Research Article

An Approach to Convert XMI Representation of UML 2.x Interaction Diagram into Control Flow Graph

Debasish Kundu,¹ Debasis Samanta,¹ and Rajib Mall²

¹School of Information Technology, Indian Institute of Technology, Kharagpur, Kharagpur 721302, India
²Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, Kharagpur 721302, India

Correspondence should be addressed to Debasish Kundu, dkundu@sit.iitkgp.ernet.in

Received 23 August 2011; Accepted 20 September 2011

Academic Editors: F. Ipate and A. Lastovetsky

Copyright © 2012 Debasish Kundu et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

For automation of many software engineering tasks such as program analysis, testing, and coverage analysis, it is necessary to construct a control flow graph. With the advancement of UML, software practitioners advocate to construct control flow graph from some of the UML design artifacts. UML 2.x supports the modeling of control flow information in interaction diagram by means of message sequences and different types of fragments like alt, opt, break, loop, and so forth. Leading UML modeling tools, namely MagicDraw, IBM’s Rational, and so forth export models in XMI format. Construction of control flow graph from the XMI representation of an interaction diagram is not straightforward as model elements of interaction diagram are captured in XMI by means of values of attributes of multiple tagged elements and correlations among these tagged elements is not explicitly specified. This paper proposes an approach for construction of control flow graph from XMI representation of UML 2.x interaction diagram. A prototype tool based on our approach has been developed which can be plugged in any computer-aided software engineering tool.

1. Introduction

UML has now become a standard modeling language in software industries [1]. Currently, UML models are being used in many software engineering tasks such as program analysis, testing, and coverage analysis [2–11]. For automation of these tasks, it is necessary to construct a control flow graph from UML models. UML 2.x supports the modeling of control flow information in an interaction diagram by means of message sequences and different types of fragments like alt, opt, break, loop, and so forth [12–14]. UML models are usually exported from UML modeling tools in XMI representation [15]. Unfortunately, construction of control flow graph from XMI representation of interaction diagram is not straightforward. The factors contributing to the complexity of construction of control flow graph are: (i) model elements of interaction diagrams are captured in XMI by means of values of attributes of multiple tagged elements, (ii) correlations among tagged elements are not explicitly specified [16], and (iii) control flow information is modeled by means of different types of fragments that can be nested fragment with arbitrary nesting depth.

Existing approaches [2, 4, 11, 17, 18] consider a set of mapping rules to construct control flow graph from interaction diagrams [12]. Mapping rules are in essence as follows. (1) Message, start and end of a fragment correspond to nodes of the control flow graph. (2) An edge is constructed between two nodes depending on various conditions such as type of nodes (message, start/end of a fragment) and type of fragments (alt, opt, loop, etc.). For applying these mapping rules to XMI representation of an interaction diagram, we need to do the following: (i) process the information associated with multiple tagged elements that correspond to a node in control flow graph, (ii) track the tagged elements stored in XMI representation between start and end of each operand of a fragment as well as between start and end of each fragment, and (iii) consider the interpretations of different fragments to map edges. The situation becomes complicated when an operand of a fragment contains a nested fragment of arbitrary nesting depth.
depth. In fact, the existing mapping rules are difficult to apply to the XMI representation of interaction diagram. This issue has been addressed in this paper. We propose an approach to parse the XMI representation of interaction diagram in an easy and tangible way and store model information in an intermediate data structure. This intermediate data structure enables us to apply the mapping rules to construct control flow graph. Apart from this, information retrieved from XMI representation can be embedded into control flow graph for many of the automated software engineering tasks such as test case generation, and coverage analysis. Based on our approach, a prototype tool XMI2CFG has been developed which can take any interaction diagram in XMI representation, parse it, construct its control flow graph, and visualize the graph to the users with details in it.

The rest of the paper is organized as follows. Few basic definitions, terminologies, and the control primitives for different types of fragments of interaction diagram have been discussed in Section 2. In Section 3, we discuss some issues of dealing with XMI representation for construction of control flow graph. Section 4 presents the proposed conversion approach from XMI of interaction diagram into control flow graph. We discuss our prototype tool in Section 5. Section 6 discusses related work and compares our work with the existing work. Finally, Section 7 concludes the paper.

2. Preliminaries

In this section, we first present few definitions and terminologies that have been used in the later sections. Then, we discuss control flow primitives for most commonly used fragments such as alt, opt, break, loop of interaction diagram.

2.1. Definitions and Terminologies

XMI. XMI stands for XML metadata interchange. It is a standard representation used for exchanging metadata information by means of extensible markup language (XML) [19].

Interaction Diagram. Interaction diagram is a UML behavioral diagram of a use case [12, 13]. This diagram models control flow by means of sequence of messages among objects of a system. The objects are shown as rectangular boxes arranged horizontally. The vertical lines falling from the rectangular boxes represent lifelines. The message is represented by means of an arrow connecting two lifelines attached to sender and receiver object of the message. Time is represented in the vertical direction showing the sequence of messages.

Message. A message refers to an instruction sent to an object. Sender object sends a message to a receiver object to invoke a method defined for that receiver object. Difference between message and method is that a message is a request for a receiver object to perform some task(s), and the message consists of a message name and a list of (zero or more) arguments. On the other hand, when a message is sent to a receiver object, a method with the same name and argument list as the message gets executed [20].

Fragment. A fragment is a group of set of messages together to show conditional flow in an interaction diagram. A fragment is also termed as interaction operator that operates on a group of operands, and each operand represents a set of messages that occur in a sequence under some guard condition. UML 2.x specification supports different types of fragments like alt (alternative), opt (optional), loop, break, and so forth [1]. A fragment specifies how execution of operands is to be interpreted. For example, alt fragment specifies that only one operand whose guard condition gets satisfied, would be executed.

Model Element. Model element refers to the elements in a UML interaction diagram such as the start of a fragment, and end of a fragment, message.

Control Flow Graph. Control flow graph for an interaction diagram say ItrDgm, is a directed graph CFG = (N, E), where \( N = \{n_1, n_2, \ldots, n_i, \ldots, n_d\} \) is a set of nodes and \( E = \{(n_i, n_i)\} \) is a set of edges. A node \( n_i \in N \) represents a model element of the ItrDgm, and an edge \( (n_i, n_i) \in E \) represents a control flow between two model elements say \( m_i \) (corresponding to \( n_i \)) and \( m_j \) (corresponding to \( n_j \)).

