Implied scenario analysis in UML 2.0 scenario specification
Implied scenario analysis in UML 2.0 scenario specification
Implied scenario analysis in UML 2.0 scenario specification

Advisor : Professor Bae, Doo-Hwan

by

Song, In-Gwon

Department of Electrical Engineering and Computer Science
Division of Computer Science
Korea Advanced Institute of Science and Technology

A thesis submitted to the faculty of the Korea Advanced Institute of Science and Technology in partial fulfillment of the requirements for the degree of Master of Engineering in the Department of Electrical Engineering and Computer Science, Division of Computer Science

Daejeon, Korea
2006. 6. xx.

Approved by

__________________________
Professor Bae, Doo-Hwan
Advisor
UML 2.0 시나리오 명세에 내재된 시나리오 분석기법

송 인권

위 논문은 한국과학기술원 석사학위논문으로 학위논문심사위원회에서 심사 통과하였음.

2007년 6월 13일

심사위원장 배 두환 (인)
심사위원 김진형 (인)
심사위원 명승렬 (인)
Abstract

The correct requirements specification is essential to get the correct product. In the recent methodologies, the requirements specifications are usually specified as some form of scenarios. The implied scenarios are not desired, but exist behaviors in the scenario specifications. As UML (Unified Modeling Language) becomes popular, those scenarios are specified by UML scenario specification. Thus the techniques for detecting implied scenarios in UML scenario specification are needed. In this paper, we propose an approach to detecting implied scenarios on UML 2.0 scenario specification. We provide a canonicalization technique to make our algorithm simple. It normalizes notations of control structures of the specification to control nodes of IODs (Interaction Overview Diagrams). And it moves all parallelisms in SDs to IODs. Then, to find processes from UML scenario specification, we devise an algorithm of control flow analysis. We also provide algorithms to generate FSP (Finite State Process) specification from UML scenario specification and the result of control flow analysis. With the FSP specification, we can detect implied scenarios using the LTSA (Labelled Transition System Analyzer) tool. The detection of implied scenarios in UML scenario specification helps the designers to elaborate the specification.
Contents

Abstract i

Contents iii

List of Tables v

List of Figures vi

1 Introduction 1
  1.1 Motivation ...................................................... 1
  1.2 Scope and Overall Approach ..................................... 2
  1.3 Organization .................................................. 3

2 Background 4
  2.1 UML interaction ............................................... 4
  2.2 Terminology .................................................. 6

3 Canonicalizing scenario specification 8
  3.1 Eliminating interaction operator ................................. 8
  3.2 Removing internal implied scenarios .......................... 9
    3.2.1 Definitions .............................................. 10
    3.2.2 Partitioning bSD ......................................... 13
    3.2.3 Connecting partitioned sequence diagrams .................. 14
    3.2.4 Validation of decomposition ............................... 15

4 Control Flow Analysis 20
  4.1 Control flow graph .......................................... 20
  4.2 Generating control flow graph ................................ 21
  4.3 Validation for control flow analysis .......................... 25
List of Tables

3.1 Mapping table between interaction operator and control nodes of IOD ........ 9
7.1 Comparison among related works ............................................ 58
List of Figures

1.1 Overview of approach .................................................. 2
2.1 Example SD ............................................................... 4
2.2 Example IODs ............................................................. 5
2.3 Internal implied scenarios .............................................. 6
3.1 Example of Causal Order Graph ....................................... 13
3.2 Partitioned causal order graph ....................................... 14
3.3 Example of asynchronous concatenation ............................ 16
4.1 CFG creation for bSDs .................................................... 21
4.2 Asynchronous concatenation of bSDs ................................. 23
4.3 Example IOD ............................................................... 24
4.4 Partially created CFG ................................................... 25
4.5 Fully created CFG ....................................................... 25
5.1 Abstraction of thread .................................................... 30
5.2 Illustrations of fork constraints ....................................... 31
5.3 Handling fork nodes ..................................................... 32
5.4 Handling fork nodes with decision nodes .......................... 33
6.1 Class diagram of SFT .................................................... 39
6.2 IOD of SFT ............................................................... 40
6.3 UserInput bSD ........................................................... 41
6.4 FileReading bSD ........................................................ 42
6.5 NetworkSending bSD ................................................... 42
6.6 Encrypt bSD .............................................................. 43
6.7 Finalize bSD .............................................................. 43
6.8 DisplayResult bSD ....................................................... 44
6.9 IOD of decomposed bSD ................................................ 45
6.10 UserInput.0 bSD ........................................................ 45
6.11 UserInput.1 bSD ........................................................ 45
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.12</td>
<td>Finalize.0 BSD</td>
<td>46</td>
</tr>
<tr>
<td>6.13</td>
<td>Finalize.1 BSD</td>
<td>46</td>
</tr>
<tr>
<td>6.14</td>
<td>CFG for SFT system</td>
<td>47</td>
</tr>
<tr>
<td>6.15</td>
<td>Abstracted threads for SFT system</td>
<td>48</td>
</tr>
<tr>
<td>6.16</td>
<td>FSP obtained from IOD</td>
<td>52</td>
</tr>
<tr>
<td>6.17</td>
<td>FSP obtained from abstracted CFG</td>
<td>52</td>
</tr>
<tr>
<td>6.18</td>
<td>Redefined Figure 6.16 by property keyword</td>
<td>53</td>
</tr>
<tr>
<td>6.19</td>
<td>FSP representing implied scenarios</td>
<td>54</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Motivation

Requirements specifications are artifacts which are met in earlier time of software development process. Thus their invalidity causes significant problems to products. Recently scenarios become popular as a means of specifying the requirements. In terms of notations, simple scenarios can be described as natural language for their ease of use. But the requirements written as natural language may have ambiguity and they are very hard to verify or validate automatically. To avoid these problems, requirements of mission critical software are written as formal language. But they also have a problem that they are hard to use and understand. As a compromise of natural language and formal language, the scenarios are written as graphical semi-formal languages such as MSCs (Message Sequence Charts) [13] or UML (Unified Modeling Language) [16] interaction diagrams.

The graphical semi-formal languages usually describe the interactions in global view. But their implementation is executed with local view: a process of it does not know about states of other processes without communications. Since that difference of view level causes non-local choice problem [4], the implementation may exhibit unexpected behavior that is not intended but implied by scenarios. These unexpected behaviors are called as implied scenarios [2,20].

As the UML [6,16] becomes popular in software development, its diagrams, especially the SDs (Sequence Diagrams), are also used as a scenario description language. To reduce inconsistencies between the UML scenario description and its implementation, the techniques for detecting implied scenarios in it are needed. However, prior approaches in the implied scenario detection mainly target the MSCs ‘96 [1,3,20]. Those approaches can not be directly applied to UML SDs because there are several differences between them and the MSCs. One of them, for instance, is that vertical lines mean objects in SDs, while they mean processes or agents in MSCs [11]. Moreover, prior approaches which is targeted to UML assumed quite hard restrictions; i.e. there should be one active objects [8,14], there should be no parallelism [15], or all objects in the specification should be active objects [21]. Since those assumptions may lead the approaches to impractical one, another approach is needed to detect implied scenarios in UML.
1.2 Scope and Overall Approach

In this thesis, we propose an approach to detecting implied scenarios in UML 2.0 scenario specification. Our scenario specification consists of two UML 2.0 interaction diagrams; the IOD (Interaction Overview Diagrams) shows the overview of the flow of control and nodes in IODs are realized with the SDs. Their semantics and syntax are a subset of what is defined in UML 2.0 specification [16]. The restriction of notations are carefully decided not to limit the expressiveness. For instance, the nested IOD is prohibited. But the nested IOD can be easily flattened to one IOD. Moreover, our approach detect the implied scenarios which are not only between SDs, but also in a SD.

![Figure 1.1: Overview of approach](image-url)

Our approach consists of four steps as shown in Figure 1.1. First, UML scenario specification is canonicalized through two steps. The first step, which is called as eliminating interaction operators, makes the scenario specification have only control nodes among the notation of control structures. And then, as the second step of canonicalization, SDs of the specification are decomposed in order to remove implied scenarios in each SD. The decomposed SDs are connected by edges of the IODs. Through the connections, removed implied scenarios appear in the IOD of the specification.

Secondly, the scenario specification is analyzed to find processes. Since each vertical line
in SDs represents an object, while that of MSCs represents a process, processes that are scattered in UML scenario specification should be identified. To identify processes in UML scenario specification, we conduct the control flow analysis. We also devise algorithms to transform them to a CFG (Control Flow Graph). Each weakly connected components in the CFG represents a concurrent process.

And then we define the state-based formalism, FSP (Finite State Process) specification in this paper, of the UML scenario specification as the intended scenarios, which are scenarios that have designer’s intension, and the parallel composition of the processes as the realized scenario, which are behaviors of minimal implementation. To avoid state explosion in implied scenario detection, we abstracted the UML scenario specification and the CFG. From the abstracted UML scenario specification and CFG, both of which are a kind of IODs, the intended scenarios and realized scenarios are generated as the form of FSP specification. We provide algorithms to generate the FSP specifications from the UML scenario specification and CFG.

As the last step, implied scenarios are detected from the FSP specifications. The detection of implied scenarios is essentially a model checking problem with the intended scenarios as properties and the realized scenarios as a model. The LTSA (Labelled Transition System Analyzer) tool [18] supports the parallel composition of those FSP specification as the properties and the model. Using the tool, we can identify implied scenarios from the composed FSP specification.

1.3 Organization

The rest of this paper is organized as follows. Chapter 2 introduces the background of this study. In Chapter 3, we canonicalize UML 2.0 scenario specification and formally define the IODs and SDs. Chapter 4 defines the CFGs, provides algorithms which transform the IODs and SDs into a CFG, and provides a proof for the transformation. In Chapter 5, we provide the method of abstracting UML scenario specification and CFGs, and define the intended scenarios, the realized scenarios and the implied scenarios in our term. And then, with the definitions, we provide algorithms to generate FSP specification, and shows an example of detecting implied scenarios. Chapter 6 shows a case study to demonstrate our approach. After discussing related works in Chapter 7, we conclude our approach with the discussion of open problem and future work in Chapter 8.
2. Background

2.1 UML interaction

Our UML 2.0 scenario specification consists of SDs and an IOD. To handle them, the name of elements in the diagrams must be defined. In UML 2.0 specification [16], the syntax and semantics of the diagrams are defined as metamodels. Thus we call the elements as the name of corresponding metamodel elements. In this section, we briefly introduce the elements of SDs and IODs with examples.

![Example SD](image)

Figure 2.1: Example SD

Figure 2.1 shows an example of the SDs. The vertical lines of the diagram are called as lifelines. The horizontal arrows represent messages. In the diagram, there are ‘noticeStock’, ‘startProducing’, ‘stopProducing’, ‘start’, and ‘stop’ messages. Each of their end is called as MessageOccurrenceSpecification. The bar on the ‘Producer’ lifeline is ExecutionOccurrence in metamodel. And its top and bottom points are called as ExecutionOccurrenceSpecification. In this thesis, both of the ExecutionOccurrence and MessageOccurrenceSpecification are noted as event occurrences or simply occurrences. The CombinedFragment is shown as a rectangle, which has a dashed line in the middle; we denote it to combined fragment. The
combined fragments combine scenarios according to *interaction operator* which specifies the way of combining. In Figure 2.1, a combined fragment combines two scenarios; starting and stopping the production. And *alt* interaction operator is used. The *alt* interaction operator designates that the combined fragment represents a choice of behavior. Thus only one scenario, either start of heating or stop of heating, is executed. The scenarios that is combined by an interaction operator are noted as *interaction operands*.

