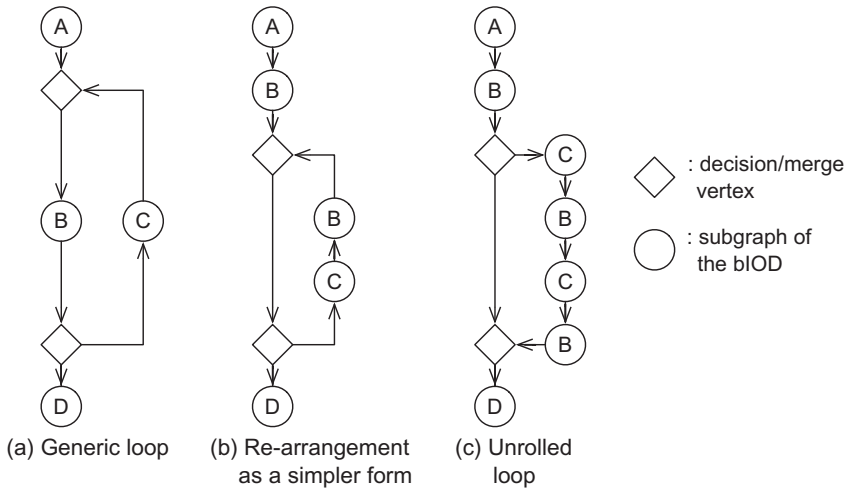


**Fig. 3.** Unrolling of generic loop.

before $e'$. Note that the causality relation $\prec$ has the same semantics with the causal order graph.

Now, we can show the validity of our unrolling technique with a simple argument. First, since Fig. 3c has all the bSDs in Fig. 3a, causality relations in each bSD are obviously preserved after the loop unrolling. Let $e_A$, $e_B$, $e_C$, and $e_D$ be events that are arbitrarily selected from A, B, C, and D in Fig. 3, respectively. Then, Fig. 3a directly shows the causality relations $e_A \prec e_B$,$e_B \prec e_D$, $e_B \prec e_C$, and $e_C \prec e_B$. By the transitiveness of the causality relation, the relations $e_A \prec e_B$, $e_A \prec e_C$, $e_A \prec e_D$, $e_B \prec e_B$, $e_B \prec e_C$, $e_B \prec e_D$, $e_C \prec e_B$, $e_C \prec e_C$, and $e_C \prec e_D$ are inferred. These causality relations are preserved in Fig. 3c. Therefore, Fig. 3c conserves all causality relations of Fig. 3a. This means that all orders that are enforced by an original UML scenario specification are preserved in the unrolled UML scenario specification. In addition, since Fig. 3a is a generic loop, our loop unrolling can be applied to all cases.

The form for the unrolled loop, shown in Fig. 3c, has more instances of "B" and "C" than the original one. For the optimization of our approach, we devised several techniques to reduce them. However, since the optimization is not in the scope of this paper, we will not describe them.

## 5. Detecting unenforceable orders

The unenforceable orders are relative orders that are enforced by the scenario specification, but cannot be enforced when the specification is implemented. Obviously, if they exist in a scenario specification, the implied scenarios also exist. Since they consist of the events that are specified in the scenario specification, they can be pointed out in the scenario specification itself. Therefore, the unenforceable orders can be used to identify the causes of implied scenarios. In our approach, the unenforceable orders are obtained by differentiation between two causal order graphs representing the scenario specification and its implementation model. We refer to the two causal order graphs as the *specification order graph* and the *implementation order graph*, respectively.

Now we briefly illustrate our idea of detecting the unenforceable orders with an example shown in Fig. 4. In Fig. 4a, the object "Producer" should send the message "provide" to the object "Store" before the object "Buyer" sends the message "buy". The "Buyer" cannot know whether the "Producer" already sent the message "provide" or not. Thus, the "Buyer" may send the message "buy" before the message "provide" is sent. By this reasoning, we can come up with an implied scenario as shown in Fig. 4b. This reasoning shows

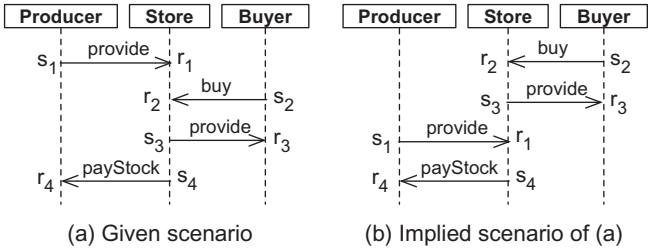


**Fig. 4.** Example bSD and its implied scenario.

another fact: the order "$r_1$ to $r_2$" is defined in the scenario specification, but not held in the implementation model. It means that the order "$r_1$ to $r_2$" is included in *specification order graph*, but not in the *implementation order graph*. According to the above definition, the order "$r_1$ to $r_2$" is an unenforceable order.

If the order can be enforced by introducing additional mechanisms, such as the coordinator, the implied scenario shown in Fig. 4b is also removed. Therefore, we can regard the unenforceable order "$r_1$ to $r_2$" as a cause of implied scenarios.

Fig. 5 illustrates the overview of our approach. Our approach consists of the following steps: (1) From the UML scenario specification unrolled by our loop-unrolling technique, the specification order graph is obtained. (2) Then, the implementation order graph is obtained by manipulating the specification order graph. (3) Finally, the differences between the specification order graph and

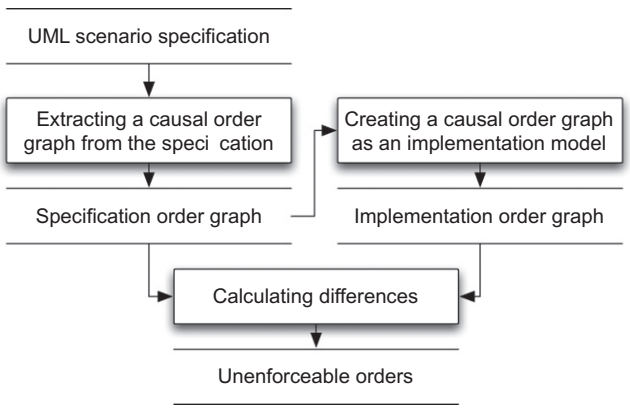


**Fig. 5.** Overview of our approach.

implementation order graph are calculated. For simplicity, in this section, we consider only the asynchronous communication style. The FIFO and synchronous communication styles will be covered in Section 6.

Now, according to the overview, we first define the enforceable orders with the definitions of the specification and implementation order graph in Section 5.1. Then, to show the efficiency of our approach, the algorithms for detecting the enforceable orders will be shown in Section 5.2.

### 5.1. Definitions

#### 5.1.1. Specification order graph

The specification order graph represents order relationships specified in the scenario specification. Since the UML scenario specification consists of bSDs and a bIOD, we first describe the definition of the specification order graph of a bSD. Then, we define the specification order graph of the whole UML scenario specification.

In Section 3, we briefly illustrated the causal order graph of the bSD. Now, we provide its formal definition.

**Definition 5.** The specification order graph $g^{B}_{spec}$ of a bSD $B = (O,M,L,loc,<)$ is defined as follows:

$$g^{B}_{spec} = (\{(b,e)|(b,e) \in <_l, {}^{\forall} l \in L\} \cup \{(s,r)|(s,r) \in M\}, O)$$

Fig. 6 shows a UML scenario specification of a small delivery system. The lower parts of Fig. 6, "**sd** deliveryA" and "**sd** deliveryB", are borrowed from Letier et al.'s work [9]. To effectively show our approach, we extend the example. The scenario specification in Fig. 6 describes the procedure of product sales with three bSDs and one bIOD. When the scenario begins, the delivery department transmits an order to the factory to produce a product. Then, the client requests a product from the seller with the type of the product. The seller who receives the request from the client transmits the request to the delivery department. Finally, the delivery department sends the requested product to the client. The specification order graphs for the bSDs in Fig. 6 are shown in Fig. 7.

Through the composition of the specification order graphs of bSDs, we can define the specification order graph for the entire scenario specification. We first define the composition operator between two bSDs. Then, by using it, the definition of the specification order graph for the entire scenario specification will be given. There are two kinds of composition: the asynchronous concatenation and synchronous concatenation [4]. The synchronous concatenation assumes the synchronization scheme, such as a coordinator or mutex, while the asynchronous concatenation does not. In this paper, the asynchronous concatenation is only used because we do not assume such a synchronization scheme. In the asynchronous concatenation, each lifeline is independently composed. To formalize the asynchronous concatenation, we devise an asynchronous concatenation operator $\oplus$. The $\oplus$ sequentially concatenates causal order graphs of two bSDs in an asynchronous manner. According to the definition of asynchronous concatenation [4], if $g_1 \oplus g_2$ is given and the $g_1$ and $g_2$ are causal order graphs, the asynchronous concatenation operator creates a new edge that connects the last event of $g_1$ to the first event of $g_2$ in each lifeline.