Nodes of the control flow graph are of two types: message node and fragment node representing message and fragment boundary, respectively. The message and fragment node is uniquely represented by a tuple \( \langle T, ID, B, RO, M, PR, RVa r \rangle \), where

\begin{enumerate}
  \item \( T \) is the type of the fragment or message. A fragment is of type like alt, opt, loop, and break, whereas message is of type synchronous, reply, and so forth.
  \item \( ID \) is the identification number of message/fragment. Each message/fragment is identified by a unique identification number.
  \item \( B \) is the start or end boundary of a fragment. The value of \( B \) is set as \( S \) and \( E \) for the start and end boundary of the fragment, respectively. It is null for message node.
  \item \( RO \) is a reference to the receiver object of the message corresponding to message node. It has the value null for fragment node.
  \item \( M \) is a method gets executed when RO receives the message corresponding to message node and is null for fragment node.
  \item \( PR \) is the set of parameters of the method \( M \), and is null for fragment node.
  \item \( RVa r \) is a return variable that keeps the value returned by the method \( M \). It is null for fragment node.
\end{enumerate}

Interaction. An interaction \( I \) is a set of zero or more message occurrences in an interaction diagram [1]. That is,
\[ I = \{ m_1, m_2, \ldots, m_n | n \geq 0 \} \], where \( m_1, m_2, \ldots, m_n \) are the \( n \) number of messages in the interaction diagram. If \( n = 0 \), then \( I \) is referred to as null interaction, and if \( n = 1 \), then \( I \) is referred to as a message. On the other hand, if \( n > 1 \), then \( I \) is referred to a set of messages of a fragment.

**Precedence Relation.** For any two interactions \( I_x \) and \( I_y \) in a set of interactions \( M_1 \), we say there is a precedence relation \( I_x \prec I_y \) if \( I_x \in M_1 \) occurs immediately before \( I_y \in M_1 \) according to the timing order in the interaction diagram. It implies that if there exists a precedence relation between \( I_x \) and \( I_y \) in \( M_1 \), then there would be no \( I_x \in M_1 \) such that \( I_x < I_y \) and \( I_x < I_y \). This relation satisfies the following properties.

(a) If \( I_x \prec I_y \), then \( I_x \not\simeq I_y \), where \( I_x, I_y \in M_1 \) (asymmetric).
(b) If \( I_x \prec I_y \) and \( I_y < I_z \), then \( I_x \not\simeq I_z \), where \( I_x, I_y, I_z \in M_1 \) (non-transitive).
(c) \( I_x \not\simeq I_x \) (non-reflexive).

**Null Precedence Relation.** Let \( P(M_1) \) be a set of precedence relations on \( M_1 \). Note that if \( M_2 \) is the set of messages of a fragment \( f_i \) and an operand of the fragment \( f_i \) contains a single message \( m' \), then there would be no precedence relation containing the message \( m' \) in \( P(M_1) \). To ensure the existence of some precedence relation in \( P(M_1) \) that would contain \( m' \) as an operand, null interaction may be assumed to have occurred before or after \( m' \). The precedence relation between \( m' \) and null interaction is called null precedence relation, which is defined below.

A precedence relation \( (I_x \prec I_y) \) of \( P(M_1) \) is called a null precedence relation if either \( I_x \) or \( I_y \) is null interaction. If \( I_x \) is the null interaction, then \( I_x \prec I_y \) is called the left null precedence relation and written as \( \Lambda \prec I_y \). On the other hand, if \( I_y \) is the null interaction, then \( I_x < I_y \) is called the right null precedence relation and written as \( I_x < \Lambda \).

2.2. Control Flow Primitives. We now consider control flow primitives for most commonly used fragments alt, opt, break, and loop.

**Alt Fragment.** The alt fragment is used to capture alternative flows by means of multiple operands [12]. In Figure 1(a), an interaction diagram is shown with an alt fragment. The alt fragment has two operands containing the messages \( m_2 \) and \( m_3 \), respectively.

As observed in Figure 1(a), after receiving the message \( m_1 \) from ObjectA, ObjectB sends the message \( m_2 \) to itself if \( C \) becomes true, otherwise, ObjectB sends the message \( m_3 \) to ObjectC. Finally, reply message \( m_4 \) is sent from ObjectB to ObjectA. Figure 1(b) depicts corresponding graph representation for the interaction diagram containing alt fragment. Note that four nodes \( m_1, m_2, m_3, \) and \( m_4 \) represent four messages \( m_1, m_2, m_3, \) and \( m_4 \), respectively, and two nodes alt\(_{1}\) and alt\(_{2}\) correspond to the start and end of the alt fragment, respectively. Two alternative flows are modeled by two outgoing edges from alt\(_{1}\) and two incoming edges at the node alt\(_{2}\) in the graph representation. Note that this graph representation contains the same information as in interaction diagram. In this case, control flow is explicitly modeled for alt fragment in interaction diagram and hence, control flow graph would be the same as the graph representation (see Figure 1(b)).

**Opt Fragment.** The opt fragment has only one operand that is executed optionally [12]. In Figure 1(c), an interaction diagram contains an opt fragment with two messages \( m_2 \) and \( m_3 \). These two messages are executed if the guard condition \( C_1 \) becomes true. Figure 1(d) depicts graph representation for the interaction diagram (in Figure 1(c)). Note that when the guard condition \( C_1 \) becomes false, the control flow is transferred from the start of opt fragment to its end. This control flow is implicitly captured in interaction diagram and so in its graph representation. However, this control flow is modeled explicitly in control flow graph as an edge from the node opt\(_{1}\) (corresponding to the start of opt fragment) to the node opt\(_{2}\) (corresponding to the end of opt fragment) as shown in Figure 1(c).

**Break Fragment.** The break fragment is used to capture an exit point of the systems [12]. In Figure 1(f), we see that if guard condition \( C_1 \) becomes true, then it exits after execution of two messages \( m_2 \) and \( m_3 \), following which no further message (i.e., \( m_4 \) and \( m_5 \)) would be executed. Note that the control flow which is transferred from the start of break fragment to the message \( m_4 \) when the guard condition \( C_1 \) becomes false is implicitly captured in both interaction diagram and its graph representation (see Figure 1(g)). The control flow for the break fragment is modeled explicitly by reducing the node break\(_{1}\) (corresponding to the end of the break fragment) as a sink node and introducing an edge from the node break\(_{2}\) to the node \( m_4 \) in control flow graph (see Figure 1(h)).