![Figure 2.2: Example IODs](image)

The IODs are defined as a specialized activity diagrams in UML 2.0 specification [16]. By the definition, IODs can contain most types of nodes which are contained in the activity diagrams. But a major difference between the IODs and the activity diagrams is that the IODs show control flows among SDs while the activity diagrams show control or object flows among actions. Thus nodes of IODs, except control nodes, exhibit SDs. Since the control nodes have nearly the same semantics and syntax with those of activity diagrams, we call the control nodes as their names in the activity diagrams. Two example IODs are shown in Figure 2.2. Each of them has three SDs, a decision node, and a merge node. As shown in Figure 2.2(a), SDs can be represented as nodes of the IOD. This representation shows whole scenarios in one diagram. When the scenario becomes very complex, the diagram may be too large to draw or read. In that case, the reference nodes are used instead of SDs, as shown in Figure 2.2(b). Since this representation is more simple and concise to show overall flows, we use the representation in this thesis.

5


2.2 Terminology

Since our implied scenario detection work with both of the SDs and the IODs, it is divided into two concepts; detection on a SD and detection on an IOD. One of the causes of implied scenarios is parallelism, and the parallelism is presented as several notations in UML scenario specification. We also divide the concepts of parallelism into two category because the dealing method is different for each notation. In this section, we define those divided concepts and their terms.

As stated above, implied scenarios means differences between a scenario specification and an implementation that satisfies the specification. The differences can be arisen in both of a SD and an IOD. We call implied scenarios arisen in a SD as internal implied scenarios, while implied scenarios arisen in an IOD are called as external implied scenarios.

![Diagram of internal implied scenarios](image)

(a) Original SD (b) Implied SD

Figure 2.3: Internal implied scenarios

The objects in SDs consists of active and passive ones. The passive objects are used via methods, and the method callee can not determine when or who does the call. Figure 2.3 shows an example for this. In Figure 2.3(a), ‘Producer’ put a product to ‘Store’ and then ‘Consumer’ get it from ‘Store’. But ‘Consumer’ could get a product before ‘Producer’ puts because ‘Store’ can not determine which is the first. That case is shown in Figure 2.3(b) and it is one of the internal implied scenarios. Since the internal implied scenario can cause serious defect, it should be detected.

In SDs and IODs, there are three notations to support parallelism. First one is a concept about the active object. Since active object has one thread of control, SDs that have two or more active objects have parallelism. Secondly, there is par interaction operator. When a combined fragment with par is executed, its interaction operands are concurrently executed. Finally fork and join nodes of IODs can express the parallelism.
UML 2.0 specification exhibit another representation of the par interaction operators using fork and join nodes. By the alternative representation, the parallelism caused by par interaction operators and fork/join nodes can be categorized as one type. We call the category to horizontal parallelism and the parallelism caused by active objects to vertical parallelism.
3. Canonicalizing scenario specification

In this chapter, we provide the canonicalization of scenario specification. The canonicalization is a pre-processing that helps the implied scenarios detection by normalizing a scenario specification. SDs and IODs of UML 2.0 have various similar notations to exhibit control structures even though they have similar or same semantics. Especially, interaction operators of SDs have similar semantics with corresponding control nodes of the IODs. There are also internal implied scenarios in each SD. The distinction of internal and external implied scenarios only depends on their notational location, but their cause and characteristics are very similar. To simplify our algorithm for detecting implied scenarios, interaction operators and internal implied scenarios are removed from SDs. Instead the corresponding control semantics is expressed as control nodes in the IOD. And the internal implied scenarios are also exhibited as external implied scenarios. Through those transformation, we can get a simplified UML 2.0 scenario specification.

3.1 Eliminating interaction operator

As the first step of the canonicalization, the interaction operators are eliminated. SDs can have alt, par, loop, strict, critical, ignore, and consider interaction operators. Among them, ignore and consider are not considered because they are used just for understandability without affecting semantics. The strict and critical operators can be emulated by synchronization objects like semaphores or mutexes. So we do not consider them, too. Consequently, we only consider alt, par, and loop interaction operators. Those interaction operators can be expressed as control nodes of the IOD [16]. So we can make SDs not to have them. We call SDs that have no interaction operators as bSDs(basic Sequence Diagrams).

Table 3.1 shows the transformation, graphically. In Table 3.1, each operand of interaction operators is transformed to a vertex. A par interaction operator is transformed to a pair of fork and join nodes and an alt interaction operator is transformed to a pair of decision and merge nodes. In case of loop, it is also transformed to a pair of decision and merge nodes. But they form a control structure like while-loop. The transformations are systematically defined as one-to-one mapping. So they can be automated.
3.2 Removing internal implied scenarios

In this section, we provide a decomposition technique to remove internal implied scenarios. The bSDs may have internal implied scenarios. Since detecting both internal and external implied scenarios needs sophisticated algorithms, we remove all internal implied scenarios and the removed implied scenarios are presented as external ones.

The main idea for eliminating internal implied scenarios is decomposing a bSD to several bSDs that are totally ordered. Like external implied scenarios, internal implied scenarios...
are caused by differences between a specification and an implementation that satisfies it. The differences are made by parallelisms in the specification. Thus the bSDs that have total order do not have internal implied scenarios.

And then the decomposed bSDs are connected by edges of an IOD in order to conserve their semantics. So not only bSDs are changed to the decomposed bSDs, but also the IOD is modified. By this connection, the removed implied scenarios are appeared as external ones.

However, the decomposition may change semantics of original specification. And it may also break the well-formedness of bSDs or lose occurrences of them. To prevent them, following constraints must be hold.

**Constraint 1** (Constraints for decomposition). *For the conservation of semantics, the decomposition of bSD must hold following constraints.*

1. The order of decomposed bSDs must be a subset of the causal order of the original bSD.
2. Message sending and receiving occurrences must be exist in the bSD.
3. Decomposed bSDs must have total ordering
4. An union of sets of occurrences of decomposed bSDs must be equal to a set of occurrences of the original bSD.

We use the term “decomposition” to represent both splitting bSDs and connecting them in the IOD. And we use “partition” or “partitioning” to represent only splitting bSDs.

**3.2.1 Definitions**

Before describing decomposition technique, we first formally define the specifications and semantics of bSDs. To devise the decomposition, the formal definition of the specification is needed. And also, the definition of semantics is needed to define the decomposition that conserves their semantics.

First, we present the definition of bSD as follows:

**Definition 1** (Basic Sequence Diagram). *A basic sequence diagram \( \mathcal{B} \) is a structure \((\mathcal{O}, M, L, \prec, \text{loc})\), where:*

- \( \mathcal{O} \) is a set of event occurrences.
- \( M \) is a set of messages. A message \( m \) is a structure \((s, r, n, t)\) such that \( s \) is an occurrence that sends \( m \), \( r \) one that receives \( m \), \( n \) is the name of \( m \) and \( t \in \{\text{sync}, \text{ret}, \text{async}\} \) is the type of \( m \).
• $L$ is a set of lifelines

• $< \subseteq O \times O$ is total ordering functions of event occurrences in graphical order. To present a projection of the order for lifeline $l$, it is denoted to $<_{l}$ such that $<_{l} \subseteq <$ and $l \in L$.

• $\text{loc} : O \rightarrow L$ maps an event occurrence to a lifeline.

An IOD provides a means of describing control flows between bSDs. It consists of a graph which represents related bSDs and control flows between them. It also has control vertice: initial, final, fork, join, decision, and merge. For simplicity, we assume that there is only one initial vertex.

**Definition 2** (Interaction Overview Diagram). *Interaction overview diagram $I$ is a structure $(E, V)$, where:

• $V$ is a set of vertex. It consist of:
  
  – $\mathcal{R}$ is a set of vertex referencing bSDs.
  
  – $\top$ is a set of initial vertice such that $|\top| = 1$.
  
  – $\bot$ is a set of final vertice.
  
  – $\wedge$ is a set of fork vertice.
  
  – $\neg \wedge$ is a set of join vertice.
  
  – $\lor$ is a set of decision vertice.
  
  – $\neg \lor$ is a set of merge vertice.

• $E \subseteq (V \times V)$ is a set of directed edges that represent control flows.

We will note $\text{top}(I) = \top$ and $\text{bot}(I) = \bot$.

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

**Definition 3** (UML scenario specification). *UML scenario specification is a structure $(B, I, \text{ref})$, where:

• $B$ is a set of bSDs.

• $I$ is an IOD.
ref : $R \rightarrow B$ is a mapping function from a referencing vertex in $I$ to a referenced bSD.

The semantics of a bSD is given as causal order of event occurrences in the bSD. To define the semantics, we define the causal order of bSD as follow.

**Definition 4 (Causal Order).** Causal order of a bSD $(O, M, L, \prec, loc)$ is defined as following a partially ordered set $\prec_{\text{causal}}$.

$$\prec_{\text{causal}} = \{(b, e) | (b, r, t, n) \notin M, t \in \{\text{sync, ret}\}, (b, e) \in \prec, \forall l \in L\} \cup \{(s, r) | (s, r, t, n) \in M\}$$

The causal order consists of the order of the occurrences in a lifeline and the order of occurrences for message sending and receiving. Since scenario description diagrams, such as MSCs and SDs, are formalized by similar ways in prior works [1–3, 20], we adopt them. The only difference is the considerations of sync and ret type messages to deal with synchronous messages. If there is a sending occurrence of a synchronous message, its next occurrence is only receiving occurrence of the message because the synchronous message delegate the thread of control.

An example of the causal order is shown in Figure 3.1. In Figure 3.1(a), “Producer” calls the “put” method of “Store”. And then it sends a message “reportPutting”. “Consumer” is executed in similar ways. The causal order for Figure 3.1(a) is illustrated in Figure 3.1(b). The nodes correspond event occurrences in the bSD and edges represent ordering relation. For understandability, the nodes of the causal order graph are located with respect to the locations of corresponding event occurrences of bSDs.