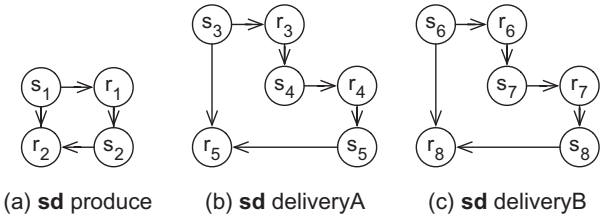


**Fig. 7.** Specification order graph of bSDs in Fig. 6.

**Definition 6.** Asynchronous concatenation operator $\oplus$ for the causal order graphs $g^{B_1}$ and $g^{B_2}$ is defined as follows:

$$g^{B_1} \oplus g^{B_2} = g^{B_1} \cup g^{B_2} \cup (E_{\oplus}, \emptyset)$$
$$E_{\oplus} = \{(max(g^{B_1}|_l), min(g^{B_2}|_l))|^{\forall} l \in L_{B_1,B_2}\}$$

where $L_{B_1,B_2}$ is a set of lifelines of $B_1$ and $B_2$, the union operator $\cup$ returns a graph that has all edges and vertices in both operands, and the projection operator $|_l$ results in a subgraph that only has vertices whose corresponding events are on a lifeline $l$ with edges connecting them. $min(g)$ and $max(g)$ functions return initial and final vertices of a graph $g$, respectively.

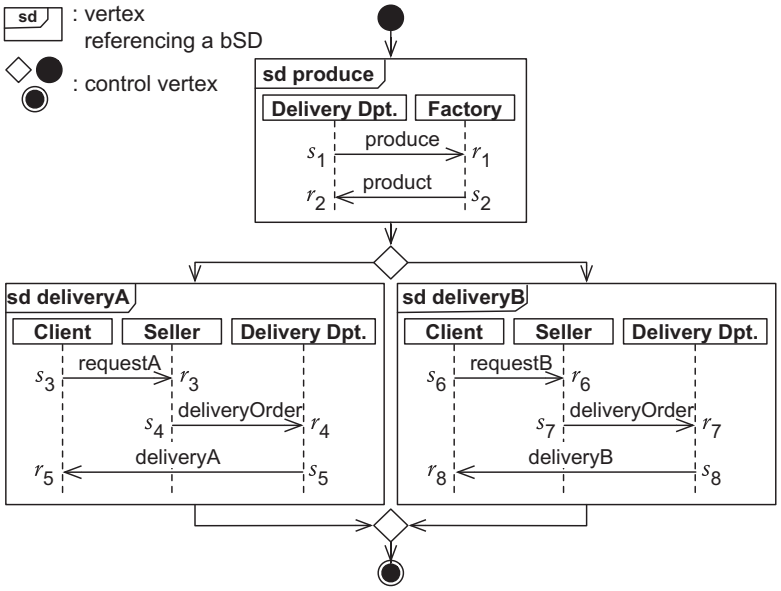


**Fig. 6.** Example scenario specification.

**Definition 7.** The specification order graph for UML scenario specification $\mathcal{U} = (\mathcal{B}, \mathcal{I}, ref)$ is defined as follows:

$$g^{\mathcal{U}} = \bigcup_{b_1 b_2 \ldots b_n \in P(\mathcal{I})|_{\mathcal{B}}} g^{ref(b_1)} \oplus g^{ref(b_2)} \oplus \cdots \oplus g^{ref(b_n)}$$

where $P(\mathcal{I})$ is a set of all paths from the initial vertex to all vertices in $\mathcal{I}$, and $|_{\mathcal{B}}$ is a projection operator that removes control vertices of a given path and keeps vertices referencing bSDs.

For example, in Fig. 6, there are two paths: "**sd** produce" to "**sd** deliveryA" and "**sd** produce" to "**sd** deliveryB". Thus, to obtain the specification order graph for the whole scenario specification, the asynchronous concatenation is used for composing those paths. For the path "**sd** produce" to "**sd** deliveryA", the lifeline "Delivery Dpt." exists in both the bSDs. The last event of "Delivery Dpt." is $r_2$ in the bSD "**sd** produce" and the first event of "Delivery Dpt." is $r_4$ in the "**sd** deliveryA". Thus, the asynchronous concatenation operator creates an edge from $r_2$ to $r_4$. Similarly, the events $r_2$ and $r_7$ are connected according to the path "**sd** produce" to "**sd** deliveryB". Consequently, we can obtain the specification order graph as shown in Fig. 8 for the scenario specification shown in Fig. 6.

### 5.1.2. Implementation order graph

The implementation order graph is an implementation model represented as a form of the causal order graph. From the literature [18,3,9], we have observed that the three properties of the implementation make the implementation model different from the scenario specification: (1) From the viewpoint of each lifeline, the states that are reached by the same sequence of receiving events are unified [18,3]; (2) the receiving events cannot be controlled, but can be monitored [9]; and (3) lifelines may make different decisions on the non-local choice [6]. To reflect the properties in the implementation order graph, the first and third properties are reflected by the addition of *permutating orders* and *non-local choice orders*, respectively. The second one is handled by the removal of *uncertain orders*. In this subsection, we will explain such orders and present the definition of the implementation order graph.

Here, we first explain the permutating orders with an example shown in Fig. 6. In Fig. 6, the object "Delivery Dpt." falls into the same state when the event $r_4$ or $r_7$ occurs because it receives the same sequence of the messages ("product" and "deliveryOrder"). In the both cases, the object "Delivery Dpt." can send the message "deliveryA" or "deliveryB" arbitrarily. This means that not only $s_5$, but also $s_8$, can occur after $r_4$. Similarly, $s_5$ can occur after $r_7$. However, the specification order graph in Fig. 8 does not have the order "$r_4$ to $s_8$" or "$r_7$ to $s_5$". Thus, to obtain the implementation order graph, those orders should be added to the specification order graph. Such orders are the *permutating orders*. Fig. 9 presents the specification order graph shown in Fig. 8 to which the permutating orders are added. In Fig. 9, the bolded lines are the permutating orders. In general, when two or more events $e_1, e_2, \ldots, e_n$ lead an object to the same state $s$ and their very next events $e'_1, e'_2, \ldots, e'_k$ are specified in the scenario specification, the object in the state $s$ can proceed to any of $e'_1, e'_2, \ldots, e'_n$ regardless of other conditions.

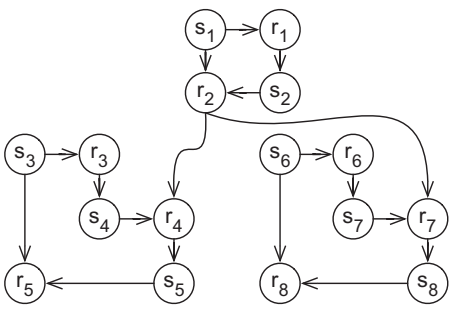


**Fig. 8.** Specification order graph of Fig. 6.

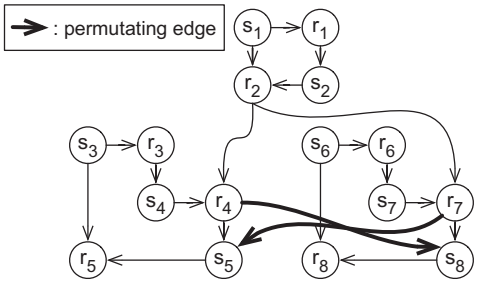


**Fig. 9.** Permutating orders with Fig. 8.

**Definition 8.** For the specification order graph $g^{\mathcal{U}}_{spec} = (E, V)$, the permutating orders are defined as follows:

$$E^{\mathcal{U}}_{pm} = \{(v_n, v'_{m+1}) | h(v_n) = h(v'_m), h(v_{n+1}) \neq h(v'_{m+1}), (v_n, v_{n+1}), (v'_m, v'_{m+1}) \in E, v_n, v'_m \in \mathcal{R}\}$$
$$h(v_n) = \{lbl(v_1) lbl(v_2) \ldots lbl(v_n) | v_1 v_2 \ldots v_n \in p(v_n)\}$$

where $p(v)$ is a set of paths from initial vertex to a vertex $v$, $\mathcal{R}$ is a set of receiving events, and $lbl(v)$ returns a message label corresponding to $v$.

In the above definition, we added the condition " $h(v_{n+1}) \neq h(v'_{m+1})$" to exclude useless edges. For instance, if the sending events $s_5$ and $s_8$ send the same messages in Fig. 9, regardless of inserting the orders " $r_4$ to $s_8$" and "$r_7$ to $s_5$", the behaviors of Fig. 9 are the same as in Fig. 8.

Second, we present the uncertain orders. In the implementation model, the receiving events cannot be controlled, but only monitored in an object [9]. This means that an object cannot control receiving events, so they can only be controlled by the messages sent from other objects. Therefore, if the specification order graph has an order enforcing that a receiving event occur after another event and those events exist in the same lifeline, then the order may not be preserved in the implementation order graph. Such orders are the *uncertain order*. To obtain the implementation order graph, the uncertain orders are removed from the specification order graph.