**Loop Fragment.** The loop fragment is used to model repetitive interactions [12]. In Figure 1(i), we see that message \( m_2 \) of the loop fragment gets executed repeatedly until the guard condition \( C_1 \) associated with loop fragment becomes false. When \( C_1 \) becomes false, loop terminates and execution of the message \( m_3 \) commences. Figure 1(j) shows the graph representation for the interaction diagram Figure 1(i). The control flow for the loop fragment is explicitly modeled in the control flow graph by a back edge from the node loop\(_{1}\) to the node loop\(_{0}\) and another loop exit edge from loop\(_{0}\) to \( m_1 \) (see Figure 1(k)). There are many more fragments such as par and ref defined in UML 2.x [12], whose graph representations and control flow graphs are not being discussed in this paper due to space limitation; however, they can be treated likewise.

3. Issues with Construction of Control Flow Graph

In this section, we discuss the issues that arise in construction of control flow graph from XMI representation of interaction
Figure 1: The control primitives for the fragments alt, opt, break, and loop of interaction diagram.
Let us consider a simple interaction diagram as shown in Figure 2(a). The interaction diagram in Figure 2(a) has five messages \( m_1, m_2, m_3, m_4 \), and \( m_5 \) and two fragments \( opt_1 \) and \( alt_1 \). Figure 2(b) shows the graph representation with usual bearings as discussed in the previous section. The XMI representation of the interaction diagram as exported from MagicDraw 16.0 tool \(^{21}\) appears as shown in Figure 3. All tagged elements in the XMI representation are referred by line numbers printed on the left side (see Figure 3).

Let us consider the mapping rules used for construction of a graph representation from an interaction diagram \(^{2, 4, 11, 17, 18}\). As per the mapping rules: (1) message, start and end of a fragment are mapped into nodes in the graph representation; (2) an edge is considered between two nodes representing (a) two messages where one message follows another, or (b) one message and start of a fragment where the fragment follows the message, or (c) end of a fragment and one message where message follows the fragment, or (d) end of a fragment and start of another fragment where second fragment follows the first one; (3) for each fragment, edges are drawn (a) from the node representing the start of a fragment to the node representing the first element (message, fragment) in each of its operands and (b) from the node corresponding to the last element of each operand of the fragment to the node corresponding to end of that fragment.

Let us see how the information such as sender and receiver objects and their classes of a message (say, \( m_1 \)) can be obtained from XMI representation. For this, we need to identify a tagged element that specifies the method whose name is the same as the message \( m_1 \). In XMI representation, we observe that name attribute of the tagged element \( "ownedOperation" \) (representing an operation) at line 4 contains the value as same as the name of the message \( m_1 \). The corresponding method is defined as \( call \) event \( "443" \) in XMI representation (line 66). The \( call \) event \( "443" \) actually corresponds to the \( receive \) event \( "441" \) of the message \( "439" \) (lines 66 and 25). Note that send and receive events of the message \( "439" \) correspond to the tagged elements containing the attribute value as \( "xmi: type=MessageOccurrenceSpecification" \) (lines 24 and 25). These send and receive events occur at the object lifelines referred as the attribute \( "covered=.376" \) and \( "covered=.393" \) (lines 24 and 25), respectively. The tagged elements corresponding to the lifelines \( "xmi:id=.376", "xmi:id=.393" \) refer to the associated objects as the attribute "\( xmi: type=MessageOccurrenceSpecification " \) (lines 24 and 25).

Their class types are obtained as \( ClassA \) and \( ClassB \) from tagged elements at lines 2 and 3. All these information are for the message \( m_1 \), which imply that \( ObjectA \) of \( ClassA \) sends a message \( m_1 \) to \( ObjectB \) of \( ClassB \). In other words, for the node \( m_1 \) (corresponding to the message \( m_1 \)) in graph representation, we are to retrieve the information encapsulated in the tagged elements 4, 66, 25, 24, 62, 20, 16, 2, 21, 17 and 3 of the XMI representation.
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <packagedElement xmi:type="uml:Class" xmi:id="_386" name="ClassA"/>
3.  <packagedElement xmi:type="uml:Class" xmi:id="_406" name="ClassB"/>
4.  <ownedOperation xmi:id="_689" name="m1" visibility="public"/>
5.  <ownedOperation xmi:id="_691" name="m2" visibility="public"/>
6.  </packagedElement>
7.  <packagedElement xmi:type="uml:Class" xmi:id="_426" name="ClassC"/>
8.  <ownedOperation xmi:id="_699" name="m3" visibility="public"/>
9.  </packagedElement>
10. <packagedElement xmi:type="uml:Class" xmi:id="_670" name="ClassD"/>
11. <ownedOperation xmi:id="_701" name="m4" visibility="public"/>
12. </packagedElement>
13.  ...
14. <packagedElement xmi:type="uml:Collaboration" xmi:id="_351" name="SDEmp">
15.   <ownedBehavior xmi:type="uml:Interaction" xmi:id="_350" name="SDEmp">
16.     <ownedAttribute xmi:id="_377" name="ObjectA" visibility="private" type="_386"/>
17.     <ownedAttribute xmi:id="_394" name="ObjectB" visibility="private" type="_406"/>
18.     <ownedAttribute xmi:id="_411" name="ObjectC" visibility="private" type="_426"/>
19.     <ownedAttribute xmi:id="_428" name="ObjectD" visibility="private" type="_670"/>
20.     <lifeline xmi:id="_376" name="" visibility="public" represents="_377"/>
21.     <lifeline xmi:id="_393" name="" visibility="public" represents="_394"/>
22.     <lifeline xmi:id="_410" name="" visibility="public" represents="_411"/>
23.     <lifeline xmi:id="_427" name="" visibility="public" represents="_428"/>
25.     <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_441" message="_439" event="_443"/>
26.     <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_529" message="_528" event="_532"/>
27.     <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_530" message="_528"/>
28.     <fragment xmi:type="uml:CombinedFragment" xmi:id="_453" interactionOperator="opt">
29.       <operand xmi:id="_454" name="" visibility="public">
30.         <guard xmi:id="_468" />
31.         <specification xmi:type="uml:LITERALSTRING" xmi:id="_469" name="" value="C1"/>
32.       </guard>
33.     </fragment>
34.     <fragment xmi:type="uml:CombinedFragment" xmi:id="_487" visibility="public" interactionOperator="alt">
35.       <operand xmi:id="_488" name="" visibility="public">
36.         <guard xmi:id="_537" />
37.         <specification xmi:type="uml:LITERALSTRING" xmi:id="_538" name="" value="C2"/>
38.       </guard>
39.     </fragment>
40.   </ownedBehavior>
41. </packagedElement>