We also define the min, max functions for directed graphs representing causal order, such as IODs and CFGs. Each function takes a set of edges $E$ and a set of vertex $V$ as inputs. The min function returns a set of vertex that have no incoming edges; i.e. initial vertex in case of a tree. The max function returns those that have no outgoing edges; i.e. final vertex in case of a tree. The definitions of the min, max functions are as follows:

**Definition 5 (Min, Max function of a graph).** Min function $\min : E \times V \rightarrow V$ and max function $\max : E \times V \rightarrow V$ are defined as:

$$\min(E, V) = \{v \in V | \forall w \in V, (w, v) \notin E\}$$

$$\max(E, V) = \{v \in V | \forall w \in V, (v, w) \notin E\}$$

such that $E \subseteq V \times V$
3.2.2 Partitioning bSD

In this step, a bSD is split to a set of bSDs that are totally ordered. For the splitting, we choose an approach to partition the bSD using equivalence relation. The definition of the relation is as follows:

**Definition 6 (Partitioning relation).** For a causal order $<$\textsubscript{causal} of a bSD $(O, M, L, <, \text{loc})$, partitioning relation $R : O \times O$ is defined as follows:

$$R = \lessdot_{\text{causal}}$$

$$\setminus \{(o_1,o_2)|(o_1,o_3),(o_1,o_2) \lessdot_{\text{causal}}, o_2 \neq o_3\} \quad (3.1)$$

$$\setminus \{(o_1,o_2)|(o_1,o_2),(o_3,o_2) \lessdot_{\text{causal}}, o_1 \neq o_3\} \quad (3.2)$$

$$\cup \{(o_1,o_2)|(o_1,o_2,s,t) \in M, (o_1,o_2) \lessdot_{\text{causal}}\} \quad (3.3)$$

The definition of partitioning relation is designed in order to hold the constraint 1. $R \in \lessdot_{\text{causal}}$ makes first statement of it hold. A pair of occurrences that send and receive a message must exist in the same partition by 3.3. That lets the relation hold second statement. And there are no multiple incoming/outgoing edges by 3.1 and 3.2. By these, the third statement is hold. We denote the closure of $R$ to $R^*$, which is trivially equivalence relation. Using $R^*$, we can define the partition of causal order.

**Definition 7 (Partition of Causal Order).** With an equivalence relation $R^*$ partitions $\lessdot_{\text{causal}}$, a causal order of a bSD is partitioned as a set of causal orders. We call the partition to partition of causal order and denote to $P(\lessdot_{\text{causal}})$. 

![Causal Order Graph](image_url)
Now we define a bSD partitioning function with partitioned causal order $P(<_{\text{causal}})$.

**Definition 8 (SD partitioning function).** For an original bSD $b = (O, M, L, <, \text{loc})$ and its causal order $<_{\text{causal}}$, SD partitioning function $D : B \rightarrow 2^B$ is defined as follows:

$$D(b) = \{(O_p, D_M(O_p), D_L(O_p), <_p, D_{\text{loc}}(O_p)) | O_p = D_O(<_p), <_p \in P(<_{\text{causal}})\}$$

where:

- $D_O(<_b) = \{o_1|(o_1, o_2) \text{ or } (o_2, o_1) \in <_b\}$
- $D_M(O_b) = \{(s, r, t, n) | (s, r, t, n) \in M, s \text{ and } r \in O_p\}$
- $D_L(O_b) = \{l | \text{loc}(o) = l, \forall o \in O_p\}$
- $D_{\text{loc}}(O_b) = \{(o, l) | \text{loc}(o) = l, \forall o \in O_p\}$

![Partitioned causal order graph](image)

**Figure 3.2: Partitioned causal order graph**

Partitioned causal order graph for Figure 3.1(b) is shown in Figure 3.2. The receiving occurrence of “reportGetting” message has two incoming edges. An edge from the sending occurrence of “reportGetting” is account for the “reportGetting” message. For well-formedness of partitioned bSD, the edge could not be cut. So another edge, from the receiving occurrence of “reportPutting”, is cut. The dotted line means this cutting.

### 3.2.3 Connecting partitioned sequence diagrams

The partitioned bSDs are just fragments of an original bSD. Thus the bSDs do not have the same semantics with that of original bSD. To conserve the semantics, as the last step of the decomposition of bSD, partitioned bSDs are connected as nodes of the IOD. We call the
partitioned and connected results, for one original bSD, as decomposed bSD. Its definition is as follows:

**Definition 9 (Decomposed bSD).** If there is an original bSD \( b = (O, M, L, <, \text{loc}) \) and its causal order \(<_{\text{causal}} \), decomposed bSD is an UML scenario specification \( U = ((E, V), B, \text{ref}) \), where:

- \( V = \{ [b]| b \in D(b) \} \)
- \( E' = \{ ([b_1], [b_2])| (o_1, o_2) \in <_{\text{causal}}, \exists o_1 \in O_1, \forall o_2 \in O_2, b_1, b_2 \in D(b) \} \)
  \( \cup \{ ([b_1], [b_2])| (o_1, o_2) \not\in <_{\text{causal}}, \forall o_1 \in O_1, \exists o_2 \in O_2, b_1, b_2 \in D(b) \} \) such that
  - \( b_1 = (O_1, M_1, L_1, <_1, \text{loc}_1) \)
  - \( b_2 = (O_2, M_2, L_2, <_2, \text{loc}_2) \)
- \( E = \{ (e_1, e_2)| e_2 < e_3, \forall (e_1, e_3) \in E', \exists (e_1, e_2) \in E' \} \)
- \( B = \{ [b]| b \in D(b) \} \)
- \( \text{ref} = \{ (b, [b])| b \in B \} \)

### 3.2.4 Validation of decomposition

Now we verify the semantical equality and the conservation of graphical order between the decomposed bSD and the original bSD. As mentioned above, the semantics of a bSD is defined as the causal order. These can be easily generalized from bSD scope to UML scenario specification scope. That means, if causal order of the decomposed bSD can be obtained, the semantics of the decomposed bSD and the original bSD can be compared.

Since we interpret UML scenario specification as asynchronous manner, the causal order of the decomposed bSD is obtained by asynchronous concatenation of bSDs. For simplicity, we define the concatenation as binomial operator. The asynchronous concatenation is defined as follow.

**Definition 10 (Asynchronous concatenation of bSDs).** The asynchronous concatenation of bSDs is an operator \( \odot : \mathcal{B} \times \mathcal{B} \to \mathcal{B} \).

\[
b_1 \odot b_2 = (O_1 \cup O_2, M_1 \cup M_2, L_1 \cup L_2, <_1 \cup <_2 \cup <_{\text{loc}}, \text{loc}_1 \cup \text{loc}_2)
\]

such that

- \( b_1 = (O_1, M_1, L_1, <_1, \text{loc}_1) \)
\[ b_2 = (O_2, M_2, L_2, <_2, loc_2) \]

\[ <_c = \{(o_1, o_2)|loc_1(o_1) = loc_2(o_2), o_1 \in max(O_1, <_m), o_2 \in min(O_2, <_m)\} \]

such that

\[ <_m = <_1 \cup <_2 \backslash \{(r, s)|t = async, (r, s, t, n) \in M_1 \cup M_2\} \]

Intuitively, the asynchronous concatenation is simply the union of two bSDs and then the first and last occurrences of each bSD on the same lifeline are connected. The same lifelines means same threads of control if there are no delegation of controls and there are no delegation of controls between bSDs. Thus the connection approach is valid. An the \( <_m \) is introduced to deal with the characteristics of asynchronous messages, since asynchronous messages do not affect the threads of control.

![Diagram](image_url)

**Figure 3.3: Example of asynchronous concatenation**

Figure 3.3 shows an example of the asynchronous concatenation. Figure 3.3(a) represents the decomposed bSD obtained from Figure 3.1(a). Each causal order graph for Figure 3.3(a) is presented as solid lines in Figure 3.3(b). By the asynchronous concatenation of the two bSDs, their corresponding causal orders are connected by the dashed lines. In detail, “f” and “l” are connected because they are on the same lifeline. On the other hand, “c” and “g” are not connected although they are on the same lifeline. This is because they are occurrences related to synchronous messages.
The causal order graph in Figure 3.3(b) is same with that in Figure 3.1(b) if the dashed line is assumed to be equal to other solid lines. That equality shows intuitively that the semantics is conserved after the decomposition. Based on that intuition, we present following theorem.

**Theorem 1** (Semantical equality of decomposition). A decomposed bSD $U$ with asynchronous interpretation is semantically equal to the original bSD. In more detail, the causal order of a decomposed bSD, that is created with asynchronous concatenation of bSDs, is equal to that of the original bSD.

In a case that a bSD is decomposed to two, the proof of the theorem is as follows:

**Proof.** $\precsim_{bcausal}^{b_1 \oplus b_2} = \precsim_{bcausal}^b$, such that bSD $b_1$ and $b_2$ are partitions of bSD $b$

When the causal order of bSD $b$ is denoted to $\precsim_{bcausal}^b$,

$$\precsim_{bcausal}^{b_1 \oplus b_2} = \precsim_{bcausal}^{b_1} \cup \precsim_{bcausal}^{b_2} \cup \precsim_c$$ \quad \text{； definition } 9 \quad (3.4)

If $b_1$ and $b_2$ are decomposed fragments from an original SD $b$, $\precsim_{bcausal}^{b_1} \cup \precsim_{bcausal}^{b_2} \subseteq R$, such that $R$ is partitioning relation.

Moreover, if $b$ is decomposed only to $b_1$ and $b_2$,

$$\precsim_{bcausal}^{b_1} \cup \precsim_{bcausal}^{b_2} = R \quad (3.5)$$

By 3.4, 3.5, and partitioning relation $R$,

$$\precsim_{bcausal}^{b_1 \oplus b_2} = R \cup \precsim_c$$

such that $A_1$, $A_2$ and $B$ are 3.1, 3.2 and 3.3, respectively.

If $A_1 \cup A_2 \setminus B = \precsim_c$, then $\precsim_{bcausal}^{b_1 \oplus b_2} = \precsim_{bcausal}^b$. $A_1 \cup A_2 \setminus B$ can be rewritten as follow.

$$A_1 \cup A_2 \setminus B = (A_1 \setminus B) \cup (A_2 \setminus B)$$

$$= \{(o_1, o_2) | (o_1, o_2, t, n) \not\in M, \exists (o_1, o_3) \in \precsim_{bcausal}, \forall (o_1, o_2) \in \precsim_{bcausal}, o_2 \neq o_3\}$$

$$\cup \{(o_1, o_2) | (o_1, o_2, t, n) \not\in M, \exists (o_1, o_2) \in \precsim_{bcausal}, \forall (o_3, o_2) \in \precsim_{bcausal}, o_1 \neq o_3\}$$

(3.6) and (3.7) is denoted to $A_1'$ and $A_2'$, respectively.

Now we will prove $A_1' \cup A_2' = \precsim_c$.
i. $<_{c} \subseteq A_1' \cup A_2'$

For $\forall (o_1, o_2) \in <_{c},$

\begin{align*}
\exists (o_1, o_2, s, t) \in M \quad &\because (o_1, o_2, s, t) \notin M_1 \text{ or } (o_1, o_2, s, t) \notin M_2 & (3.8) \\
(o_1, o_2) \in <_I &\because \text{loc}(o_1) = \text{loc}(o_2) \text{ and } o_x < o_y, \forall o_x \in O_1, \forall o_y \in O_2 & (3.9) \\
(o_1, o_2) \in <_{\text{causal}} &\because 3.8, 3.9 \text{ and the definition of causal order} & \\
(o_1, o_2) \notin R &\because O_1 \text{ and } O_2 \text{ is partitioned by } R \text{ such that } o_1 \in O_1, o_2 \in O_2 & \\
R = <_{\text{causal}} \setminus A_1' \setminus A_2' & \\
(o_1, o_2) \in A_1' \cup A_2' & \\
\therefore <_{c} \subseteq A_1' \cup A_2' & (3.10)
\end{align*}

ii. $A_1' \cup A_2' \subseteq <_{c}$

Recall the definition of causal order,

\begin{align*}
<_{\text{causal}} = \{(b, e) | (b, r, t, n) \notin M, t \in \{\text{sync, ret} \}, (b, e) \in <_I, \forall l \in L\} & (3.11) \\
\cup \{(s, r) | (s, r, t, n) \in M\} & (3.12)
\end{align*}

The 3.11 is noted to $C_1$ and 3.12 is noted to $C_2$.