**Definition 9.** The uncertain orders for the specification order graph $g^{\mathcal{U}}_{spec} = (E, V)$ are defined as follows:

$$E^{\mathcal{U}}_{uc} = \{(v_1, v_2) | loc(v_1) = loc(v_2), (v_1, v_2) \in E, v_2 \in \mathcal{R}\}$$

where $\mathcal{R}$ is a set of the receiving events

It is worth explaining that the removal of the uncertain orders effectively reflects the characteristics of the receiving events. Note that the receiving events cannot be controlled by an object itself, but can be controlled by messages sent from another object. The removal of the uncertain orders makes the receiving events free in an object, but the receiving events should arise after their corresponding sending events. Consequently, when we remove the uncertain orders, the characteristics are reflected. For example, Fig. 10 shows the implementation order graph that is obtained by adding the permutation orders and removing the uncertain orders. With the removal of the uncertain orders, the orders "$s_1$ to $r_2$", "$s_3$ to $r_5$", "$r_2$ to $r_4$", "$r_2$ to $r_7$" and "$s_6$ to $r_8$" are removed. With the removal of the order "$s_1$ to $r_2$", the object "Delivery Dpt." shown in Fig. 6 cannot solely control the receiving event $r_2$. However, the event $r_2$ can still be controlled by the message "product", which is

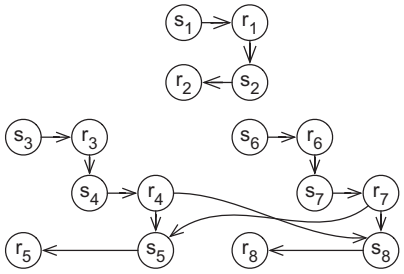


**Fig. 10.** Implementation order graph of Fig. 6.

- The two events should be on neither the same lifeline nor the same bSD because if the two events are on the same lifeline or the same bSD, those two events can be determined by a lifeline locally.

From the conditions, we can define the non-local choice orders as follows.

**Definition 10.** The non-local choice orders for the specification order graph $\mathcal{U} = (\mathcal{B}, \mathcal{I}, ref)$ are defined as follows:

$$E^{\mathcal{U}}_{nl} = \left\{ (v_1, v_2) | loc(v_1) \neq loc(v_2), v_1, v_2 \in \mathcal{S}, bsd(v_1)(v_2) v_1 = min(g^{\mathcal{U}}_d|_{l_1}), v_2 = min(g^{\mathcal{U}}_d|_{l_2}), g^{\mathcal{U}}_d = \text{suffix}(\mathcal{U}, d) l_1 \neq l_2, {}^{\forall} l_1, l_2 \in L, {}^{\forall} d \in \mathcal{D} \right\}$$

$$\text{suffix}(\mathcal{U}, \mathbf{d}) = \bigcup_{b_1 \ldots b_{k-2} \mathbf{d} b_k \ldots b_n \in P(\mathcal{I})|_{\mathcal{B}}} g^{ref(b_k)} \oplus \ldots \oplus g^{ref(b_n)}$$

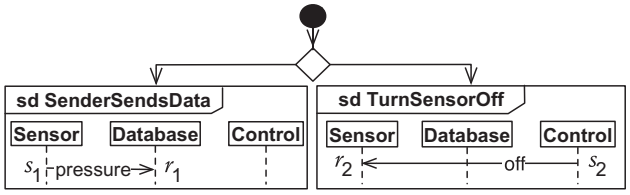


**Fig. 11.** Example for the non-local choice.

represented as the order "$s_2$ to $r_2$" in Fig. 10. This example shows that the characteristics of the receiving events are reflected in the removal of the uncertain orders.

The non-local choice is the last characteristics of the implementation model. Each object that has control can independently and concurrently behaves in an implementation model. Therefore, when each object arrives at a decision node of the interaction overview diagram, it chooses its direction if it has control, regardless of other objects' decisions.

Fig. 11 shows a simple example of the non-local choice. In the example, two mutually exclusive cases of scenarios are described: in one case, the "Sensor" sends the message "pressure"; in the other case, "Control" sends the message "off". On the other hand, with respect to the implementation model, both the "Sensor" and "Control" can send the message "pressure" and "off", respectively.[1] Therefore, although those two cases are mutually exclusive, the events $s_1$ and $s_2$ may occur at the same time. This is the non-local choice.

To represent the non-local choice, we insert orders between events such as $s_1$ and $s_2$ in Fig. 11. Such events should satisfy the following conditions.

- With the non-local choice, some events, located right after a decision node, are decided. Thus, each event is the very first event that occurs for each lifeline after a decision node
- Both events should be sending events. Receiving events are not decided by each lifeline itself, but by a lifeline that has opposing sending events.

[1] The "Sensor" and "Control" have controls and there are neither communications nor coordinators between them. Note that a sending event can be controlled by a lifeline which has the sending event.

Finally, using the permutating orders, uncertain orders, and non-local choice orders, we can give the definition of the implementation order graph as follows:

**Definition 11.** The implementation order graph for the specification order graph $g^{\mathcal{U}}_{spec} = (E, V)$ is defined as follows:

$$g^{\mathcal{U}}_{impl} = \left( E \setminus E^{\mathcal{U}}_{uc} \cup E^{\mathcal{U}}_{pm}, V \right)$$

where $\setminus$ is the set minus.

With the above definitions, the implementation order graph of Fig. 6 is given in Fig. 10.

*5.1.3. Unenforceable orders*

The unenforceable orders are identified by differentiating between the specification order graph and the implementation order graph. Essentially, both graphs represent partially ordered sets with transitiveness. Thus, to obtain valid differences between them, the transitive closures of both graphs should be used in the differentiation. For example, in the first column of the table in Fig. 12, the differences between the specification order graph and the implementation order graph in the original graph are the orders "$r_1$ to $r_2$" and "$s_2$ to $r_3$". However, the order "$s_2$ to $r_3$" is transitively satisfied in the implementation order graph. To avoid this, the transitive closures of both graphs should be used. However, a

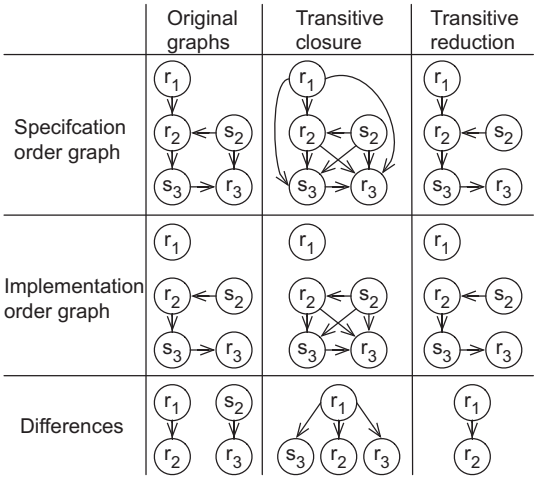


**Fig. 12.** Transitive closure and reduction.

problem still remains. As shown in the second column of the table in Fig. 12, the orders "$r_1$ to $s_3$", "$r_1$ to $r_2$" and "$r_1$ to $r_3$" are the differences between the transitive closures of the two graphs. Only the order "$r_1$ to $r_2$" is an essential cause because the absence of the order "$r_1$ to $r_2$", which is originally given by the specification, leads to the absence of the orders "$r_1$ to $s_3$" and "$r_1$ to $r_3$" as a result of the transitive closure. To minimize the result, we adopted the transitive reduction. It produces a minimal graph that preserves the same partial orderedness in the original one. With the transitive reduction, we can obtain the final result as shown in the third column of the table in Fig. 12.

However, the transitive reduction causes another problem although, in Fig. 12, the differentiation between the transitive reductions of the specification order graph and implementation order graph correctly identifies the unenforceable orders. When the transitive reduction removes an order from a graph and does not remove the order from another graph, the order is identified as one of the unenforceable orders. Since the orders that are removed by the transitive reduction are always conserved by remaining orders, the causes that correspond to such orders are invalid. Thus, we calculate the differences between the transitive closure and transitive reduction of graphs in order to obtain the unenforceable orders.

**Definition 12.** The unenforceable orders for UML scenario specification $\mathcal{U}$ are defined as follows:

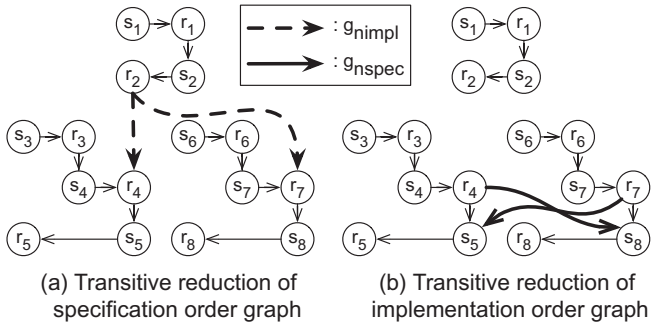


**Fig. 13.** Transitive reductions of Figs. 8 and 10.

$$g^{\mathcal{U}}_{cause} = g^{\mathcal{U}}_{nspec} \cup g^{\mathcal{U}}_{nimpl}$$
$$g^{\mathcal{U}}_{nimpl} = tr(g^{\mathcal{U}}_{spec}) \setminus (g^{\mathcal{U}}_{impl})^*$$
$$g^{\mathcal{U}}_{nspec} = tr(g^{\mathcal{U}}_{impl}) \setminus (g^{\mathcal{U}}_{spec})^*$$

where $tr$ is a transitive reduction and $^*$ is a transitive closure. The set minus $g_1 \setminus g_2$ between the graphs $g_1 = (E, V)$ and $g_2 = (E', V)$ is defined as $(E \setminus E', V)$.