Figure 3: Continued.
51. <fragment xmi:type="uml:Interaction" xmlns="" xmi:id=".899" name="">  
52. <fragment xmi:type="uml:MessageOccurrenceSpecification" xmlns="" xmi:id=".472" message=".471"/>
53. <fragment xmi:type="uml:MessageOccurrenceSpecification" xmlns="" xmi:id=".473" message=".471"/>
54. <message xmi:id=".471" receiveEvent=".473" sendEvent=".472"/>
55. </operand>
56. </fragment>
57. </fragment>
58. <fragment xmi:type="uml:MessageOccurrenceSpecification" xmlns="" xmi:id=".723" message=".722"/>
59. <fragment xmi:type="uml:MessageOccurrenceSpecification" xmlns="" xmi:id=".724" message=".726"/>
60. <fragment xmi:type="uml:MessageOccurrenceSpecification" xmlns="" xmi:id=".726" message=".733"/>
61. <message xmi:id=".528" name="" receiveEvent=".441" sendEvent=".440"/>
62. <message xmi:id=".529" name="" receiveEvent=".530" sendEvent=".529"/>
63. </ownedBehavior>
64. </packageElement>
65. </packageElement>
66. <packageElement xmi:type="uml:CallEvent" xmlns="" xmi:id=".443" operation=".689"/>
67. <packageElement xmi:type="uml:CallEvent" xmlns="" xmi:id=".479" operation=".691"/>
68. <packageElement xmi:type="uml:CallEvent" xmlns="" xmi:id=".726" operation=".699"/>
69. <packageElement xmi:type="uml:CallEvent" xmlns="" xmi:id=".737" operation=".701"/>
70. <packageElement xmi:type="uml:CallEvent" xmlns="" xmi:id=".532"/>
71. . . .
72. </xmi:XMI>

**Figure 3:** XMI representation of Figure 2(a) exported from Magic Draw 16.0.

Table 1: Association of each node in graph representation with a group of tagged elements in XMI representation. It is evident from Table 1 that the information of each node in graph representation is not only associated with multiple tagged elements but also spread in different places of XMI representation. This implies that the mapping of model elements from XMI representation of interaction diagram to the nodes of its graph representation is not necessarily straightforward. In fact, conversion process becomes complex when we try to apply the mapping rules to construct edges in graph representation for the fragments of an interaction diagram. This is due to the association of multiple tagged elements in XMI representation with each model element in interaction diagram and tracking the tagged elements between start and end of each operand of a fragment as well as between start and end of each fragment. This is difficult because tagged elements for a fragment are stored in an unstructured way, that is, intermingled with tagged elements of other fragments. The situation becomes more complicated when an operand of a fragment contains a nested fragment of arbitrary nesting depth. That is why the mapping rules are indeed too difficult to apply straightforward to XMI representation.

<table>
<thead>
<tr>
<th>Nodes in Figure 2(b)</th>
<th>Tagged elements in Figure 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>m&lt;sub&gt;i&lt;/sub&gt;</td>
<td>4, 6, 6, 25, 24, 62, 20, 16, 2, 21, 17, 3</td>
</tr>
<tr>
<td>opt&lt;sub&gt;k&lt;/sub&gt;</td>
<td>5, 7, 53, 42, 21, 17, 3</td>
</tr>
<tr>
<td>alt&lt;sub&gt;q&lt;/sub&gt;</td>
<td>33, 34, 2</td>
</tr>
<tr>
<td>alt&lt;sub&gt;r&lt;/sub&gt;</td>
<td>8, 68, 59, 58, 39, 21, 17, 3, 22, 18, 2</td>
</tr>
<tr>
<td>m&lt;sub&gt;t&lt;/sub&gt;</td>
<td>6, 69, 3, 27, 20, 17, 3, 22, 18, 2</td>
</tr>
<tr>
<td>alt&lt;sub&gt;s&lt;/sub&gt;</td>
<td>63, 26, 27, 20, 17, 3, 22, 18, 2</td>
</tr>
</tbody>
</table>

Similar to a message, start and end of a fragment also correspond to multiple tagged elements in XMI representation. For example, the tagged elements at 34 and 42 correspond to start of two operands of the fragment alt<sub>l</sub> (xmi:id=".487"; line 33) and tagged elements at 41 and 49 correspond to the end of two operands of the fragment alt<sub>r</sub>. Furthermore, the tagged elements 33 and 50 correspond to the start and end of the fragment alt<sub>l</sub>, respectively. Therefore, the node alt<sub>s</sub> (representing the start of the fragment alt<sub>l</sub>) would correspond to the tagged elements at 33, 34, and 42, whereas the tagged elements at 41, 49, and 50 would correspond to the node alt<sub>q</sub> (representing the end of the fragment alt<sub>r</sub>). Note that correspondence among tagged elements (33, 34 and 42) or (41, 49 and 50) are not explicitly specified in XMI representation.

4. Proposed Approach

To overcome the difficulties pointed out in the previous section, we have proposed the concept of interaction sequence. Here, the term interaction signifies either a single message or a set of messages of a fragment. We propose a solution to extract the interaction sequences for each fragment precisely
and then map them to control flow graph following the mapping rules as discussed in Section 2.2. The major steps in our approach are shown in Figure 4. The first step in our approach is to synthesize metainformation from XMI representation of a given UML interaction diagram. These metainformation are then processed to identify the fragment set and message set. These sets are used to determine the nodes of graph representation in the second step. The fragment structure is obtained in next step. The fourth step determines the edges among nodes of the graph. The edges are labeled in the fifth step. This completes the revealing graph representation of the input interaction diagram. The last step applies set of control flow rules for different types of fragments to obtain the control flow graph. We now discuss the various steps in detail in the following.

**Step 1** (identifying the fragments and their message sets). The first step of our conversion approach is to identify a set of fragments and their message sets from values of the attributes of the tagged elements in XMI representation. For this, we use standard SAX (Simple API for XML) parser [22]. Note that SAX is an event-based parser. As the name implies, SAX parser generates events while reading an XML document. The events are related to element opening tags, element closing tags, content of elements, and so forth in the XML document. These events notify an application by calling appropriate event handlers implemented by the application. For example, two event handlers: `startElement()` and `endElement()` are invoked when parser reads the opening tag and closing tag of an element, respectively. During invocation of event-handlers, attributes of the tagged elements are passed as a list of parameters. The processing of values of the attributes of the elements in XMI representation is necessary to identify the fragments and their message sets. For this, following the steps need to be carried out.