For $\forall (o_1, o_2) \in C_1,$

\begin{align*}
\exists (o_1', o_2') \in C_1 \text{ and } \exists (o_1', o_2) \in C_1 & \because <_I \text{ of } C_1 \text{ is total ordered set} & (3.13) \\
\therefore (o_1, o_2) \in C_1 & \\
\therefore (o_1, o_2) \in C_2 & \\
\text{loc}(o_1) = \text{loc}(o_2) & \because 3.13 & (3.14)
\end{align*}

For $\forall o_m \in O_1, m, \forall o_n \in O_2,$

\begin{align*}
(o_1, o_m) \notin <_m, (o_n, o_2) \notin <_m & \quad (<_m \text{ is represented in the definition of } <_{c}) \\
\therefore O_1 < O_2 & \\
\therefore o_1 \in \max(O_1, <_m), o_2 \in \min(O_2, <_m) & (3.15)
\end{align*}

By 3.14 and 3.15,

\begin{align*}
(o_1, o_2) \in <_{c} & \\
\therefore A_1' \cup A_2' \subseteq <_{c} & (3.16)
\end{align*}
Finally, by 3.10 and 3.16

\[
\prec_c = A'_1 \cup A'_2
\]
\[
\vdots \prec_{b_1 \odot b_2}^{\text{causal}} \prec_b^{\text{causal}}
\]

\[\Rightarrow \text{The semantics of } b \text{ is kept in } b_1 \odot b_2 \text{ after the decomposition}\]

This proof can be easily applied for the general case. A decomposed bSD \(U = ((E, V), (b_1 \ldots b_n))\) is given in general case. And the \(U\) can be concatenated as \(U = b_1 \odot \ldots \odot b_n\). By extension of \(\odot\), the \(U\) can be expressed as \(U = \odot_{i=1}^n b_i\). The extended version of \(\odot\) is defined as \(\odot_{i=1}^t b_i = (\odot_{i=1}^{t-1} b_i) \odot b_t\), recursively. Intuitively, we can think that \(\odot_{i=1}^{t-1} b_i\) is not total order. Since we do not use total ordered characteristics of the causal order of \(b_1\) or \(b_2\) in the proof, our proof can be applied to the prior recursive definition and the extended \(\odot\) holds semantical equality of decomposition theorem. Therefore, we showed the proof of semantical equality of decomposition in general cases.

The decomposed bSDs have nodes of an IOD that is sorted with causal order. And, if there are nodes that can not be compared directly by partial orderedness, they are sorted with graphical order. So the graphical order of an original bSD is partially broken. As mentioned above, implied scenarios are arisen when there are parallelism and the parallelism is caused by the partial orderedness of bSDs. If there are parallelism at an original bSD, the decomposed bSDs also have one since the partial orderedness is conserved after the decomposition by the 1. Therefore, if there are implied scenarios in an original bSD, the decomposed bSDs for it must have implied scenarios. In other words, even if all the implied scenarios in original bSD are not conserved in decomposed bSDs, the decomposed bSDs have at least one implied scenario. Generally there could be so many internal implied scenarios f because of all possible sequences of partially ordered nodes. In the viewpoint of detecting and fixing implied scenarios, the detection of all the implied scenarios are not needed. Only the existence and the location of implied scenarios are needed. So the omission of the lost implied scenario is valid.
4. Control Flow Analysis

As we stated in previous chapter, the implied scenarios are caused by parallelisms among multiple processes. Therefore, finding processes is important to analyze parallelism. In bSDs, a process is scattered to multiple lifelines because the transmission of synchronous messages and their corresponding return messages mean the delegations of control. So processes that are scattered in the behavioral models should be identified to detect implied scenarios. To identify those processes, we choose to conduct the control flow analysis. Through the control flow analysis, we can find all possible execution traces and concurrent processes. In this chapter, we describe the way of transforming an IOD and SDs, which have been canonicalized, into a CFG.

4.1 Control flow graph

We first define the CFG as follows.

**Definition 11** (Control Flow Graph). A control flow graph is defined by graph $G = (E, V, ⊤, ⊥)$, where:

- $V$ is a set of vertex. $V$ can be partitioned into:
  - $O$ is a set of event occurrences on bSDs.
  - $∧$ is a set of fork vertex.
  - $∃$ is a set of join vertex.
  - $∨$ is a set of decision vertex.
  - $∀$ is a set of merge vertex.
- $⊤ ∈ V$ is a set of initial vertex.
- $⊥ ∈ V$ is a set of final vertex.
- $E ⊆ V × V$ is a set of edges.

We denote $⊤$ and $⊥$ of $G$ to $\text{top}(G)$ and $\text{bot}(G)$.

If $(a, b) ∈ E$, we will note $a ⇒ b$ and its closure relation will be denoted to $a ⇒⇒ b$. 
The CFG, which is generated from IODs and bSDs, is not fully connected graph, but consists of several weakly connected components because they have parallelisms in general. An weakly connected component in the CFG represents one thread of control in the model. So we call it as a thread and the thread means a process in terms of software system. The thread is defined as follows:

**Definition 12 (Thread).** $\Rightarrow$ is partitioning CFG into a partition $P$ because $\Rightarrow$ is an equivalence relationship. A thread $T$ is an element of the partition $P$.

### 4.2 Generating control flow graph

Now we describe the algorithm to transform the UML scenario specification to a CFG. First, each bSD in an IOD is transformed into a CFG. Then that CFG is connected to each other according to the relationships of their corresponding bSDs in the IOD. We first explain the way of transforming a bSD into a CFG.

Our algorithm starts with the creation of vertice and edges representing all occurrences of synchronous and return messages in a bSD. Then, those vertice are connected according to the temporal order of their corresponding occurrences in the bSD. Each connected component represents a thread in the bSD.

Figure 4.1 shows an example of generating CFGs from bSDs. In the figure, the vertice
‘a’ and ‘b’ of the CFG are connected through a direction of “put” message. With the same manner, the other vertex are connected for the return of “put” message, “get” message and its return. And then (‘b’, ‘c’), (‘d’, ‘e’), (‘g’, ‘i’) and (‘j’, ‘k’) pairs are connected according to the temporal order because they are on the lifelines. However the (‘e’, ‘f’) and (‘k’, ‘l’) pairs are not connected since they are sending and receiving occurrences of asynchronous messages. The generated CFGs is similar to the causal order graph. This is because the control flow is one of the elements that affects causal order. But the orders caused by asynchronous messages are ignored in control flow graph because the asynchronous messages do not delegate threads of control.

The algorithm to transform a bSD into corresponding CFG is shown in Procedure 1.

<table>
<thead>
<tr>
<th>Procedure 1</th>
<th>TransformSD2CFG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameter:</strong></td>
<td>S:SD</td>
</tr>
<tr>
<td>$G \leftarrow \text{an empty CFG}$</td>
<td></td>
</tr>
<tr>
<td>$G.O \leftarrow S.O$</td>
<td></td>
</tr>
<tr>
<td>for all $m \in S.M$ do</td>
<td></td>
</tr>
<tr>
<td>$(s, r, n, t) \leftarrow m$</td>
<td></td>
</tr>
<tr>
<td>if $t = \text{sync}$ or $t = \text{ret}$ then</td>
<td></td>
</tr>
<tr>
<td>$G.E \leftarrow G.E \cup {(s, r)}$</td>
<td></td>
</tr>
<tr>
<td>for all $o \in S.O$ and $(s, t) \in &lt;_{I}$ such that $\forall t \in S.L$ do</td>
<td></td>
</tr>
<tr>
<td>if $\exists (s, o) \in G.E$ then</td>
<td></td>
</tr>
<tr>
<td>$G.E \leftarrow G.E \cup (s, o)$</td>
<td></td>
</tr>
<tr>
<td>$G.\top \leftarrow \text{min}(G.E, G.V)$, $G.\bot \leftarrow \text{max}(G.E, G.V)$</td>
<td></td>
</tr>
<tr>
<td>return $G$</td>
<td></td>
</tr>
</tbody>
</table>

Now we have CFGs for all bSDs in an IOD. The next step is connecting those CFGs according to the relationships of their corresponding bSDs. First, for each pair of bSDs that are connected to each other in IODs, their corresponding CFGs are connected. The way of connecting them is already stated in Section 3.2.4. Procedure 2 shows the algorithm for the asynchronous concatenation of bSDs. In the Procedure, ‘Map’ Procedure is used as a mapping function. The ‘Map’ Procedure maps a vertex of an IOD to a CFG which has been created from a bSD realizing the vertex of the IOD.

Figure 4.2 shows the asynchronous concatenation for Figure 4.1. The dashed line represents the edge that is connected by asynchronous concatenation. The vertex ‘f’ and ‘l’ are connected to each other because they are on the same lifeline and are initial and final vertex of the same process in each bSD. However, even though the vertex ‘c’ and ‘g’ is on the same lifeline, they are not connected since they are not final or initial vertex of each
Procedure 2 ConnectBSDs

Parameter: $E_{iod}$: set of edges of IOD, $E_{cfg}$: set of edges of CFG

for all $e \in E_{iod}$ do

$(s, t) \leftarrow e$

if $s \in R$ and $t \in R$ then

$g_s \leftarrow \text{Map}(s)$, $g_t \leftarrow \text{Map}(t)$

for all $o_s \in \max(g_s.E, g_s.V)$ and $o_t \in \min(g_t.E, g_t.V)$ do

if $\text{loc}(o_s) = \text{loc}(o_t)$ then

$E_{cfg} \leftarrow E_{cfg} \cup \{(o_s, o_t)\}$

Figure 4.2: Asynchronous concatenation of bSDs

process. Then, for each control vertex in IODs, they are duplicated as many threads as corresponding CFG of a neighbor bSD has, and corresponding CFGs of the neighbors are connected to them. That algorithm is shown in Procedure 3.

Figure 4.3 is an example of IOD for illustration of Procedure 3. In the example, “Producer” put the product to “Store” and report to “StoreMgr” like previous examples. And then “Customer” get the product from “Store” and report it if “Customer” is normal customer. Otherwise, when the “Customer” is a retailer of products, the “Customer” get the product as a bundle.