Due to the semantics of the set minus $\setminus$, the unenforceable orders have two components: $g^{\mathcal{U}}_{nimpl}$ and $g^{\mathcal{U}}_{nspec}$. Interestingly, the two components have different semantics: $g^{\mathcal{U}}_{nimpl}$ represents orders that are enforced in the scenario specification, but that cannot be enforced in the implementation model, and $g^{\mathcal{U}}_{nspec}$ represents orders that should not be enforced, but that are held in the implementation model. Therefore, when we interpret the unenforceable orders, such a difference should be considered.

Now, we describe the unenforceable orders detected in the example shown in Fig. 6. Fig. 13 shows the transitive reductions of the specification and implementation order graphs shown in Figs. 8 and 10. Since the example is small, in Fig. 13, we can easily figure out differences between them; the bold solid lines represent $g^{\mathcal{U}}_{nspec}$ and the bold dashed lines are $g^{\mathcal{U}}_{nimpl}$. Overlapping the unenforceable orders with the given scenario specification helps to easily understand them. Fig. 14 shows such an overlapped view. According to the semantics of $g^{\mathcal{U}}_{nspec}$ and $g^{\mathcal{U}}_{nimpl}$, we can interpret Fig. 14 as follows: (1) "Delivery Dpt." may receive a message "deliveryOrder" before the message "product" arrives even though the scenario specification enforces that the message "deliveryOrder" is received after receiving the message "product". (2) Although the execution flow branches out of the "**sd** deliveryA" scenario, "Delivery Dpt." may send the "deliveryB" message. (3) Although the execution flow branches out of the "**sd** deliveryB" scenario, "Delivery Dpt." may send the "deliveryA" message.

### 5.2. Algorithms

In this section, we provide algorithms for detecting unenforceable orders and analyze their complexity. Through the complexity, we want to show that our approach can be applied in the large-scale scenario specification and is more efficient than detecting the implied scenarios. According to the definition of the unenforceable orders, we first describe algorithms for creating a specification

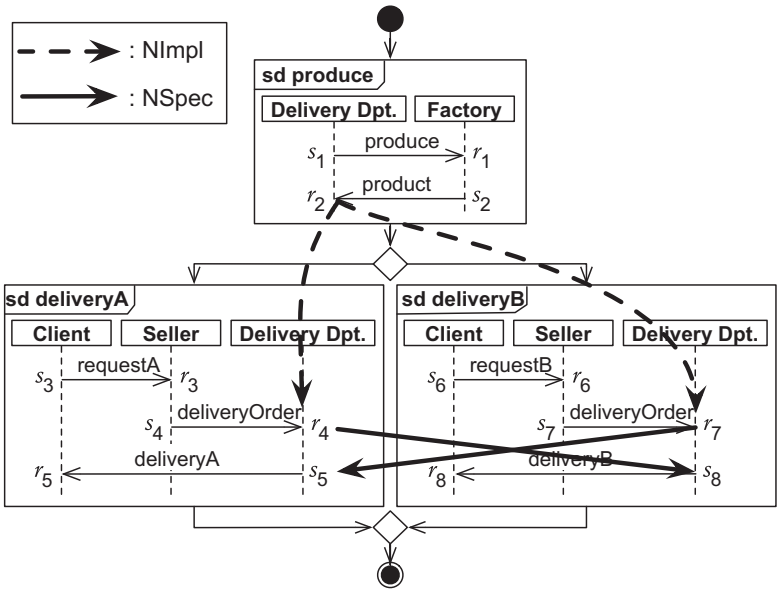


**Fig. 14.** Unenforceable orders in Fig. 6.

order graph and an implementation order graph. Then, the procedure for differentiating those two graphs will be shown.

Procedure 1 shows an algorithm for creating a specification order graph. The procedure firstly removes control nodes from a bIOD: line 2. Then, it creates a specification order graph for each bSD: lines 5–16. At last, those specified properties of bSDs are asynchronously concatenated: lines 17–21. In the procedure, there are four undefined procedures: "removeControlNodes", "removeTau", "$min_\tau$", and " $max_\tau$". "removeControlNodes" and "removeTau" procedures remove control nodes of a bIOD and $\tau$ from a specification order graph, respectively. By modifying the transitive closure algorithm [17], we can easily implement them with worst-case complexity $O(|V_\mathcal{I}|^3)$ and $O(|V_{spec}|^3)$, respectively. The "$min_\tau$" and "$max_\tau$" procedures return the first and last vertices of a given causal order graph if there is a vertex, respectively. Otherwise, they return $\tau$. Note that, without the $\tau$, the asynchronous concatenation operator cannot be run, correctly. For example, if bSDs *A*,*B*, and *C* should be concatenated, and if the *B* does not have an event for lifeline *l*, a last event on *l* in *A* is not concatenated with a first event on *l* in *C* without $\tau$.

**Procedure 1.** [createSpecificationOrderGraph]

**Require**: $\mathcal{U}$: UML scenario specification
1: $(\mathcal{B},\mathcal{I},ref) \leftarrow \mathcal{U}$
2: $(E_\mathcal{I}, V_\mathcal{I}) \leftarrow$ removeControlNodes($\mathcal{I}$)
3: $E_{spec} \leftarrow \emptyset$
4: $V_{spec} \leftarrow \emptyset$
5: **for all** $v \in V_\mathcal{I}$ **do**
6: $(O_b,M_b,L_b,loc_b, <_b) \leftarrow ref(v)$
7: **for all** $l \in L_b$ **do**
8: **for all** $(b,e) \in <_b|_l$ **do**
9: $E_{spec} \leftarrow E_{spec} \cup \{(b,e)\}$
10: **end for**
11: **end for**
12: **for all** $(s,r) \in M_b$ **do**
13: $E_{spec} \leftarrow E_{spec} \cup \{(s,r)\}$
14: **end for**
15: $V_{spec} \leftarrow V_{spec} \cup O_b$
16: **end for**
17: **for all** $(v_1, v_2) \in E_\mathcal{I}$ s.t. $ref(v_1) = (O_1,M_1,L_1,loc_1, <_1), ref(v_2) = (O_2,M_2,L_2,loc_2, <_2)$ **do**
18: **for all** $l \in L_1 \cup L_2$ **do**
19: $E_{spec} \leftarrow E_{spec} \cup \{(max_\tau(<_1|_l), min_\tau(<_2|_l))\}$
20: **end for**
21: **end for**
22: **return** (removeTau($E_{spec}$), $V_{spec}$)

With appropriate data structures, the complexity of Procedure 1 is bound to $O(|V_\mathcal{I}||L_b||<_b|_l| + |E_\mathcal{I}||L_1 \cup L_2| + |V_\mathcal{I}|^3 + |V_{spec}|^3)$: the first and second terms correspond to the first **for all** (lines 5–16) and second **for all** (lines 17–21), respectively, and the third and fourth terms correspond to "removeControlNodes" and "removeTau", respectively. Roughly, the complexity of Procedure 1 is abstracted to $O(|V_{spec}|^3)$.

**Procedure 2.** [prepareHistory]

**Require**: $(E_{spec},V_{spec})$: Specification order graph
1: $h \leftarrow$ empty history function
2: **for all** $v \in V_{spec}$ **do**
3: $h(v) \leftarrow (\emptyset, lbl(v))$
4: **end for**
5: **for all** $l \in L_{total}$ **do**
6: **for all** $(v_1, v_2) \in E_{spec}$ **do**
7: **if** $v_1$ and $v_2$ is on *l*, and $v_2$ is receiving event **then**
8: $(H, lb) \leftarrow h(v_2)$
9: $h(v_2) \leftarrow (H \cup \{h(v_1)\}, lb)$
10: **end if**
11: **end for**
12: **end for**
13: **return** *h*

Procedure 2 returns a history function, which returns states of each vertex of the specification order graph. This procedure is needed to obtain the permutating orders.

It begins by setting a history function of each vertex to an empty set: lines 2–4. Then, for each lifeline *l* and each edge $(v_1,v_2)$, if a vertex $v_1$ is a predecessor of $v_2$ on the same lifeline *l*, then $v_1$'s history is added to the history of $v_2$: lines 5–12.

The complexity of Procedure 2 is roughly bound to $O(|V_{spec}|^2)$ because $|L_{total}| \leqslant |V_{spec}|$ and $|E_{spec}| \leqslant |V_{spec}|$.

**Procedure 3.** [firstReactEvent]

**Require**: *v*: a vertex of a bIOD, *l*: a lifeline, *V*: a set of visited vertices
1: *r*← empty set
2: **if** $v \notin V$ **then**
3: $V \leftarrow V \cup \{v\}$
4: **if** *v* is not a control node and $ref(v)$ has an event on the lifeline *l* **then**
5: $(O_b,M_b,L_b,loc_b, <_b) \leftarrow ref(v)$
6: **if** $min(<_b|_l)$ is a sending event **then**
7: $r \leftarrow r \cup \{min(<_b|_l)\}$
8: **end if**
9: **else**
10: **for all** $v' \in$ a set of next nodes of *v* **do**
11: $r \leftarrow r \cup$ firstReactEvent($v'$,*l*)
12: **end for**
13: **end if**
14: **end if**
15: **return** r

Procedure 3 is a helper procedure for detecting non-local choice orders. It returns a first event for the given lifeline after the given vertex, and is bound to $O(|V_l| + |E_l|)$ since it is a kind of a Depth First Search (DFS) algorithm.