**(a) Storing Values from Tagged Elements.** We store values of the attributes of tagged elements as objects of the following classes: `EMessage`, `EMessageEvent`, `ECallEvent`, `EOperation`, `EFragment`, `EOperand`, `EObject`, `EClass`, and `ELifeline` as referred in Table 2. The association of tagged element and value of its attribute “xmi : type” (see Figure 3) with an event is shown in Table 2. This table also depicts which class of object is used to store the values of attributes of a tagged element. Attributes of the classes are shown in Figure 5. The values of the most of the attributes for the objects of these classes would be obtained directly from the values of the attributes of the corresponding tagged elements. Moreover, values of some attributes of objects are to be either obtained from other objects or set with specific value. For example, value of the attribute “type” of an object say, `aMEvent` (of `EMessageEvent` class) corresponding to a send event is set as `aMEvent.type = “sendEvent”`, whereas for receive event, it is set as `aMEvent.type = “receiveEvent”`. If an operand contains an inner fragment with its Id as fi and `aOperand` (of `EOperand` class) is the object corresponding to the operand, then fi would be stored in “MessageList” of `aOperand` along with Ids of other messages in the same sequence as they appear in the operand. In a nutshell, after storing the values of attributes of the tagged elements we would obtain a set of lists of objects.
Table 2: Classes to store information of elements in XMI representation.

<table>
<thead>
<tr>
<th>Event name</th>
<th>XMI tagged element</th>
<th>Value of attribute “xmi: type”</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>⟨ownedOperation⟩</td>
<td>-</td>
<td>EOperation</td>
</tr>
<tr>
<td>CallEvent</td>
<td>⟨packagedElement⟩</td>
<td>“uml: CallEvent”</td>
<td>ECallEvent</td>
</tr>
<tr>
<td>Message</td>
<td>⟨message⟩</td>
<td>“uml: MessageOccurrenceSpecification”</td>
<td>EMessageEvent</td>
</tr>
<tr>
<td>MessageEvent</td>
<td>⟨fragment⟩</td>
<td>“uml: CombinedFragment”</td>
<td>EFragment</td>
</tr>
<tr>
<td>Lifeline</td>
<td>⟨lifeline⟩</td>
<td>-</td>
<td>ELifeline</td>
</tr>
<tr>
<td>CombinedFragment</td>
<td>⟨fragment⟩</td>
<td>“uml: CombinedFragment”</td>
<td>EFragment</td>
</tr>
<tr>
<td>Operand</td>
<td>⟨operand⟩</td>
<td>-</td>
<td>EOperand</td>
</tr>
<tr>
<td>Class</td>
<td>⟨packageElement⟩</td>
<td>“uml: class”</td>
<td>EClass</td>
</tr>
<tr>
<td>Object</td>
<td>⟨ownedAttribute⟩</td>
<td>-</td>
<td>EObject</td>
</tr>
</tbody>
</table>

![Class diagram for metadata of interaction diagram in XMI.](image)

Figure 5: Class diagram for metadata of interaction diagram in XMI.

of the following classes: (a) EMessage, (b) EMessageEvent, (c) ECallEvent, (d) EOperation, (e) EFragment, (f) EOperand, (g) EObject, (h) EClass, and (i) ELifeline (see Table 2).

(b) Synthesizing Details of a Message. We need to synthesize detail information of a message such as name of object method that gets executed when the message is sent, sender object and its class, receiver object and its class, parameters of the message, and return variable (if any). In other words, we are to find the values of the attributes: MethodName, SenderObject, SenderClass, ReceiverObject, ReceiverClass, SeqNumber, ParameterList, and ReturnVar of the object aMessage (of EMessage class) corresponding to a message from the data stored in other objects. To do this, it is necessary to find relationships among classes of these objects. For this, we consider the inherent structure of XMI representation as well as metamodel of interaction diagram as given in UML superstructure specification [1]. The class diagram for representing the relationships among the classes which are used to store tagged elements is shown in Figure 5. Note that one message corresponds to two message events, namely, send and receive events, which may or may not
correspond to a call event. This is because only receive event of a message (other than reply message) and send event of reply message correspond to call events. Further, a call event (except for reply message) corresponds to an operation. All this information is represented in the class diagram (see Figure 5). To synthesize values of attributes for an object $aMessage$ of $EMessage$ class using associations (as shown in Figure 5), we identify two objects $sMEvent$, and $rMEvent$ from the list of $EMessage$ objects such that (i) $sMEvent$ and $rMEvent$ correspond to send and receive events of the message corresponding to $aMessage$, respectively, (ii) if $aMessage$ represents a reply message, then $sMEvent$ should correspond to a call event (represented by an object say $aCallEvent$ in the list of $ECallEvent$ objects) otherwise, $rMEvent$ would correspond to a call event (corresponding to $aCallEvent$) that is associated with an operation (represented by an object say, $aOperation$ in the list of $EOperation$ objects). In other words, the conditions are to be satisfied as follows.

(a) $sMEvent.type = "sendEvent" \& sMEvent.Id = aMessage.SendEventId.$

(b) $rMEvent.type = "receiveEvent" \& rMEvent.Id = aMessage.SendEventId.$

(c) $(aCallEvent.Id = rMEvent.CallEventId \& aOperation.Id = aCallEvent.OperationId) \lor (aCallEvent.Id = sMEvent.CallEventId \& aMessage.messageType = "reply").$

These send and receive events (represented by $sMEvent$ and $rMEvent$) must occur at two lifelines (represented by two objects say $sLifeline$ and $rLifeline$ in the list of $ELifeline$ objects). These two lifelines ($sLifeline$ and $rLifeline$) are associated with two objects $sObject$ and $rObject$ in the list of $EObject$ objects, respectively. All these imply the satisfiability of the following conditions.

(d) $sLifeline.Id = sMEvent.LifelineId \& rLifeline.Id = rMEvent.LifelineId.$

(e) $sObject.Id = sLifeline.ObjectId \& rObject.Id = rLifeline.ObjectId.$

(f) $sClass.Id = sObject.ClassId \& rClass.Id = rObject.ClassId.$

Using the objects: $sObject$, $rObject$, $sClass$, $rClass$, $aCallEvent$, and $aOperation$ as referred in conditions (a)–(f), we obtain the values of instance variables: $MethodName$, $SenderObject$, $SenderClass$, $ReceiverObject$, $ReceiverClass$, and $SeqNumber$ of the $aMessage$ object in the list of “$EMessage$” objects as follows.