Figure 4.4 shows partially created, which means control vertex are not handled, CFGs for the IOD in Figure 4.3. Since there are three processes in Figure 4.4, three decision nodes are created. Each decision node connects CFGs for processes with the same name. Figure 4.5 shows completed CFGs for the IOD in Figure 4.3.
Procedure 3 ConnectControlVertex

Parameter: $I : IOD, G : CFG, Map, C, \hat{C}$

for all $f \in C$ do

$S_f \leftarrow \emptyset$

for all $o_s \in \max(Map(s).E, Map(s).V)$ such that $\gamma(s, f) \in I.E$ do

$f_{os} \leftarrow$ new vertex

$C \leftarrow C \cup \{f_{os}\}$

$G.E \leftarrow G.E \cup \{(o_s, f_{os})\}$

$S_f \leftarrow S_f \cup \{f_{os}\}$

for all $o_t \in \min(Map(t).E, Map(t).V)$ such that $\gamma(f, t) \in I.E$ do

$f_{ot} \leftarrow S_f(\text{loc}(o_t))$

$G.E \leftarrow G.E \cup \{(f_{ot}, o_t)\}$

Figure 4.3: Example IOD

Finally, by setting initial vertice and final vertice, the CFG for the specification is constructed. To find the initial and final vertice, we conduct the concept of maximal spanning tree. The maximal spanning tree selects a sub graph that do not have loops. Thus the $\max$ or $\min$ could find the initial or final vertice even though there are loops in the CFG.

The whole algorithm to transform IODs to CFGs is shown in Procedure 4. In the Procedure, $I_{\text{max}}$ means a maximal spanning tree of an IOD $I$. For the relationship $G = \text{TransformIOD2CFG}(I)$ and $G = \text{TransformSD2CFG}(S)$, we will note $I \leadsto G$ and $S \leadsto G$, respectively.
4.3 Validation for control flow analysis

In this section, we provide the equivalence of semantics between UML scenario specification and the CFG, which is created from the specification, as the validation of control flow analysis. To show the equivalence, the semantics of UML scenario specification is needed to compare with one of the CFG. UML 2.0 specification [16] does not provide the formal definition of semantics for bSDs or IODs. Instead, it provides descriptive definitions. Thus we check the validity of our control flow analysis via comparing each step of it with the
**Procedure 4 TransformIOD2CFG**

**Parameter:** $I, IOD$

$G \leftarrow$ an empty CFG
$Map \leftarrow$ an empty map function

for all $r \in I.R$ do

$g \leftarrow$ TransformSD2CFG$(ref(r))$

$G \leftarrow G \cup g$

$Map \leftarrow Map \cup (r, g)$

ConnectBSD$(I.E, G.E)$

for all $i \in \{I.\wedge, I.\overline{a}, I.\vee, I.x\}$ and $g \in \{G.\wedge, G.\overline{a}, G.\vee, G.x\}$ do

ConnectControlVertex$(I, G, Map, i, g)$

if $I_{mst} \neq I$ then

$G' \leftarrow$ TransformIOD2CFG$(I_{mst})$

$G.\top \leftarrow \{i | i \in \text{top}(G')\}$, $G.\bot \leftarrow \{f | f \in \text{bot}(G')\}$

else

$G.\top \leftarrow \min(G.E, G.V)$, $G.\bot \leftarrow \max(G.E, G.V)$

return $G$


**Definitions.** First, we check the control flow analysis in a bSD.

Procedure 1, which performs the analysis, consists of two ways of connecting the occurrences. Firstly, two nodes on the same lifeline are connected. And then the two nodes, which represents the ends of a message, are connected except ones of an asynchronous message.

The UML specification defines the semantics of SDs as its trace. And the trace is defined as “weak sequencing” and “message”. The definitions of them are as follows:

**Citation 1** Weak sequencing is defined by the set of traces with these properties:

1. The ordering of OccurrenceSpecifications within each of the operands are maintained in the result.
2. OccurrenceSpecifications on different lifelines from different operands may come in any order.
3. OccurrenceSpecifications on the same lifeline from different operands are ordered such that an OccurrenceSpecification of the first operand comes before that of the second operand.

*UML superstructure specification, v2.1.1 [16], p468-p469*
The semantics of a complete Message is simply the trace \(<\text{sendEvent}, \text{receiveEvent}>\)

\textit{UML superstructure specification, v2.1.1 [16], p492}

The first and second items in the definition of weak sequencing describe the same ways of connecting with our first one. And the definition for the message is similar to the our second way. But our CFG does not have edges for asynchronous messages. Instead, we assume that the receiving occurrences of asynchronous messages waits until arriving the corresponding message. With the assumption, our CFG for a bSD has the same meaning with the trace of it.

After the creation of CFGs for each bSD, our control flow analysis connect them according to the IOD. Now we check the validity of connecting the CFGs. The connecting has two steps; connecting between bSDs, connecting between control nodes and bSDs. The connecting between bSDs are defined as “weak sequencing” in following statements.

\textit{UML superstructure specification, v2.1.1 [16], p517}

In our approach, the connecting between bSDs are defined as asynchronous concatenation. The asynchronous concatenation consists of an union of two CFG for bSDs and connections between final and initial occurrences on the same lifelines. According to citation 2, the weak sequencing also have the same meanings. Thus we could conclude that the asynchronous concatenation and weak sequencing have the same meanings.

And then the connecting with the control nodes is checked. The connection between bSDs and control nodes are not defined in UML specification. Instead we provide our insight for the control nodes as follows.

\textbf{Insight for control nodes of IODs} Control nodes of IODs means branches or concurrency of scenarios. To give concurrency to the scenarios, all members of scenarios should be forked. And also, to give branches to the scenarios, all members of scenarios also should be branched. Since the concurrency or branches are related to a thread of control, the members is presented as processes.
On the other hand, control nodes of CFGs give branches or concurrency to a thread or a process. With above insight, our algorithms duplicate control nodes for each process. And then they are connected with corresponding processes.
5. Detecting Implied Scenarios

5.1 Definitions for implied scenarios

For a minimal implementation that satisfies a specification, an implied scenario is a behavior that is included in a set of behaviors of the implementation, but not in a set of behaviors of the specification [20]. In this section, we define the specification, its minimal implementation and their behaviors in terms of our formalism. And then we define implied scenarios in terms of them. We call the behaviors of the specification as ‘intended scenarios’ while the behaviors of the implementation is called as ‘realized scenarios’.

In our approach, UML scenario specification and its CFG correspond to the specification and its implementation, respectively. Since CFG of the scenario specification represents all the traces of its occurrences, the CFG satisfies the specification. Moreover, the CFG minimally satisfies the specification because it is created from only the specification. Thus we regard the CFG as minimal implementation of the specification.

Informally, the intended scenarios are traces that follow the graphical order of event occurrences in UML scenario specification. And the realized scenarios are traces that follow the CFG. The simplest way for the detection of implied scenarios is to find the differences between those traces. However that method leads to state explosion. To avoid it, we abstract a bSD to an atomic action and the detailed behaviors of the bSD are omitted. Since the bSDs have no internal implied scenarios through canonicalization, our abstraction does not make the specification lose implied scenarios. This abstraction technique is motivated from Uchitel’s works [20]. Now we present the abstracted specification as follows:

Definition 13 (Abstracted Specification). For a UML scenario specification \( U = (B, I, ref) \), the abstracted specification \( M_{Spec} \) is defined as \( I \).

As we stated, we regard the CFG as the implementation for the UML scenario specification. Since the CFG consists of threads, a weakly connected component in it, we first abstract each thread. To obtain abstracted thread, we map a thread to a sub-graph of IODs. For a sequence of vertex in a thread, there is a bSD that contains the corresponding sequences of event occurrences. Then we can identify a set of bSDs for a thread. Figure 5.1 shows an example mapping. A sequence of vertex ‘A1’ and ‘A2’ is created from the bSD ‘B1’. Thus it is mapped to the ‘B1’. In similar way, a sequence of ‘A3’ and ‘A4’ and
another sequence of ‘A5’ and ‘A6’ are mapped to ‘B2’ and ‘B3’, respectively. The definitions of abstracted thread is as follows:

**Definition 14** (Abstracted Thread). For $U = (B, I, ref) \rightsquigarrow G$ and a thread $T \in \text{th}(G)$, an abstracted thread $M^T_{\text{impl}}$ is defined as a structure $(E_A, V_A)$. And the abstracted thread is a sub graph of IOD $I$, where:

- $V_A = \{r | \text{ref}(r) \rightsquigarrow T', T' \cap T \neq \emptyset\}$
- $E_A = \{(a, b) | a \Rightarrow w \Rightarrow b, a$ and $b \in V_A, w \notin V_A\}$

The behaviors of each abstracted thread constitute behaviors of abstracted implementation. And the abstracted threads are concurrently executed. Thus we define the abstracted implementation with parallel composition as follows:

**Definition 15** (Abstracted Implementation). For the UML scenario specification $U \rightsquigarrow G$ and $\forall T \in \text{th}(G)$, an abstracted implementation $M_{\text{impl}}$ is defined as parallel composition of all $M^T_{\text{impl}}$.

Now we define the intended scenario and realized scenario. Since they are behaviors of the abstracted specification and implementation, they are represented through function $L(M)$ that exhibits a set of execution traces, which mean behaviors, of $M$. Their definitions are as follows:

**Definition 16** (Intended Scenario). For the abstracted specification $M_{\text{spec}}$, the intended scenarios are defined as $L(M_{\text{spec}})$.

**Definition 17** (Realized Scenario). For the abstracted implementation $M_{\text{impl}}$, the realized scenarios are defined as $L(M_{\text{impl}})$. 

30
With above definitions, the implied scenarios are defined as follows.

**Definition 18** (Implied Scenario). Implied scenarios of UML scenario specification are given by \( L(M_{\text{Impl}}) \setminus L(M_{\text{Spec}}) \).

### 5.2 Creating intended scenarios and realized scenarios

As we explained, the intended scenarios and the realized scenarios are represented as the FSP specifications of abstracted specification and implementation, respectively, both of which have the form of IODs. In this section, we provide algorithms to translate IODs into FSP specification.

One issue to be solved here is the handling of fork and join vertex whose concepts the FSPs do not support. Thus, through pre-processing, we should generate FSPs for every process that occurs via fork vertex. In that case, an infinite number of processes could be generated. To avoid such case, we assume that each control flow from a fork vertex always passes a join vertex. We define that assumption as a constraint as follows.

**Definition 19** (Fork constraint). For IOD \( I = (E, V) \), a set of edges which participate \( v \Rightarrow j \Rightarrow v' \) should be equal to a set of edges which participate \( v \Rightarrow v' \) such that \( ^y f \in \land \), \( ^y j \in \lor \), \( f \Rightarrow v, j \Rightarrow v' \). And, for that \( f \) and \( j \), a set of edges which participate \( v \Rightarrow f \Rightarrow v' \) should be equal to a set of edges which participate \( v \Rightarrow v' \) such that \( ^y v \Rightarrow f \) and \( ^y v' \Rightarrow j \).

![Illustrations of fork constraints](attachment:image.png)

**Figure 5.2**: Illustrations of fork constraints

Figure 5.2(a) shows the case that satisfies the fork constraint. Control flows having node “2” and “3” are forked at the same fork node, and joined at the same join node. In Figure
control flows having nodes “2” and “3” are not joined. Furthermore there could be unbounded number of processes because node “3” is connected to node “1”. So the case in Figure 5.2(b) does not hold the fork constraint. Figure 5.2(c) shows another case. Control flows having nodes “2”, “6” and “7” are forked at different fork nodes. But they are joined at the same join node. We call the forks that hold the fork constraints to _structured forks_.