**Procedure 4.** [detectNonLocalChoiceOrder]

**Require**: $\mathcal{U}$: UML scenario specification, $E_{impl}$: Set of edges of implementation order graph
1: $(\mathcal{B},\mathcal{I},ref) \leftarrow \mathcal{U}$
2: $(E_\mathcal{I}, V_\mathcal{I}) \leftarrow \mathcal{I}$
3: $L_{total} \leftarrow$ union of all lifelines in $\mathcal{B}$
4: $D_\mathcal{I} \leftarrow$ a set of decision vertices in $V_\mathcal{I}$
5: **for all** $v \in D_\mathcal{I}$ **do**
6: $S \leftarrow$ empty set
7: **for all** $l \in L_{total}$ **do**
8: $S_l \leftarrow$ empty set
9: $N_v \leftarrow$ a set of next nodes of *v*
10: **for all** $v' \in N_v$ **do**
11: $e \leftarrow$ firstReactEvent($v'$,*l*)

```
12:          S_l ← S_l ∪ e
13:          for all s ∈ S do
14:            E_impl ← E_impl ∪ {(s,e)}
15:            E_impl ← E_impl ∪ {(e,s)}
16:          end for
17:        end for
18:        S ← S ∪ S_l
19:      end for
20:    end for
21:    return E_impl
```

Procedure 4 is a procedure for creating the non-local choice orders. For each decision node and each lifeline, it finds a first sending event through Procedure 3. Then, edges between every pair of the events is added to the implementation order graph, except in the case in which a pair of events is on the same lifeline or in the same bSD.

The complexity of Procedure 4 is bound to $O(|D_{\mathcal{I}}||L_{total}||N_v|(|S|+|E_{\mathcal{I}}|+|V_{\mathcal{I}}|))$, and $O(|V_{spec}|^3)$ can be roughly used as a concise form of the complexity.[2]

**Procedure 5.** [createImplementationOrderGraph]

```
Require: U: UML scenario specification,
  (E_spec,V_spec):specification order graph
1:   E_impl ← ∅
2:   for all (v1,v2) ∈ E_spec do
3:     if v2 is a sending event or (v1,v2) is originated from a
     message then
4:       E_impl ← E_impl ∪ {(v1,v2)}
5:     end if
6:   end for
7:   h← prepareHistory(g_spec)
8:   for all v1 ∈ V_spec do
9:     for all v2 ∈ V_spec do
10:      if h(v1) = h(v2) and v1 ≠ v2 then
11:        for all v3 ∈ V_spec and (v2,v3) ∈ E_spec do
12:          if v3 and v2 is on the same lifeline then
13:            E_impl ← E_impl ∪ {(v1,v3)}
14:          end if
15:        end for
16:      end if
17:    end for
18:  end for
19:  E_impl← detectNonLocalChoiceOrder(U, E_impl)
20:  E_impl
```

Procedure 5 presents an algorithm for creating the implementation order graph. First, in lines 2–6, the uncertain orders are removed from the specification order graph. Before adding the permutating orders, the procedure prepares the history function using Procedure 2. Then, in lines 8–18, the permutating orders are added to the implementation order graph: for every pair of distinct events $v_1$ and $v_2$, if they have the same history, edges between them their very next events are inserted. Note that we do not need to return $V_{impl}$ like Procedure 1 because $V_{impl}$ is the same with $V_{spec}$. The complexity of Procedure 5 is clearly $O(|V_{spec}|^3)$.

[2] $|S|+|E_{\mathcal{I}}|+|V_{\mathcal{I}}| \leqslant \alpha|V_{spec}|$ and $|D_{\mathcal{I}}||N_v||L_{total}| \leqslant |E_{\mathcal{I}}||L_{total}| \leqslant |V_{\mathcal{I}}|^2|L_{total}| \leqslant |V_{\mathcal{I}}||V_{spec}| \leqslant |V_{spec}|^2$.

**Procedure 6.** [detectUnenforceableOrders]

```
Require: U:UML 2.0 scenario specification
1:   (E_spec,V_spec) ← createSpecificationOrderGraph(U)
2:   E_impl ← createImplementationOrderGraph((E_spec,V_spec))
3:   E+_spec ← transitiveClosure(E_spec)
4:   E+_impl ← transitiveClosure(E_impl)
5:   E−_spec ← transitiveReduction(E*_spec)
6:   E−_impl ← transitiveReduction(E*_impl)
7:   E_nspec ← ∅,V_nspec ← ∅,E_nimpl ← ∅,V_nimpl ← ∅
8:   for all (e,e′) ∈ E−_impl do
9:     if (e,e′) ∉ E*_spec then
10:      E_nspec ← E_nspec ∪ {(e,e′)}
11:      V_nspec ← V_nspec ∪ {e,e′}
12:    end if
13:  end for
14:  for all (e,e′) ∈ E−_spec do
15:    if (e,e′) ∉ E*_impl then
16:      E_nimpl ← E_nimpl ∪ {(e,e′)}
17:      V_nimpl ← V_nimpl ∪ {e,e′}
18:    end if
19:  end for
20:  (E_nspec,V_nspec), (E_nimpl,V_nimpl)
```

Finally, our main algorithm is described in Procedure 6. First of all, the specification order graph and implementation order graph are created by Procedures 1 and 5. Then, their transitive closures and transitive reductions are calculated. Finally, their differences are calculated in lines 8–19.

The complexity of Procedures 1 and 5 is roughly $O(|V_{spec}|^3)$, and the transitive closure and transitive reduction also have $O(|V_{spec}|^3)$ complexity using Warshall's algorithm [17]. Therefore, the complexity of our whole approach is bound to $O(|V_{spec}|^3)$. This complexity is relatively acceptable for the large-scale scenario specification because it is polynomial.

Now, let's compare our approach with the prior approaches that detect the implied scenarios. As we mentioned previously, such works synthesize an automata-based model, which is created by parallel compositions between local automata that represent lifelines' behaviors. According to [1], under the asynchronous communication style, the realizability check for scenarios with loops is undecidable if sizes of queues, that used to store received messages between agents, are not bound. Even though the queue sizes are bound, the state space of such approaches is still exponential to the size of queues under the asynchronous communication style [1].

The synchronous communication is a special case that the size of queue is zero: the number of states of the synthesized automaton is given as $|Q_1|\ldots|Q_n|$ where $Q_1,\ldots,Q_n$ are states of local automata. In worst case, the events are equally distributed to each local automaton: $Q_i = |V_{spec}|/n$ and $i = 1\ldots n$ where $V_{spec}$ is a set of events. Therefore, the space complexity of the synthesized automaton is $O((\frac{|V_{spec}|}{n})^n)$. When $n$ is 2, previous approaches show better performance than ours, while, when $n$ is larger than 3, our approach shows better performance than others.

Therefore, we can argue that, in case of large scenarios, our approach is generally more efficient than prior approaches that detect the implied scenarios, especially under the asynchronous communication style.

## 6. Supporting other communication styles

In Section 5, we explained the detection of unenforceable orders under the asynchronous communication style.

The asynchronous communication style enforces only the orders between the sending event and their corresponding receiving event for messages, which are already included in the specification order graph and implementation order graph. However, in other communication styles, more orders are enforced. Hence, to support the other communication styles, the specification order graph and implementation order graph should have additional orders to reflect their characteristics. In this subsection, we describe such additional orders for supporting the following communication styles.

**FIFO communication.** A message sender does not wait for the arrival of the message at the receiver. A message sent first is received first.
**Synchronous communication.** A message sender waits until the message arrives at the receiver.

In the FIFO communication style, messages are received in the order in which the messages are sent. In Fig. 15, the message "questionnaire" is sent from the object "Producer" to the object "Consumer" before the message "advertise". Under the FIFO communication style, the message "questionnaire" always arrives before the message "advertise". Thus, the order between the receiving events of the messages "questionnaire" and "advertise" is conserved, although the order is the uncertain order. To reflect such conservation, the implementation order graph should have additional orders between the receiving events, in a lifeline, corresponding to the sending events that have orders between them. We refer to such orders as the *FIFO orders*. The FIFO orders are already included in the specification order graph. Thus, under the FIFO communication style, they are only added to the implementation order graph. Their definition is as follows:

**Definition 13.** Let $x$ and $y$ be receiving events such that $loc(x) = loc(y)$, and $(x,y) \in E$ for a specification order graph $g^{\mathcal{U}}_{spec} = (E, V)$. Likewise, let $x'$ and $y'$ be sending events of messages whose receiving events are $x$ and $y$, respectively. If $loc(x') = loc(y')$, then the order "$x$ to $y$" is the FIFO order.

Figs. 16 and 17 present the specification order graph, implementation order graph, and unenforceable orders for the bSD shown in Fig. 15 under asynchronous communication style and FIFO communication style, respectively. According to the above definition, in Fig. 17, the order "$r_1$ to $r_3$" is the FIFO order. Therefore, in Fig. 17b, the order "$r_1$ to $r_3$" is included in the implementation order graph. Under asynchronous communication style, the order "$r_1$ to $r_3$" is an unenforceable order. However, under FIFO communication style, it does not cause implied scenarios due to the FIFO orders as shown in Fig. 17c.