$aMessage.MethodName = aOperation.Name$

(if $aMessage.messageType \neq "reply"$).

$aMessage.SenderObject = sObject.Name.$

$aMessage.ReceiverObject = rObject.Name.$

$aMessage.SenderClass = sClass.Name.$

$aMessage.ReceiverClass = rClass.Name.$

$aMessage.SeqNumber = aCallEvent.SeqNo.$

$aMessage.parameterList = aOperation.parameterList.$

Next, we are to find the return variable associated with the message say, $m_r$, corresponding to object $aMessage$. Note that the return variable of $m_r$ is represented as a reply message and specified immediately after $m_r$ in XMI representation. For this, we identify an object $bMessage$ from a list of $EMessage$ objects such that

(g) $bMessage.SenderObject = aMessage.ReceiverObject.$

(h) $bMessage.ReceiverObject = aMessage.SenderObject.$

(i) $bCallEvent.SeqNo. - aCallEvent.SeqNo. = 1.$

(j) $bMessage.messageType = "reply".$

(k) $aMessage.messageType \neq "reply".$

Here, $bCallEvent$ refers to a call event for the message corresponding to $bObject$. If such $bMessage$ object exists for the $aMessage$, then we set $ReturnVar$ of the $aMessage$ object same as the return value stored in $argumentValueList[0]$ of the $bMessage$ (because reply message can not have more than one return value). In other words,

$$aMessage.ReturnVar = bMessage.argumentValueList[0]$$

Once value of $aMessage.ReturnVar$ is synthesized from the $bMessage$ object, then $bMessage$ becomes redundant and it should be removed from the list of $EMessage$ objects.

(c) Finding Message Sets and Set of Fragments. After synthesis of values of the attributes of an object $aMessage$ in the list of $EMessage$ objects, we need to find a set of messages $M_seq$, set of reply messages $R_seq$ (using the value of attribute “messageType”) from that list. A set of fragments $F_seq$ is also to be obtained from the list of $EFragment$ objects. $M_seq$, $R_seq$, $F_seq$ can be determined as follows.

$$M_seq = \{ aMessage.Id \mid aMessage \text{ exists in the list of } EMessage \text{ objects} \}.$$  

$$R_seq = \{ aMessage.Id \mid aMessage.Id \in M_seq \& aMessage.messageType = "reply" \}.$$  

$$F_seq = \{ aFragment.Id \mid aFragment \text{ exists in the list of } EFragment \text{ objects} \}.$$  

(d) Finding Message Sets of Fragments. To determine the set of messages of a fragment $f_i \in F_seq$, we identify the $aFragment$ object (corresponding to the fragment $f_i$) from the list of $EFragment$ objects and then the set of $EOperand$ objects $S_{EOperands}$ such that an object $aOperand \in S_{EOperands}$ is associated with $aFragment$ object,
that is, $\text{aFragment.OperandList}$ contains the $\text{aOperand.Id}$. In other words,

$$S_{E\text{Oprds}}^f = \{ \text{aOperand} \mid \text{aOperand exists in the list of EOperand objects} \}$$

$$\text{aFragment.OperandList contains aOperand.Id} \& f_i \rightarrow \text{aFragment}. \}.$$  

We then find the set of $\text{EMessage}$ objects $S_{\text{EMsgs}}^f$ for the fragment $f_i$ such that aMessage $\in S_{\text{EMsgs}}^f$ is associated with an object $\text{aOperand} \in S_{E\text{Oprds}}^f$, that is, $\text{aOperand.MessageList}$ contains aMessage.Id. In other words,

$$S_{\text{EMsgs}}^f = \{ \text{aMessage} \mid \text{aMessage exists in the list of EMessage objects} \}$$

$$\text{aOperand.MessageList contains aMessage.Id} \& \text{aOperand} \in S_{E\text{Oprds}}^f \}.$$  

We may note that for the nested fragment $f_i$, $S_{\text{EMsgs}}^f$ would contain an object $\text{aFragment}$ in the list of $\text{EFragment}$ objects and therefore, for each $\text{aFragment}$ (corresponding to a fragment $f_j$) in $S_{\text{EMsgs}}^f$, we update $S_{\text{EMsgs}}^f$ as

$$S_{\text{EMsgs}}^f = S_{\text{EMsgs}}^f \cup S_{\text{EMsgs}}^f - \{ \text{aFragment} \},$$  

where $S_{\text{EMsgs}}^f$ is the set of $\text{EMessage}$ objects for the fragment $f_j$.

Once the update of $S_{\text{EMsgs}}^f$ is complete, we obtain the set of messages $M_{f_i}$ of the fragment $f_i$ from $S_{\text{EMsgs}}^f$ as

$$M_{f_i} = \{ \text{aMessage.Id} \mid \text{aMessage} \in S_{\text{EMsgs}}^f \}.$$  

Step 2 (determining the nodes). Initially, control flow graph CFG $(N, E)$ is empty. That is, the set of nodes $N$ and set of edges $E$ are both null. In this step, we determine the message nodes and fragment nodes (i.e., $N$) of CFG. For this, we use the set of messages $M_{\text{seq}}$, set of fragments $F_{\text{seq}}$ as obtained in Step 1.

(a) For each fragment $f_i \in F_{\text{seq}}$, we add two fragment nodes $f_i^{\text{start}}$ and $f_i^{\text{end}}$ into $N$ of CFG $(N,E)$, where $f_i^{\text{start}}$ and $f_i^{\text{end}}$ represent the start and end of the fragment $f_i$, respectively. For each fragment node, we consider (i) the values of instance variables: $\text{type}(T)$, $\text{ID}$ of the $\text{EFragment}$ object corresponding to the fragment $f_i$ and (ii) the boundary ($B$) of the fragment $f_i$ (i.e., start boundary or end boundary represented by the fragment node). The value of attribute ($B$) for the start boundary and end boundary is $S$ and $E$, respectively. For fragment node, the values of the attributes: $\text{RO}$, $\text{M}$, $\text{PR}$, and $\text{RVar}$ of corresponding tuple $(T, \text{ID}, B, \text{RO}, M, \text{PR}, \text{RVar})$ would be null.