![Diagram](image)

(a) Abstracted thread with forks and joins  
(b) Transformed abstracted thread  

Figure 5.3: Handling fork nodes

Under the constraints, we transform the IOD and abstracted threads to create all the processes that occur via fork vertex. We first traverse a given graph; an IOD or an abstracted thread. On the way of traversing, the paths from an initial node to a final node could be obtained. The paths represent the processes that occur in execution. Then the paths are created as a graph. Figure 5.3 shows an example for the transformation. All the paths from ‘1’ to ‘10’ in Figure 5.3(a) are presented as a graph of Figure 5.3(b).

For decision and merge nodes, the number of them and their incoming and outgoing edges are conserved after the transformation, since they do not make any new process or thread of control. Figure 5.4 shows an example. The nodes “3” and “4” are tied by a decision and a merge node, and the nodes are disjointly executed. After the transformation, their incoming and outgoing edges are connected to the decision and the merge node, and the nodes are still disjointly executed. The Figure 5.4 does not show a case with loops. But the transformation still can be applied to that case since a loop also consists of a pair of the
(a) Abstracted thread with all types of control nodes

(b) Transformed abstracted thread for 5.4(a)

Figure 5.4: Handling fork nodes with decision nodes

decision node and merge node.

Procedure 5 TransformGraph
Parameter: cur ∈ V, prev_c ∈ V_c, G = (E, V): graph, G_c = (V_c, E_c): graph, R: set of CFG // V consists of F, a set of fork nodes, M, a set of merge nodes, and O
cur_c ← duplicateNode(cur)
V_c ← V_n ∪ {cur_c}
E_c ← E_n ∪ {(prev, cur_c)}

if cur ∈ F then
    for all e_o ∈ E, such that e_o = (cur, t) do
        G_n = (V_n, E_n) ← duplicateGraph(G_c)
        R ← R ∪ {G_n}
        TransformGraph(t, cur_c, G, G_n, R)

else if ¬cur.isVisited then
    cur.isVisited ← true
    for all e_o ∈ I.E, such that e_i = (cur, t) do
        TransformGraph(t, cur_c, G, G_c, R)

Procedure 5 shows an algorithm that creates transformed CFGs from a graph. Basically, the algorithm traverse source graph; an IOD or abstracted threads. On the way it create new graphs that are transformed graph. In detail, each time the algorithm visits a vertex, except
the fork vertex, the vertex is copied and connected to previously copied graph according to the source graph. In case of a fork vertex, currently copied entire graph is duplicated as many as the number of outgoing edges of the fork vertex. And then it traverse with each duplicated graph as a copied graph. The function “duplicateGraph” and “duplicateNode” just duplicate a graph and a node as their names, respectively.

Before introducing FSP generation algorithm, the LTS (Labelled Transition System) structure should be defined because the FSP specification is textual representation of the LTS. The definition of the LTS is as follow.

**Definition 20 (Labeled Transition System).** Labeled transition system is a structure \((S, T, \text{lbl})\) where:

- \(S\) is a set of states.
- \(T : S \times S\) is a set of transitions.
- \(\text{lbl} : T \rightarrow \text{label}\) is a labeling function for transitions \(T\).

The algorithm to create LTSs for a graph, which could be an IOD or abstracted CFG, is shown in Procedure 6. The behavior of a LTS is specified in terms of labels in transition, while that of an abstracted thread or IOD is specified in terms of vertex. So the algorithm generates LTSs by mapping each node in the graph to a labeled transition in LTS.

Procedure 8 describes the algorithm for generating FSPs. First, it creates LTSs for an IOD and an abstracted CFGs. Then the created LTSs are encoded as FSPs. The algorithm of the encoding is shown in Procedure 9. The LTS obtained from an IOD is specified as a “property”. If a LTS is specified as a property, all transitions that is not specified in the LTS are fallen into a dead state. Finally, the property and LTSs obtained from abstracted CFGs are parallely composed. By the parallel composition, a LTS that represents the implied scenarios is created.

Procedure 9 prints given LTSs as textual FSP specification. However, it prints not only LTSs, but also parallel compositions. The parallel composition composes processes that is partitioned by Procedure 5. The processes are originated from one process. By this composition, the semantics, that is lost by the transformation, is recovered.
Procedure 6 CreatingLTS
Parameter: $G = (E, V)$: graph

$L \leftarrow \emptyset$

for all $v \in V$ do
    $v$.isVisited $\leftarrow false$

for all $v \in \min(E, V)$ do
    $G_c \leftarrow$ new empty graph
    $R \leftarrow \{G_c\}$
    TransformGraph($v$, nil, $G$, $G_c$, $R$)

for all $G_c = (E_c, V_c) \in R$ do
    $l = (S, T, lbl) \leftarrow$ new LTS
    for all $v \in V_c$ do
        if $v \in \min(E_c, V_c)$ then
            for all $(v, t) \in E_c$ do
                AddTransition($l$, $[I]$, $v$, $t$)
            if $v \in \max(E_c, V_c)$ then
                for all $(s, v) \in E_c$ do
                    AddTransition($l$, $s$, $v$, $[F]$)
        for all $(s, v), (v, t) \in E_c$ do
            AddTransition($l$, $s$, $v$, $t$)
    $L \leftarrow L \cup \{l\}$
return $L$

Procedure 7 AddTransition
Parameter: $l = (S, T, lbl)$: LTS, $s \in V \cup \{[I], [F]\}$, $v \in V$, $t \in V \cup \{[I], [F]\}$

$L \leftarrow \emptyset$

$T \leftarrow T \cup \{(s, v), (v, t)\}$
$S \leftarrow S \cup \{(s, v), (v, t)\}$
$lbl \leftarrow lbl \cup \{(s, v), (v, t)\} \rightarrow \text{labelOf}(v)$
Procedure 8 GenerateFSP

Parameter: $I: IOD$, $C$: a set of abstracted CFGs

$Spec \leftarrow \text{CreateLTS}(I)$
$Impl \leftarrow \emptyset$

for all $cfg \in C$ do
    $Proc \leftarrow \text{CreateLTS}(cfg)$
    $Impl \leftarrow Impl \cup \{Proc\}$
end for

print PrimitiveLTS($Spec$, “Spec”)

for all $impl \in Impl$ do
    print PrimitiveLTS($impl$, getProcessName($impl$))
end for

print “deterministic property $\parallel Prop=\parallel Spec.$”
print “deterministic $\parallel \text{ImpliedScenario = (Prop }$"

for all $impl \in Impl$ do
    print “$\parallel ” + getProcessName($impl$)
end for

print “$\parallel ”).”
Procedure 9 printPrimitiveFSP

Parameter: $L$ : a set of LTS, $label$ : String

$i ← 0, j ← 0, ctrl ← ∅$

for all $lts = (S, T, lbl) ∈ L$ do

print “deterministic ” + label + i + “ = I”

$i ← i + 1$

for all $s ∈ S$ do

if $s = (a, [F])$ such that $∀ a ∈ V$ then

continue

print “,” + newline + s.name + “ = (”

for all $(s, t) ∈ T$ do

if $¬isFirstLoop$ then

print “∥”

print $lbl((s, t)) + “ → ” + t.name$

if $v$ is control node such that $s = (a, v), t = (v, b), ∀ a, b ∈ V$ then

$ctrl ← ctrl ∪ \{(s, t)\}$

print “)”

print “.” + newline + “||” + label + “ = (”

$i ← 0$

for all $lts = (S, T, lbl) ∈ L$ do

if $¬isFirstLoop$ then

print “||”

print label + i

print “)”

if $|ctrl| > 0$ then

for all $v ∈ ctrl$ do

if $¬isFirstLoop$ then

print “,”

else

print “{”

print $lbl((s, t))$

print “}”

print “.” + newline

37
6. Case study

In this chapter, we describe a case study to evaluate feasibility and usefulness of our approach. As input of our case study, we write the UML scenario specification for SFT (Secure File Transfer) system. We analyze the UML scenario specification using an implementation of our approach. And then we identify implied scenarios. In the following sections, we describe the SFT and their the result of implied scenario analysis.

6.1 Secure file transfer system

The SFT system encrypts a file and transfers the file to another peer via network. Indeed such system usually not only sends encrypted file but also receives it. For simplicity, we restrict the system to have only sending functionality. The SFT system is originally designed as a case study for ESUML project [12]. In the project, the SFT system use FIPS-197 AES (Advanced Encryption Standard) [7] to encrypt a file and the detail of the encryption algorithm is specified as UML 2.0 scenario specification. Since the algorithm is designed sequentially, it does not contribute to arise the implied scenarios. Thus we abstract the algorithm in this case study.

Required behaviors of the SFT system are briefly as follows:

- The system reads a file.
- The system encrypts contents of the read file.
- The system transfer the encrypted contents.

And the system should be designed to have parallelism for high-performance. Under above requirements, we model the SFT system. Due to the requirements for the performance, encryption and file-reading functionalities are designed to execute parallely. To support the parallelism, the data transmission between processes are performed only through a kind of message queues.

We use RSA (Rational Software Architecture) to draw those diagrams. The RSA is easy to integrate with Eclipse environments because it is also implemented in Eclipse environments. The Eclipse environments provide very convenient UML utilities. And also the
implementation of LTSA, which is used in FSP processing, is based on the Eclipse environments. For the ease of integration with LTSA and UML handling utilities, the tool is selected. But the diagrams in RSA are not fully compatible with UML 2.0 specification because of limitations or restrictions of the tool. The tool does not support the drawing of IODs. So, we draw an activity diagram instead of the IOD. Also, the tool presents active classes as bold boxes instead of double vertical lined boxes.

Figure 6.1: Class diagram of SFT

Figure 6.1 shows the class diagram of the SFT system. There are 6 active classes. “User” class is external class. But we regard the “User” to one of the active classes. “UserInterface” class presents GUI(Graphic User Interface) as its name implies. The class is not control class. But, since general GUI needs a thread of control, we defined the class as active. “SFTController” class is main control class. It receives commands from “UserInterface” and controls other classes. “FileReader”, “CipherStream” and “NetworkWriter” classes take charges of file reading, encrypting and network sending, respectively. And there are 3 passive
classes. “Cipher” class has an algorithm for encryption. “Buffer” class represents a queue for transferring data between “FileReader” and “CipherStream” or between “CipherStream” and “NetworkWriter”. “OSInterface” represents APIs that are provided by operating system or system base library. These three classes behave as library or provide functionalities to other classes. Thus they are defined as passive classes. The class diagram is not needed to detect implied scenarios. But we presents a class diagram of SFT system for better understanding.

![Class Diagram of SFT System](image)

**Figure 6.2: IOD of SFT**

An IOD for SFT system is shown in Figure 6.2. The system starts with “UserInput”. In the “UserInput”, the system receives target file name and IP address of a peer. Then “FileReading” and “Encrypt” are parallely executed. In those behaviors, a piece of the target file is read. At the same time, the read piece is encrypted. After the end of those two behaviors, “NetworkSending” is executed in order to send the encrypted piece to peer through the network. This sequence is repeated until all the contents of target file are read, encrypted, and sent. If the loop reaches to the end, the system frees resources by “Finalize”. Then the result of the transmission is displayed to the user and the SFT system is terminated. As stated, the nodes of IODs are realized through bSDs. The bSDs are presented in Figure 6.3 to 6.8. Now we present descriptions of those figures.
Figure 6.3: UserInput bSD
Figure 6.3 shows UserInput bSD. It starts with a request from “User”. Then the request is forwarded to “SFTController” by “UserInterface”. “SFTController” sends messages to “FileReader” and “NetworkWriter” with target file name and an IP address of peer. Then “FileReader” open target file and “NetworkWriter” opens the network connection with peer.