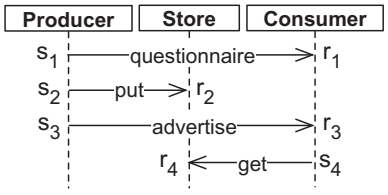


**Fig. 15.** Example bSD.

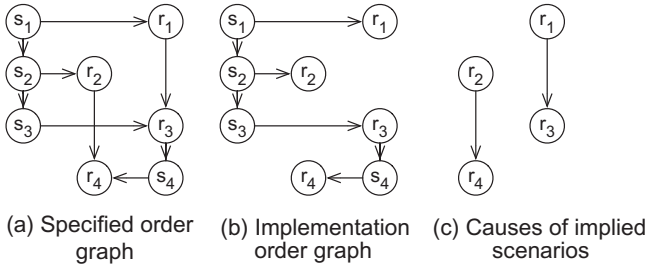


**Fig. 16.** Graphs generated from the bSD in Fig. 15 under asynchronous communication style.

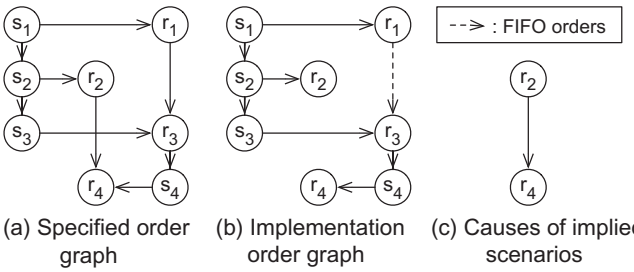


**Fig. 17.** Graphs generated from the bSD in Fig. 15 under FIFO communication style.

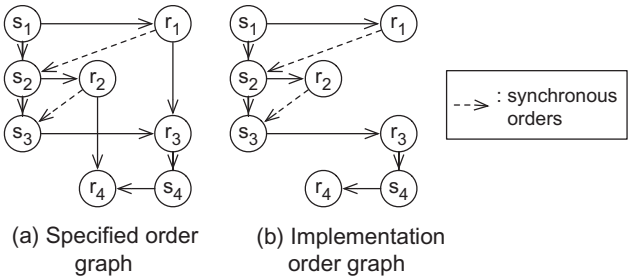


**Fig. 18.** Graphs generated from the bSD in Fig. 15 under synchronous communication style.

Now, we explain the additional orders for synchronous communication. In synchronous communication, after an object sends a message, the object waits until the message is received by another object. This means that no event can occur in the object before the receiving event of the message. Thus, for each message, the specification order graph and implementation order graph should have an order enforcing that the receiving event occur before any subsequent events. Those orders are referred to as *synchronous orders*. They are formally defined as follows:

**Definition 14.** Let $x$ and $y$ be the sending event and receiving event of a message, respectively. Also, let $z$ be the very next sending event of $x$ in the lifeline of $x$. Then, the order "$y$ to $z$" is the synchronous order.

Fig. 18 shows the specification order graph and implementation order graph under the synchronous communication style. In Fig. 15, after sending the message "questionnaire", the object "Producer" waits until the receiving event $r_1$ arises. In other words, the event $s_2$ always arises after the event $r_1$. To reflect such characteristics, the specification order graph and implementation order graph should have the order "$r_1$ to $s_2$", the synchronous order. Similarly, the order "$r_2$ to $s_3$" should also be added. Due to the synchronous orders, the bSD, shown in Fig. 15, does not have any unenforceable orders under synchronous communication style.

## 7. Case study

In this section, we present two case studies for scenario specifications on a boiler control system and mobility management in a Global Systems for Mobile communications (GSM) network. The purpose of the former case study is to show the usefulness of identifying the unenforceable orders in dealing with the implied

scenarios, and to show that our approach provides more fine-grained detection than recent approaches. Through the latter case study, we show the importance in considering various communication styles and the performance of our algorithm. These case studies were carried out on a computer with the following specifications: Intel(R) Quad Core 3 Ghz, 4GB with Windows 7 64bit. Our tool is implemented as an Eclipse plug-in and it uses the Eclipse UML plug-in. There are pure Java and Java Native Interface (JNI) versions, but, in these case studies, we use only the pure Java version. The implementation and results of the case studies can be downloaded from http://se.kaist.ac.kr/.

### 7.1. Boiler control system

We borrowed a scenario specification of a boiler control system from [9,15]. The scenario specification in [9,15] is described with the MSC. We converted it into a form of UML scenario specification. This example originally consists of four MSCs and an hMSC with 14 events. The application of our loop unrolling produces 12 MSCs and an hMSC, with 34 events in them. The seven unenforceable orders are detected under the synchronous communication style, as shown in Fig. 19.

In Fig. 19, the detected unenforceable orders are shown as lines: the dashed lines denote $g_{nimpl}$ that represents orders which are enforced in the scenario specification but cannot be enforced in the implementation model, while the solid lines denote $g_{nspec}$ that represents orders which should not be enforced, but are held in the implementation model. The parenthesized number beside each unenforceable order is its identifier. Each circle represents an event whose label consists of the name of a lifeline holding the event, the type of event, and the name of a message related to the message, from the first line.

To get an intuitive viewpoint, we overlay those unenforceable orders on the scenario specification [15,9], as shown in Fig. 20. According to the semantics of $g_{nimpl}$ and $g_{nspec}$, each of the unenforceable orders in Fig. 20 is interpreted as follows:

*Unenforceable order.* (*1*) The scenario specification enforces that the "Database" receives the message "pressure" after sending of the message "data". However, the message "pressure" may arrive before the sending. The "Sensor" does not know about the internal state of the "Database", so the "Sensor" may send the message "pressure" before sending the message "data".
*Unenforceable order.* (*2*) The scenario specification enforces that the "Database" receives the message "query" after receiving the message "pressure". However, the message "query" may arrive before the reception of the message "pressure". The "Sensor" and "Control" do not know about each other's internal state, so the "Control" may send the message "query" before "Sensor" sends the message "pressure".
*Unenforceable orders.* (*3*), (*4*) The scenario specification enforces that the sending events for the message "off" and "pressure" should not occur concurrently. However, after the decision node "d", those events may simultaneously occur.[3] At the decision node "d", the "Sensor" and "Control" independently make a decision, so that the "Sensor" may proceed to "**sd**SensorSendsData", while the "Control" may proceed to "**sd**TurnSensorOff". Then, they may make send the messages "pressure" and "off" concurrently.
*Unenforceable orders.* (*5*), (*6*) The scenario specification enforces that the sending events for the message "query" and "pressure" should not occur concurrently. However, after the decision node "d", those events may simultaneously occur.3
At the decision node "d", the "Sensor" and "Control" independently make a decision, so that the "Sensor" may proceed to "**sd** SensorSendsData" while the "Control" may proceed to "**sd** CommandActuator". Then they may send the messages "pressure" and "query" concurrently.
*Unenforceable order.* (*7*) The scenario specification enforces that the "Sensor" receives the message "off" after sending the message "pressure". However, the message "off" may arrive before sending the message "pressure". The "Control" does not know about the internal state of the "Sensor", so the "Control" may send the message "off" before sending the message "pressure".

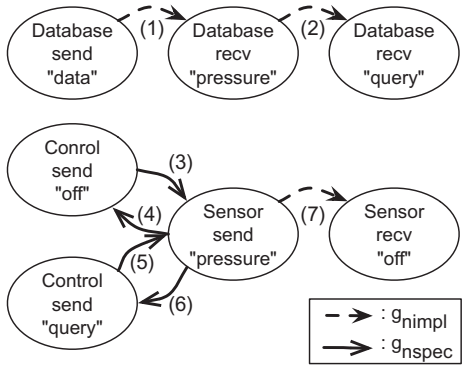


**Fig. 19.** Unenforceable orders in the boiler control system under synchronous communication style.

For the comparative case study, we choose Letier et al.'s work [9] since only the work covered the input–output implied scenarios among the related works. However, there were no its concrete and full implementations. Thus, we manually made LTSs according to their approach and used Labeled Transition System Analyzer (LTSA) [14] for calculating the implied scenarios. The LTSs used for this case study can be downloaded in http://se.kaist.ac.kr/.

Fig. 21 shows the detected implied scenarios and corresponding unenforceable orders which are detected by our approach. All detected implied scenarios correspond to a subset of the unenforceable orders. Therefore, we can argue that our approach is consistent with previous approaches.

As shown in Fig. 21, the implied scenario is just an error trace. Such error traces do not identify which part of the scenario specification leads to the implied scenarios. Thus, using existing approaches, the designer who wants to treat the implied scenarios should compare the implied scenarios with the scenario specification. On the other hand, in our approach, the unenforceable orders reveal where the problems exist and what events are relevant to the problems, as shown in Fig. 20. For example, through the unenforceable order (2) in Fig. 21, a designer can focus on devising a means to coordinate the receiving events of the message "pressure" and "query" without analysis of the implied scenario. Therefore, we argue that identifying the causes of implied scenarios provides an easier means for treating the implied scenarios than just providing the implied scenarios, particularly in the large scenario specification.