(b) For each $m_i \in M_{\text{seq}}$, we add a message node $m_i$ into $N$. To obtain the values of associated attributes: $T$, $\text{ID}$, $\text{RO}$, $\text{M}$, $\text{PR}$, and $\text{RVar}$ of the tuple for the message node $m_i$, we consider the values of the instance variables: $\text{messageType}(T)$, $\text{Id}$, $\text{ReceiverObject}(\text{RO})$, $\text{MethodName}(\text{M})$, $\text{ParameterList}$ and $\text{ArgumentValueList}$ (set of parameters and their values, PR), and $\text{ReturnVar}(\text{RVar})$ (if any) of the $\text{EMessage}$ object corresponding to the message node $m_i$. For message node, value of the attribute $B$ would be null.

(c) For each node in $N$ of CFG $(N, E)$, we store corresponding tuple $(T, \text{ID}, B, \text{RO}, M, \text{PR}, \text{RVar})$ in a table, called Node table. The entries corresponding to message nodes in Node table would be in a sequence as same as the sequence number $(\text{SeqNumber})$ of corresponding EMessage objects.

Step 3 (determining fragment structure). In order to find the edges among nodes in $N$ of CFG, we need to determine the hierarchy structure of the fragments formed by the set of messages of interaction diagram $(M_{\text{seq}})$. In other words, we are to find the outermost fragments (the fragments that are not contained in another fragment) built by $M_{\text{seq}}$ and then determine the inner fragments contained in each fragment.

(a) To find the outermost fragments formed by the messages in $M_{\text{seq}}$, we need to identify the minimal set of fragments $F' \subseteq F_{\text{seq}}$ that together correspond to the largest subset of $M_{\text{seq}}$. For this, we follow two steps.

(i) Initially, set $F' = F_{\text{seq}}$ (set of all fragments in interaction diagram).

(ii) Exclude the fragment $f_i$ from $F'$ if there exists another fragment $f_j \neq f_i$ and $f_j \in F'$ such that $M_{f_i} \subseteq M_{f_j}$, where $M_{f_i}$ and $M_{f_j}$ are the sets of messages of the fragments $f_i$ and $f_j$, respectively. Repeat this step until no such fragment $f_j$ in $F'$ can be excluded. The resultant set $F'$ is the minimum set of fragments that can replace the largest subset of $M_{\text{seq}}$.

We then replace the subset of $M_{\text{seq}}$ corresponding to the $M_{f_i}$ for each fragment $f_i \in F'$ by the ID of the fragment $f_i$. This implies that the message set $M_{\text{seq}}$ covers all fragments in $F'$ as outermost fragments.

(b) To determine the inner fragments contained in each fragment $f_i$ in $F_{\text{seq}}$, we need to find the minimal set of fragments $F'' \subseteq F_{\text{seq}}$ that together correspond to the largest subset of $M_{f_i}$ (messages of the fragment $f_i$). For this, we follow two steps given below.

(i) Find the set of fragments $F'' \subseteq F_{\text{seq}}$ such that the set of messages say, $M_{f_i}$ of a fragment $f_i \in F''$ corresponds to the subset of $M_{f_i}$ such that $i \neq k$.

(ii) Exclude the fragment $f_j$ from $F''$ if there exists another fragment $f_j \neq f_k$ and $f_j \in F''$ such that
Let \( M_{f_k} \subseteq M_{f_j} \), where \( M_{f_k} \) and \( M_{f_j} \) are the sets of messages of the fragments \( f_k \) and \( f_j \), respectively. Repeat this step until no such fragment \( f_k \) in \( F'' \) can be excluded. The resultant set \( F'' \) is the minimum set of fragments that can replace the largest subset of the \( M_{f_i} \).

We then replace that largest subset of \( M_{f_i} \) by the IDs of the fragments in \( F'' \).

**Step 4** (determining the edges). After determining the hierarchy structure of the fragments formed by the set of messages \( M_{seq} \) of the interaction diagram, we find the edges among nodes in \( N \) of the CFG using the following steps.

(a) We apply the *precedence relation* \( \prec \) on \( M_{seq} \) and obtain a set of *precedence relations* say, \( P(M_{seq}) \). For this, we consider the *sequence number* of all messages (i.e., SeqNumber of corresponding EMessage objects) in \( M_{seq} \) and in case \( M_{seq} \) contains an *interaction* (i.e., fragment \( f_i \)), then we use the sequence number of a message in the \( M_{f_i} \) (message set of \( f_i \)). If \( P(M_{seq}) \) includes a precedence relation \( (I_x \prec I_y) \) such that \( I_x \in F_{seq} \) or \( I_y \in F_{seq} \), then \( P(M_{seq}) = P(M_{seq}) \cup P(M_{f_k}) \), where \( P(M_{f_k}) \) is the set of precedence relations on the set of messages of the fragment \( f_k \) and \( f_k = I_x \) or \( I_y \). The unions are repeated for all fragment operands present in the precedence relations of \( P(M_{seq}) \). The reason for union operation in computation of \( P(M_{seq}) \) is the presence of some fragment operand \( f_k \) in some precedence relation, which implies that precedence relations among the messages in the fragment \( f_k \) also need to be considered. Note that we compute \( P(M_{f_k}) \) using sequence of message Id(s) in the EOperand objects associated with EFragment object corresponding to the fragment \( f_k \).

(b) Considering a precedence relation \( (I_x \prec I_y) \) in \( P(M_{seq}) \), we draw an edge following the set of rules as mentioned below.

(i) If \( I_x, I_y \in M_{seq} \), then we draw an edge from the message node (corresponding to \( I_x \)) to the message node (corresponding to \( I_y \)). This edge implies that the message \( I_x \) occurs immediately before the message \( I_y \).

(ii) If \( I_x \in M_{seq} \), \( I_y \in F_{seq} \), then an edge is drawn from the message node (corresponding to the message \( I_x \)) to the fragment node (corresponding to the start of the fragment \( I_y \)). The signification of this edge is that the message \( I_x \) occurs immediately before the start of the interaction that corresponds to the fragment \( I_y \).

(iii) If \( I_x \in F_{seq} \) and \( I_y \in M_{seq} \), then we draw an edge from the fragment node (corresponding to the end of the fragment \( I_x \)) to the message node (corresponding to the message \( I_y \)). This edge implies that the message \( I_x \) occurs immediately after the end of the interaction that corresponds to the fragment \( I_y \).

(iv) If \( I_x, I_y \in F_{seq} \), then an edge is drawn from the fragment node (corresponding to the end of the fragment \( I_x \)) to the fragment node (corresponding to the start of the fragment \( I_y \)). The signification of this edge is that the end of the interaction which corresponds to the fragment \( I_x \) occurs immediately before the start of the interaction that corresponds to the fragment \( I_y \).