![FileReading bSD](image1)

Figure 6.4: FileReading bSD

![NetworkSending bSD](image2)

Figure 6.5: NetworkSending bSD

FileReading and NetworkSending bSD are shown in Figure 6.4 and 6.5. Their behaviors are very similar to each other because both of them are to read and then to write. In the FileReading bSD, “FileReader” reads a piece of file from “OSInterface” and then it writes the contents to “inputBuffer”. On the contrary, “NetworkWriter” reads a piece of encrypted data from “outputBuffer” and it writes the data to “OSInterface” in order to transfer the
data to peer.

Figure 6.6: Encrypt bSD

To encrypt the contents of the file, the “CipherStream” gets a piece of data from “inputBuffer”. And then the data is encrypted using “Cipher” object. The encrypted data is put to “outputBuffer” again. The encrypting is shown in Figure 6.6.

Figure 6.7: Finalize bSD
Before the termination of the system, all occupied resources should be returned. Figure 6.7 shows the Finalize bSD that is to free resources. “FileReader” first closes the opened file and “NetworkWriter” closes the connection. And then “FileReader” and “NetworkWriter” send finish messages to “SFTcontroller” to report.

![DisplayResult bSD](image)

**Figure 6.8: DisplayResult bSD**

At the end of the system, “SFTController” sends the result to “UserInterface”. And then the result is displayed to “User” by “UserInterface”. These are presented in Figure 6.8.

### 6.2 Decomposed bSD

We applied the decomposition of bSDs to the SFT system. Our implementation for the decomposition of bSDs receives UML 2.0 scenario specification as XMI (XML Metamodel Interchange) format from RSA. After the decomposition, the implementation produce decomposed results as XMI. To check them, the results are loaded and visualized by the RSA.

Figure 6.9 shows an IOD of decomposed results. In comparison with Figure 6.2, The nodes, of which the names have only suffix ‘.0’, are not decomposed nodes, while the other nodes are decomposed ones. But “UserInput” and “Finalize” nodes are decomposed to four and two pieces, respectively. Now we present decomposed bSDs for “UserInput” and “Finalize”.

The decomposed “UserInput” is shown in Figure 6.10 and 6.11. The “UserInput.0” and “UserInput.1” have total orderedness. And the total orderedness means that there is no parallelism. So we can conclude there are no implied scenarios.

The rest of pieces of “UserInput”, the “UserInput.2” and “UserInput.3”, have only one execution specification occurrence each. The occurrence means the end of the execution.
Figure 6.9: IOD of decomposed bSD

Figure 6.10: UserInput.0 bSD

Figure 6.11: UserInput.1 bSD
Thus they are not shown.

Figure 6.12: Finalize.0 bSD

Figure 6.13: Finalize.1 bSD

Figure 6.12 and 6.13 show the decomposed bSDs of “Finalize” node. Similar with “User-Input.0” and “UserInput.1”, those diagrams have no implied scenarios.

6.3 Control flow graph

The CFG, obtained from the scenario specification of the SFT system, is shown in Figure 6.14. The pentagon that has a tip above means a join node and the pentagon that has a tip below represents a fork node. The diamond shapes are decision nodes and the diamonds that have a line in each corner mean merge nodes. And colored nodes mean initial nodes or final nodes. Each node has three lines of annotations. First line means the name of active class that initiate the process. Second line means an object that has the corresponding occurrence with the node. And last line means the name of corresponding occurrence. There are six weakly connected components. The SFT system has six active classes, and that means there are six processes. So we can conclude that the CFG is correctly created.
Figure 6.14: CFG for SFT system
Figure 6.15: Abstracted threads for SFT system

Figure 6.15 shows the abstracted threads. They also consist of six weakly connected components. Since they are abstracted threads, they resemble with Figure 6.14. But, in the abstracted CFG, third line of the annotations of each node means corresponding bSD.
6.4 Implied scenarios

After the creation of CFG, in accordance to our approach, FSPs are created from an abstraced CFG and an IOD in prior sections. The created FSPs are as follows.

```plaintext
\texttt{deterministic NetworkWriter0 = I,}
\texttt{I = (userInput1 -> Q26),}
\texttt{Q26 = (m1 -> Q27),}
\texttt{Q27 = (d1 -> Q28 | d1 -> Q29),}
\texttt{Q28 = (f1 -> Q30),}
\texttt{Q30 = (f2 -> Q31),}
\texttt{Q31 = (networkSending0 -> Q32),}
\texttt{Q32 = (m1 -> Q27),}
\texttt{Q29 = (finalize1 -> END) \ \{d1, m1, f1, f2\}.}

\texttt{deterministic SFTController0 = I,}
\texttt{I = (userInput0 -> Q33),}
\texttt{Q33 = (userInput1 -> Q34),}
\texttt{Q34 = (m1 -> Q35),}
\texttt{Q35 = (d1 -> Q36 | d1 -> Q37),}
\texttt{Q36 = (f1 -> Q38),}
\texttt{Q38 = (f2 -> Q39),}
\texttt{Q39 = (m1 -> Q35),}
\texttt{Q37 = (finalize0 -> Q40),}
\texttt{Q40 = (finalize1 -> Q41),}
\texttt{Q41 = (displayResult0 -> END) \ \{d1, m1, f1, f2\}.}

\texttt{deterministic CipherStream1 = I,}
\texttt{I = (f1 -> Q23),}
\texttt{Q23 = (encrypt0 -> Q25),}
\texttt{Q25 = (f2 -> I).}

\texttt{deterministic CipherStream0 = I,}
\texttt{I = (m1 -> Q20),}
\texttt{Q20 = (d1 -> Q21 | d1 -> END),}
```
Q21 = (f1 -> Q22),
Q22 = (f2 -> Q24),
Q24 = (m1 -> Q20).

deterministic ||CipherStream2 = (CipherStream1 ||CipherStream0) \ {d1, m1, f1, f2}.

deterministic FileReader1 = I,
I = (f1 -> Q11),
Q11 = (f2 -> I).

deterministic FileReader0 = I,
I = (userInput0 -> Q6),
Q6 = (m1 -> Q7),
Q7 = (d1 -> Q8 | d1 -> Q9),
Q8 = (f1 -> Q10),
Q10 = (fileReading0 -> Q12),
Q12 = (f2 -> Q13),
Q13 = (m1 -> Q7),
Q9 = (finalize0 -> END).

deterministic ||FileReader2 = (FileReader1 ||FileReader0) \ {d1, m1, f1, f2}.

deterministic User0 = I,
I = (userInput0 -> Q0),
Q0 = (m1 -> Q1),
Q1 = (d1 -> Q2 | d1 -> Q3),
Q2 = (f1 -> Q4),
Q4 = (f2 -> Q5),
Q5 = (m1 -> Q1),
Q3 = (displayResult0 -> END) \ {d1, m1, f1, f2}.

deterministic UserInterface0 = I,
I = (userInput0 -> Q14),
Q14 = (m1 -> Q15),
Q15 = (d1 -> Q16 | d1 -> Q17),
Q16 = (f1 -> Q18),
Q18 = (f2 -> Q19),
Q19 = (m1 -> Q15),
Q17 = (displayResult0 -> END) \ {d1, m1, f1, f2}.

deterministic \ Impl = (NetworkWriter0 | SFTController0 | CipherStream2 | FileReader2 | User0 | UserInterface0).

deterministic Spec0 = Q2,
Q2 = (userInput0 -> Q3),
Q3 = (userInput1 -> Q4),
Q4 = (userInput2 -> Q5),
Q5 = (userInput3 -> Q6),
Q6 = (m1 -> Q0),
Q0 = (d1 -> Q1 | d1 -> Q7),
Q1 = (f1 -> Q9),
Q9 = (fileReading0 -> Q10),
Q10 = (f2 -> Q13),
Q13 = (networkSending0 -> Q14),
Q14 = (m1 -> Q0),
Q7 = (finalize0 -> Q8),
Q8 = (finalize1 -> Q16),
Q16 = (displayResult0 -> END).

deterministic Spec1 = I,
I = (f1 -> Q11),
Q11 = (encrypt0 -> Q12),
Q12 = (f2 -> I).

deterministic | Spec_COMP = (Spec0 | Spec1) \ {d1, m1, f1, f2}.
property | Spec = Spec_COMP.
deterministic $\|Implied\text{Scenario} = (\text{Impl} \| \text{Spec})$.

Figure 6.16: FSP obtained from IOD

Figure 6.17: FSP obtained from abstracted CFG

Figure 6.16 shows the LTS obtained from the IOD of the SFT system. And Figure 6.17 shows the LTSs obtained from abstracted CFG. In Figure 6.17, large sized words mean the names of active classes that initiate corresponding processes. As shown in the figures, six LTSs are created for the six processes.
Figure 6.18 shows redefined version of Figure 6.16 through 'property' keyword of FSPs. By the keyword, all the paths that is not contained in Figure 6.16 are transitioned into a dead state; '-1' state.

As the last step of our approach, a FSP representing implied scenarios are created by parallel composition of LTSs in Figure 6.17 and 6.18. The LTS is shown in Figure 6.19. In the FSP, all paths that reach to '-1' state mean implied scenarios because the paths are not permitted by the property, as shown in Figure 6.18. For example, the path sequence of 'userInput0', 'userInput1', 'userInput2', 'userInput3', and 'networkSending0' is one of the implied scenario. Since the 'NetworkWriter' does not know whether encryption is finished or not, it may start sending before encryption or file reading. Thus the implied scenario can be arisen and it can cause errors; when reading from empty output buffer, errors can be arisen. The path 'userInput0', 'userInput1', 'userInput2', 'userInput3', 'fileReading0', and 'networkSending0' is another implied scenario. This implied scenario is also caused by the absence of communication between 'NetworkWriter' and other processes.
Figure 6.19: FSP representing implied scenarios
7. Related Work

7.1 Implied scenarios in MSCs

Many prior works for detecting implied scenarios were researched for the MSC specification. Although they can not be applied directly to UML scenario specification, it is worth comparing with them because their objective is same with ours. The term implied scenario was first introduced by Alur et al [1,3]. They provided a framework to verify that given scenario specification is realizable by some implementation. Their realizability consists of two level. First level of it is satisfied if there are no implied scenarios, and the satisfaction of second level means dead-lock free. But, their work is limited to MSCs that specify finite system behaviors, which means that there are no loops. Uchitel et al. have extended Alur’s work by providing a framework to detect implied scenarios in MSCs that specify infinite system behaviors which are expressed by loops of hMSCs [19, 20]. They presented an algorithm that builds a LTS behavior model that describes the closest possible implementation for a specification based on MSCs and hMSCs. And also they presented the way of obtaining differences between MSC specification and the implementation. Their work use MSC ’96 standard [13] as their MSC specification.