Interestingly, the unenforceable orders (1) and (5) in Fig. 21 are related to one implied scenario. In this case, both of them should be coordinated in order to remove the implied scenario. Therefore, without the unenforceable orders, handling the implied scenarios may become harder.

Fig. 22 shows the implied scenarios that are not detected by Letier et al.'s work, but that are detected by our approach.[4] They

[3] Note that each pair of the unenforceable orders "(3) and (4)" and "(5) and (6)" make a cycle. This means that events in the cycle may occur simultaneously.

[4] In fact, our approach only detect the unenforceable orders. However, we can drive the implied scenarios from the specification order graph and the unenforceable orders.

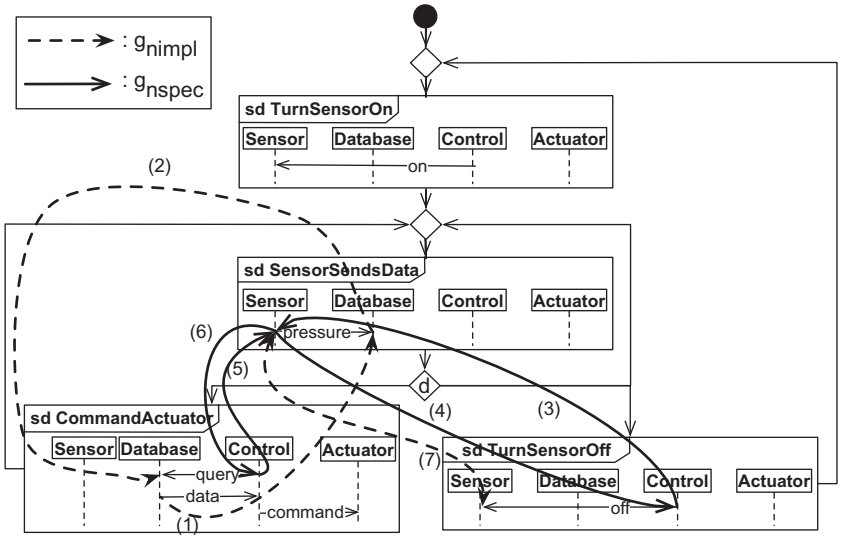


**Fig. 20.** Scenario specification for boiler control system with unenforceable orders in Fig. 19.

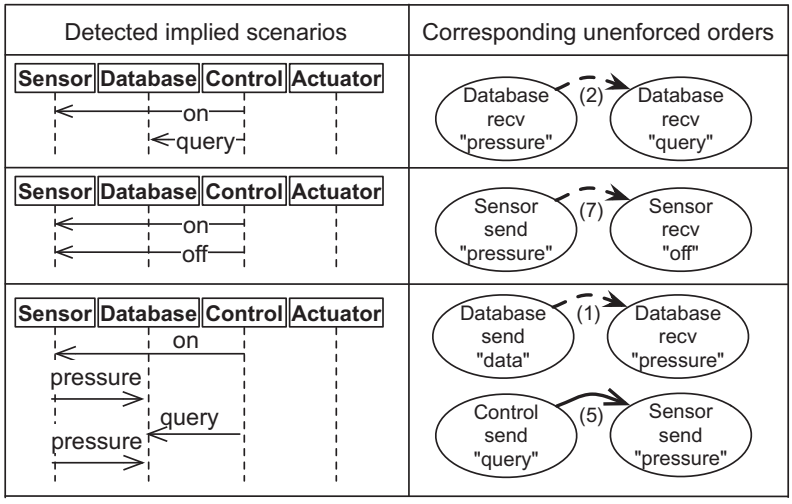


**Fig. 21.** Implied scenarios detected by previous approaches [9], and corresponding unenforceable orders.

assume the synchrony hypothesis between the sending and receiving events of a message, while we do not. Through the difference, our approach can detect more implied scenarios.

It is worth noting that some of the detected implied scenarios in Fig. 22 do not seem to be the implied scenarios(second and third ones). For example, if the boiler control system proceed along sequences of bSDs "**sd** TurnSensorOn", "**sd** SensorSendsData", "**sd** SensorSendsData", and "**sd** TurnSensorOff", the second implied scenario is explicitly presented in the scenario specification. However, the detected unenforceable order (4) explains another case. On the decision node "d", the "Sensor" can choose the path to the bSD "**sd** SensorSendsData", while the "Control" can take a path to the bSD "**sd** TurnSensorOff". Then, the bSD "**sd** TurnSensorOff" and "**sd** SensorSendsData" are concurrently executed. The first and second implied scenarios in Fig. 22 represent this phenomenon, and they are definitely undesired scenarios.

### 7.2. Mobility management in a GSM network

We borrowed a scenario specification describing mobility management in a GSM network from [10]. The specification consists of 14 MSCs and an hMSC that combines the MSCs. It has 128 events and four lifelines. After the loop unrolling, we obtained 84 MSCs and an hMSC with 420 events. Since this specification has five loops, and because four loops of them are nested, many duplications of MSCs occur.

This scenario specification has 55 unenforceable orders with the asynchronous communication style. Under the FIFO communication style and synchronous communication style, 42 and 32 unenforceable orders are detected, respectively. These results let us know that the communication style significantly affects the implied scenarios. Thus, to guarantee the absence of the implied scenarios, the appropriate communication style should be considered.

Fig. 23 shows one of the unenforceable orders detected from the scenario specification under the asynchronous and FIFO communication style. The bold line denotes a detected unenforceable order. It describes that the receiving event of the message "CONNECT_ACK" can arise after the receiving event of the message "DISC". In the synchronous communication style, the object "MS" waits until the message "CONNECT_ACK" arrives at the object "MSC". After the arrival, the messages "DISCON" and "DISC" can be sent. Therefore, under the synchronous communication style,

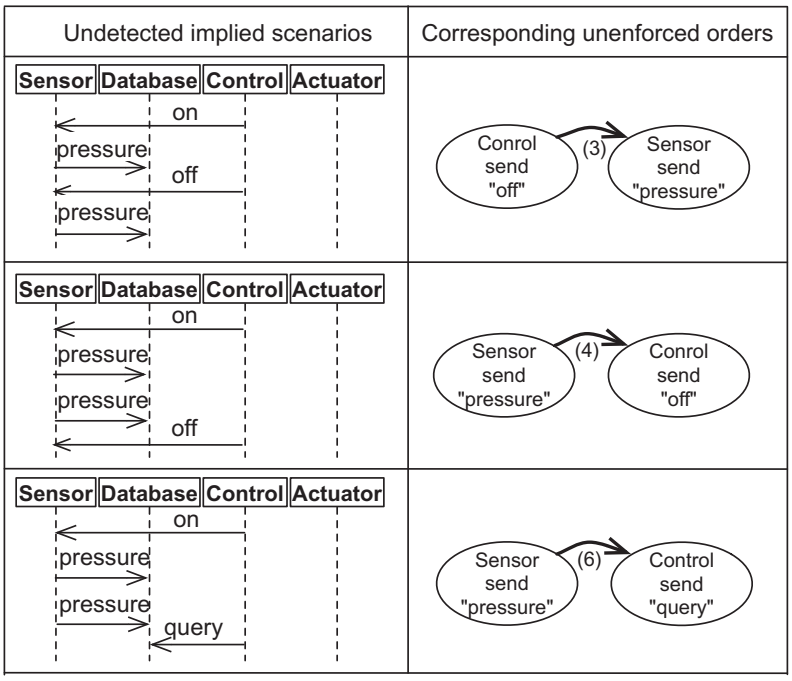


**Fig. 22.** Implied scenarios that are not detected by previous approaches [9].

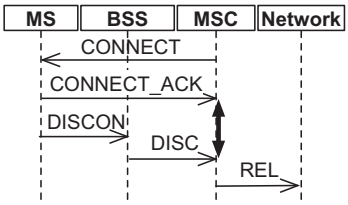


**Fig. 23.** A detected unenforceable order from the mobility management in a GSM network under asynchronous and FIFO communication styles.

the detected unenforceable order does not lead to the implied scenarios. However, in the asynchronous or FIFO communication style, the order causes the implied scenarios since each object does not wait until the arrival of the messages. From this example, we know that the communication style should be considered for detection of the correct unenforceable orders.

As a justification for the complexity of our approach, for these two case studies, we analyzed the relationship between the time elapsed to detect the unenforceable orders and the number of events. Since we use the loop-unrolling technique, we also analyzed the number of events after the loop unrolling. Table 1 shows the ratios of the number of events to the elapsed time. The elapsed time was measured under the synchronous communication style, which introduces a larger set of edges to the specification and implementation order graph than other communication styles.

As we mentioned in Section 5.2, our algorithm is bound to $O(|V_U|^3)$ in the worst case, where $V_U$ is a set of events after the loop unrolling. However, in these case studies, the elapsed time was more likely to be proportional to the $|V|^2$ or $|V_U|^2$ than $|V|^3$ or $|V_U|^3$. From this result, we can conclude that the complexity of our implementation is roughly bound to $|V_U|^3$ in the worst case, but the implementation may show better performance. In addition, Table 1 demonstrates that the loop unrolling does not hamper the performance of our implementation, significantly.