In researches for scenario synthesis, there are several approaches which support the implied scenario detection. Sgroi et al. [17] proposed a Petri Net synthesis technique from MSC specification as an approach to design the communication protocols. In their approach, detailed behaviors are described using MSCs, and relations among the MSCs are represented using Petri Net semantics. From the specifications, a Petri Net is synthesized. On last step of the synthesis, inconsistency, which corresponds external implied scenarios in our terms, between the specification and synthesized Petri Net is removed. Bontemps, et al. [5] propose synthesis for the LSCs that is an extension of the MSCs. Their first objective was fast synthesis. Thus they sacrificed some precision of synthesized result for the synthesis speed. To compensate it, they checked the inconsistencies between scenario specification and synthesized model. And their inconsistency checks contain the concepts of detecting implied scenarios.
7.2 Implied scenarios in UML

The implied scenario detection in UML is mostly researched as a part of works about scenario synthesis. First scenario synthesis approach, which contains the detection of implied scenarios, is shown in Elkoutbi and Keller’s work [8]. In the work, they proposed a requirements engineering process that generates a user interface prototype from scenarios and yields a formal specification of the system as the form of a high-level Petri Net. Their scenarios are acquired in the form of SDs. These diagrams are transformed into Petri Net specifications and merged to obtain a global Petri Net specification capturing the behavior of the entire system. On the merging, the resulting specification captures in general not only the input scenarios, but also external implied scenarios in our terms. They referred that as interleaving problem and provided a means of solving it. Their approach were not automated because a part of transformations needs manual operation. Mäkinen and Systä proposed MAS(Minimally Adequate Synthesizer) which is an interactive algorithm that synthesizes UML statechart diagrams from SDs [15]. It composes the behaviors specified in SDs using one of the machine learning techniques. The composition first creates a state machine for all possible combination of scenarios. The resulted state machine may contain undesired generalizations. Thus they remove the undesired generalization by consulting the user. In our terms, that undesired generalization means external implied scenarios. Their work is limited to single process systems since their SDs contain only one process and are composed as single process. The synthesis technique using collaboration diagrams was proposed by Khriss et al. [14]. The approach has similar basement to Elkoutbi and Keller’s work [8], but this work is automated and the synthesized model is presented as state machine diagrams. And also the work gives explicit algorithm for detection of implied scenarios as finding interleaving.

Uchitel et al. [21] proposed another synthesis and analysis technique using partial labelled transition systems. In the work, they argued that state machine based formalisms are inadequate to describe behavior models that are being developed. To deal with such cases, they proposed the partial labelled transition systems which extend LTSs to describe required behaviors. Through comparison between the partial labelled transition systems and LTSs, which is obtained from developed model at that time, they detect undefined behaviors. Their concept of undefined behaviors extends that of implied scenarios by using the partial labelled transition system. Their approach restricts objects to active ones.
7.3 Comparison

The table describing the comparison of our work with prior works is shown in Table 7.1. The description for each comparison criteria is as follows:

Notation This means notations for scenario specification that is used by corresponding detection technique. In this comparison, SDs, IODs, MSC, hMSC, LSC and collaboration diagrams are shown. The collaboration diagrams are denoted to CD in Table 7.1.

Vertical and horizontal parallelism This is about whether the source scenario specifications can have vertical or horizontal parallelism or not. Vertical parallelism is originally a sort of parallelism which is caused by active objects of SDs or processes of MSCs. In case of collaboration diagrams, we distinguish parallelism by a location causing the parallelism; if a parallelism is arisen in one diagram, it is vertical parallelism. Otherwise it is horizontal parallelism.

Branch This means whether source scenario specifications can have branches or not. Since the branch is essential control element, most scenario specifications can express the branch. The scenario specifications that can not express the branches may derive branches on compositioning scenarios. But in this comparison, we do not consider those cases.

Automation This means how much the detection is automated. The “Semi” means that it is partially automated, while the “Full” means that it is fully automated. Except earlier researches, most of researches are fully automated.

Distinction between passive and active object This means whether the approach distinguishes active and passive objects, or not. Most of researches handled it via one viewpoint; all objects are active or all objects are passive. In MSCs, there are no concepts for passive objects. So we give N/A to those approaches using the MSCs.

Internal implied scenarios This means whether implied scenarios in one elementary diagram such as SDs, MSCs, and collaboration diagrams are detected or not. Internal implied scenarios were handled in Alur’s works [1, 3]. But posterior works mostly do not handle the internal implied scenarios.
<table>
<thead>
<tr>
<th>Author</th>
<th>Elkoutbi and Keller '00</th>
<th>Mäkinen and Systä '01</th>
<th>Khriss et al. '01</th>
<th>Uchitel et al. '01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notation</td>
<td>SD, UCD</td>
<td>SD</td>
<td>CD</td>
<td>MSC, hMSC</td>
</tr>
<tr>
<td>Parallelism</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertical</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>Horizontal</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>Branch</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Automation</td>
<td>Semi</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
</tr>
<tr>
<td>Distinction between</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>passive and active object</td>
<td></td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>Internal implied scenarios</td>
<td></td>
<td>N/A</td>
<td>N/A</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Author</th>
<th>Sgroi et al. '04</th>
<th>Bontemps et al. '04</th>
<th>Uchitel et al. '03</th>
<th>My work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notation</td>
<td>MSC</td>
<td>LSC</td>
<td>SD</td>
<td>SD, IOD</td>
</tr>
<tr>
<td>Parallelism</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertical</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Horizontal</td>
<td>N/A</td>
<td>N/A</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Branch</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Automation</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td></td>
</tr>
<tr>
<td>Distinction between</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>passive and active object</td>
<td></td>
<td>N/A</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>Internal implied scenarios</td>
<td></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 7.1: Comparison among related works
8. Conclusion

8.1 Contribution

In this thesis, we have proposed an approach to detecting implied scenarios in the UML scenario specifications, which consist of an IOD and SDs. Our UML 2.0 scenario specification permits using of various notations and mechanisms which are defined in UML 2.0 specification. We canonicalize the scenario specification to make algorithms for implied scenario detection simple and to detect internal implied scenarios. The first step of the canonicalization make the SDs to bSDs by converting combined fragments to control nodes. Then, as the second step of the canonicalization, the bSDs are decomposed to detect the internal implied scenarios and the decomposed bSDs are connected. Through the connection, they can conserve their causal order. We have also provided the proof for the conservation.

To identify processes that are scattered in the UML scenario specification, we have suggested a control flow analysis method. Each weakly connected component in the generated CFGs represents a concurrent thread. To validate the control flow analysis, each step of the analysis has been compared with UML 2.0 specification. We have defined the formal representation, FSP specification, of the UML scenario specification as intended scenarios. And the formal representation of the parallel composition of the threads, the CFGs, has been defined as the realized scenarios for the specification. To avoid the state explosion in implied scenario detection, we have abstracted the UML scenario specification and the CFGs. We have provided algorithms to generate the intended scenarios and realized scenarios from the abstracted UML scenario specification and CFGs. With the intended scenarios and realized scenarios, both of which are described as FSP specifications, we can identify implied scenarios using the LTSA tool.

Scenario description is widely accepted as a means to capture requirements. In the past, the MSCs were usually used to describe scenarios. As the UML 2.0 interaction diagrams incorporate many notations in MSCs, the use of UML in scenario description becomes prevailing. However, prior approaches for the implied scenario detection are only for MSCs or restrict expressiveness of UML 2.0. With our approach, we can detect implied scenarios in the UML 2.0 scenario specification, thus provide a means to elaborate the scenario specification. Also our approach can reduce inconsistencies between the UML 2.0 based requirements specification and the implementation for it. Furthermore, we believe that our approach lays
one of the keystone of scenario synthesis for UML 2.0 based scenario specification.

8.2 Open problems

The UML 2.0 specification says that fork/join nodes pair and decision/merge nodes pair should have properly nested form, such as our fork node constraint. According to our experiences, the restriction could be annoying modelers. A system, which fork a service process when a request is accepted, is naturally designed as a unstructured form in IODs. The unstructured form of IODs causes unbounded number of states of formalism [9, 10], and the unboundedness causes state explosion. Since both of the state explosion and sacrificing expressiveness could make proposed approach to impractical one, the tradeoff between expressiveness and state explosion should be carefully considered. In this thesis, we have decided to restrict IODs. But a compensation or a breakthrough of the tradeoff should be researched.

8.3 Future work

Several works should be done to mature our approach. The detected implied scenarios should be determined whether they are accepted or not. To do that, first of all, the designer should be able to see the detected implied scenarios as a scenario specification. Since the number of the implied scenarios are generally infinite, we should investigate a method of finding finite representatives among those implied scenarios. And then, if the implied scenarios are rejected, the rejected scenarios should be removed from the specification. To support an automatic method for the rejection, we should develop an algorithm that integrates the rejected scenarios as negative scenarios with the scenario specification.

Our CFG can be an executional model if we give more rigorous definition to it. With the executional model, the scenario specification can be simulated and analyzed dynamically. Since the dynamic analysis are another way for verifying the specification, we are considering to investigate them.
요 약 문

UML 2.0 시나리오 명세에 내재된 시나리오 분석기법

정확한 요구사항 명세는 정확한 산출물을 얻기 위해 필수적이며, 이러한 요구사항 명세는 최근의 방법론들에서 일반적으로 시나리오의 형태로 기술된다. 이러한 시나리오를 기술한 시나리오 명세는 의도되지 않지만 나타나게 된 시나리오를 찾을 수 있게 되는데, 이를 내재된 시나리오라 부른다. 한편 UML(Unified Modeling Language)이 보편화됨에 따라 그러한 시나리오는 UML 시나리오 명세로 기술되고 있다. 따라서 이 UML 시나리오 명세에 대해 서로 내재된 시나리오를 찾는 방법이 필요하게 된다. 또한 UML 시나리오 명세의 경우, 다수의 연구 대상이 되는 MSC(Message Sequence Chart)와는 다르게 상위 시나리오 명세인 IOD(Interaction Overview Diagram)에서 뿐만 아니라 하위 시나리오 명세인 SD(Sequence Diagram)에서도 내재된 시나리오가 발생하게 된다. 그러나 내재된 시나리오에 대한 기존 연구들은 주로 MSC를 다루고 있고, UML에서의 내재된 시나리오는 많은 제약조건들을 전제하고 있다. 이 연구에서 우리는 UML 2.0 시나리오 명세에서의 내재된 시나리오를 찾은 기법을 제안하고 있다. 그 기법은 먼저 정규화를 제공하여 내재된 시나리오를 보다 알고리즘에 의해 다양한 표기법을 지원하고, 또한 하나의 SD내에 존재하는 내재된 시나리오 와 IOD 단계에서 존재하는 것을 찾도록 한다. 다음으로 UML 시나리오 명세 내에 있는 프로세스를 찾기 위해 세어 호름 분석을 수행하는 알고리즘과, UML시나리오 명세와 세어 호름 분석의 결과물인 FSP(Finite State Process)라는 형식언어로 표현하는 알고리즘을 제안한다. 이들 알고리즘을 통해 나타나는 FSP를 LTSA(Labelled Transition System)라는 분석 도구의 입력으로 사용하여 최종적으로 내재된 시나리오를 찾게 된다. UML 시나리오 명세에서의 내재된 시나리오를 찾는 기법은 소프트웨어 디자이너에게 시나리오 명세를 면밀하게 검토할 수 있게 한다.
References