In comparing with Uchitel et al.'s work [15], for the two case studies, their implementation shows 562 ms and 523 ms for the boiler control system and mobility management in a GSM network, respectively. In both of the case studies, our implementation shows better performance.[5]

### 7.3. Discussion

Throughout the two case studies, we demonstrated the advantages of our approach. Comparing with the previous approaches, (1) our approach identifies the causes of implied scenarios by the unenforceable orders, (2) detects more implied scenarios, (3) and is applicable to the asynchronous or FIFO communication styles.

In general, detecting the implied scenarios helps to elaborate a scenario specification that does not lead miss communication between stakeholders related to the scenario specification. Therefore, the first and second advantages may help to reduce miss-communication. Moreover, since our approach supports the asynchronous and FIFO communication styles, it is applicable to the choreography model in Business Process Modeling Notation (BPMN) 2.0 or WS-Choreography, which are based on the asynchronous or FIFO communication.

However, during the case studies, we also found two drawbacks. First, while the complexity of our main algorithm is bound to $O(|V|^3)$, the loop unrolling increases $|V|$, where $|V|$ means the number of events. In Table 1, our second case study has 420 events although it originally has 128 events. As we mentioned Section 4, our loop unrolling technique may duplicate the events: deeper nested loop may cause more duplication. To minimize such duplication, we are now working on devising new loop unrolling technique.

The other is that, in some cases, the unenforceable orders may be hard to be understood. For example, in Fig. 22, the second and third orders do not seem to be the unenforceable orders, because "**sd** TurnSensorOff" is a successor of "**sd** SensorSendsData". Therefore, when every lifeline proceeds from "**sd** SensorSendsData" to "**sd** TurnSensorOff" [6], the detected orders are not unenforceable orders. On

[5] Uchitel et al.'s work [15] implementation interestingly shows faster performance in the larger case study. According to [16], their synthesis technique is more sensitive for the number of lifelines and bSDs than the number of events.

[6] A case that no non-local choice occurs.

**Table 1**
Ratios between the elapsed time and the number of events.

| | Elapsed time (ms) | $\|V\|$[a] | $\|V_U\|$[b] | $\|V\|^2$ | $\|V_U\|^2$ | $\|V\|^3$ | $\|V_U\|^3$ |
|---|---|---|---|---|---|---|---|
| GSM | 328.00 | 128.00 | 420.00 | 16384.00 | 176400.00 | 2097152.00 | 74088000.00 |
| Boiler | 3.00 | 14.00 | 34.00 | 196.00 | 1156.00 | 2744.00 | 39,304.00 |
| Ratio (GSM/Boiler) | 109.33 | 9.14 | 12.35 | 83.59 | 152.60 | 764.27 | 1885.00 |

[a] $|V|$ is the number of events.
[b] $|V_U|$ is the number of events after the loop unrolling.

the other hand, as we previously mentioned, a non-local choice may occur on the decision vertex "d". When the non-local choice occurs on the decision vertex "d", the sending event of "pressure" and receiving event of "off" can occur simultaneously. The unenforceable orders (3) and (4) in Fig. 20 represent such a case. To help the interpretation, we are now considering to provide the context information, that represents a situation which make the orders unenforceable, to users. Such context information may help that the users can understand why the detected orders are unenforceable, and, furthermore, may enable the automatic treatment of the unenforceable orders.

## 8. Conclusion and future work

In this paper, we have presented an approach to detecting the unenforceable orders in order to identify the causes of implied scenarios. The UML scenario specification enforces relative orders between events. Among those orders, some may not be inherently enforced. Such orders are the unenforceable orders. According to the definition, the unenforceable orders cause the implied scenarios. To detect the unenforceable orders, our approach first creates the specification order graph and implementation order graph that stand for the scenario specification and implementation model. Then, through the differences between them, the unenforceable orders are obtained. We provide the pseudo-codes for our approach and analyze their complexity. Two case studies are presented to validate our approach. With the case studies, we show that the unenforceable orders not only help to efficiently handle the implied scenarios, but also detect more implied scenarios than other approaches. Furthermore, the case studies convey the importance of consideration of the various communication styles, as well as present the performance of our implementation.

The advantages of our approach is summarized in three points. First, the unenforceable orders can identify which part should be considered to handle the implied scenarios. The implied scenarios are the differences between behaviors of the scenario specification and implementation model. Previous approaches use a model-checking technique to compare them. However, since the model-checking technique returns error traces when there are errors, the approaches only figure out results as a form of error traces which are referred to as the implied scenarios. To use the model-checking technique, an automata-based implementation model should be synthesized. Through the synthesis, the traceability between the specification and the implementation model is lost. On the other hand, our approach compares the specification order graph and the implementation order graph. Since the two graphs have the same events as the scenario specification, differences between them indicate which part leads to the implied scenarios: the differences are the unenforceable orders. Second, our approach provides more fine-grained detection than previous works. Unlike previous approaches, we do not assume the synchrony hypothesis between the sending and receiving events of a message. Our approach separately uses the sending and receiving events. This allows our approach to detect more implied scenarios. Third, due to the nature of the causal order graph, our approach efficiently supports the asynchronous communication style. Based on the asynchronous communication style, we provide the technique to support various communication styles. In addition, our algorithm is relatively efficient so that we can use our approach in large-scale scenario specification.

Additional work is expected to mature our approach. First, an optimized algorithm for the loop unrolling should be devised. Although we devised several optimization techniques, we expect that more optimization can be performed. Second, using the unenforceable orders, we will work further on devising a method for automated treatment of the implied scenarios.

## Acknowledgments

"This research was supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency)" (NIPA-2010-(C1090-1031-0001)).

## References

[1] R. Alur, K. Etessami, M. Yannakakis, Realizability and verification of MSC graphs. In: Automata, Languages and Programming. vol. 2076 of Lecture Notes in Computer Science. Springer, Berlin/Heidelberg, pp. 797–808. URL: <http://www.springerlink.com/content/feqhmbmygr6dypbt/>.
[2] R. Alur, K. Etessami, M. Yannakakis, Inference of message sequence charts, IEEE Transaction on Software Engineering 29 (2003) 623–633.
[3] R. Alur, K. Etessami, M. Yannakakis, Inference of message sequence charts. Technical report, Laboratory for Foundations of Computer Science, Univ. of Edinburgh, 2003. <http://homepages.inf.ed.ac.uk/kousha/msc_inference_j_v.ps>.
[4] R. Alur, M. Yannakakis, Model checking of message sequence charts. In: Proceedings of CONCUR'99: Concurrency Theory, 10th International Conference, 1999, pp. 114–129.
[5] P. Baker, P. Bristow, C. Jervis, D. King, R. Thomson, B. Mitchell, S. Burton, Detecting and resolving semantic pathologies in UML sequence diagrams, in: Proceedings of ESEC-10/FSE-13, ACM, New York, NY, USA, 2005, pp. 50–59.
[6] H. Ben-Abdallah, S. Leue, Syntactic detection of process divergence and non-local choice in message sequence charts, in: TACAS '97: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems, Springer-Verlag, London, UK, 1997, pp. 259–274.
[7] G. Decker, M. Weske, Local enforceability in interaction petri nets, Business Process Management (2007) 305–319.
[8] ITU-T, May 1996. ITU-T recommendation Z.120. Message Sequence Charts (MSC'96). ITU Telecommunication Standardization Sector.
[9] E. Letier, J. Kramer, J. Magee, S. Uchitel, Monitoring and control in scenario-based requirements analysis, in: ICSE '05: Proceedings of the 27th International Conference on Software Engineering, ACM, New York, NY, USA, 2005, pp. 382–391.
[10] S. Leue, L. Mehrmann, M. Rezai, Synthesizing ROOM Models from Message Sequence Chart Specifications. Technical Report, Electrical and Computer Engineering, University of Waterloo.
[11] H. Muccini, Detecting implied scenarios analyzing non-local branching choices, in: Proceedings of International Conference on Fundamental Approaches to Software Engineering, Springer Verlag, 2003, pp. 372–386.
[12] OMG, Unified Modeling Language: Superstructure. version 2.1.1 (formal/2007-02-03) Edition, 2007. <http://www.omg.org>.
[13] A. Sutcliffe, Scenario-based requirements engineering, in: RE '03: Proceedings of the 11th IEEE International Conference on Requirements Engineering. IEEE Computer Society, Washington, DC, USA, 2003, p. 320.
[14] S. Uchitel, LTSA-MSC Tool. Department of Computing, Imperial College, 2001. <http://www-dse.doc.ic.ac.uk/su2/Synthesis/>.

[15] S. Uchitel, J. Kramer, J. Magee, Detecting Implied Scenarios in Message Sequence Chart Specifications, SIGSOFT Software Engineering Notes 26 (5) (2001) 74–82.
[16] S. Uchitel, J. Kramer, J. Magee, Synthesis of behavioral models from scenarios, IEEE Transactions on Software Engineering 29 (2) (2003).
[17] S. Warshall, A theorem on boolean matrices, Journal of ACM 9 (1) (1962) 11–12.
[18] J. Whittle, J. Schumann, N. Center, M. Field, Generating statechart designs from scenarios, in: Software Engineering, 2000. Proceedings of the 2000 International Conference on, 2000, pp. 314–323.
[19] J. Zaha, M. Dumas, H. ter Hofstede, A. Barros, G. Decker, Bridging global and local models of service-oriented systems, IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 38 (2008) 302–318.