(c) Next, we draw an edge corresponding to each *left null precedence relation* \( (\Lambda < I) \in P(M_{seq}) \). Note that the \( (\Lambda < I) \) implies that \( I \) is the first *interaction* in a fragment \( f_k \in F_{seq} \) and that *interaction* \( I \) is either a message or a set of messages of the inner fragment \( f_i \) of the \( f_k \) and \( f_i, f_k \in F_{seq} \), where \( f_i \neq f_k \). Thus, we identify the \( M_{f_k} \) of a fragment \( f_k \in F_{seq} \) such that \( M_{f_k} \) contains \( I \). It may be noted that \( M_{f_k} \) contains fragments other than messages because the subset of \( M_{f_k} \) has been replaced by the fragments in the preceding step (determine fragment structure). If \( I \in F_{seq} \), then we draw an edge from the fragment node (corresponding to start of the fragment \( f_k \)) to the fragment node (representing the start of the fragment \( I \) otherwise, we draw an edge from the fragment node (corresponding to start of the fragment \( f_k \)) to the message node (corresponding to the message \( I \)). The edge thus obtained for a *left null precedence relation* implies that the edge is either between start boundaries of two fragments \( f_k, I \in F_{seq} \) or between the start boundary of the fragment \( f_k \) and the message that occurs first in the \( f_k \).

(d) Similarly, for each *right null precedence relation* \( (I < \Lambda) \in P(M_{seq}) \), we draw an edge in control flow graph. Note that the \( (I < \Lambda) \) implies that \( I \) is the last *interaction* in a fragment \( f_k \in F_{seq} \) and that *interaction* \( I \) is either a message or a set of messages of the inner fragment \( f_i \) of the \( f_k \), where \( f_i, f_k \in F_{seq} \) and \( f_i \neq f_k \). Thus, we identify the \( M_{f_k} \) for a fragment \( f_k \in F_{seq} \) such that \( M_{f_k} \) contains \( I \). If \( I \in F_{seq} \), then we draw an edge from the fragment node (representing the end of the fragment \( I \)) to the fragment node (representing the end of the fragment \( f_k \)) otherwise, we draw an edge from the message node (corresponding to the message \( I \)) to the fragment node (corresponding to end of the fragment \( f_k \)). The edge thus obtained for a *right null precedence relation* implies that the edge is either between end boundaries of two fragments \( f_k, I \in F_{seq} \) or between the message that occurs last in the fragment \( f_k \) and the end boundary of the \( f_k \).

**Step 5** (identifying the labels of edges). Once the edges of the control flow graph (CFG) are determined, we assign the guard conditions associated with operands of each fragment of \( F_{seq} \) to the edges in CFG. For this, we consider the edge corresponding to each *left precedence relation* \( (\Lambda < I) \in P(M_{seq}) \) and label the edge same as the guard condition associated with the operand of the fragment that contains \( I \) (combined fragment or message). To obtain guard condition, we use value of the instance variable Guard of the
Table 3: Set of rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>$(\text{loop}^i \Rightarrow \text{m}_j &amp; \text{loop}^i \Rightarrow \text{m}_k) \Rightarrow (\text{loop}^i \Rightarrow \text{loop}^i &amp; \text{loop}^i \Rightarrow \text{m}_k &amp; \text{loop}^i \Rightarrow \text{m}_k)$.</td>
</tr>
<tr>
<td>$R_2$</td>
<td>$(\text{loop}^i \Rightarrow \text{c}_j &amp; \text{loop}^i \Rightarrow \text{m}_j) \Rightarrow (\text{loop}^i \Rightarrow \text{loop}^i &amp; \text{loop}^i \Rightarrow \text{m}_k &amp; \text{loop}^i \Rightarrow \text{m}_k)$.</td>
</tr>
<tr>
<td>$R_3$</td>
<td>$(\text{loop}^i \Rightarrow \text{m}_j &amp; \text{loop}^i \Rightarrow \text{f}_j^k) \Rightarrow (\text{loop}^i \Rightarrow \text{loop}^i &amp; \text{loop}^i \Rightarrow \text{f}_j^k &amp; \text{loop}^i \Rightarrow \text{f}_j^k)$.</td>
</tr>
<tr>
<td>$R_4$</td>
<td>$(\text{loop}^i \Rightarrow \text{m}_j &amp; \text{loop}^i \Rightarrow \text{f}_j^k) \Rightarrow (\text{loop}^i \Rightarrow \text{loop}^i &amp; \text{loop}^i \Rightarrow \text{f}_j^k &amp; \text{loop}^i \Rightarrow \text{f}_j^k)$.</td>
</tr>
<tr>
<td>$R_5$</td>
<td>$(\text{loop}^i \Rightarrow \text{c}_j &amp; \text{loop}^i \Rightarrow \text{f}_j^k) \Rightarrow (\text{loop}^i \Rightarrow \text{loop}^i &amp; \text{loop}^i \Rightarrow \text{f}_j^k &amp; \text{loop}^i \Rightarrow \text{f}_j^k)$.</td>
</tr>
<tr>
<td>$R_6$</td>
<td>$(\text{loop}^i \Rightarrow \text{c}_j &amp; \text{loop}^i \Rightarrow \text{f}_j^k) \Rightarrow (\text{loop}^i \Rightarrow \text{loop}^i &amp; \text{loop}^i \Rightarrow \text{f}_j^k &amp; \text{loop}^i \Rightarrow \text{f}_j^k)$.</td>
</tr>
</tbody>
</table>

$E\text{Operand}$ object whose $E\text{MessageList}$ contains the $E\text{Id}$ of the $E\text{ID}$. The interpretation of this label assignment is, if the guard condition associated with edge is satisfied then all messages in the operand of the fragment would be executed. All other edges would have no label.

Step 6 (applying control flow rules). Once preceding five steps are over, construction of graph representation for the interaction diagram is complete. Note that graph representation of interaction diagram captures control flow for the fragments $\text{loop}$, $\text{opt}$, $\text{break}$, $\text{ref}$, and so forth implicitly (see the discussions in Section 2.2). To reduce this graph representation into control flow graph, we propose a set of rules with the help of the notations given in Table 3.

(a) Loop Fragment. The first six rules ($R_1$–$R_6$) are for the $\text{loop}$ fragment. Rule $R_1$ says that if there is an edge labeled with $c$ from the fragment node representing the start of loop fragment $\text{loop}_i$ to message node $\text{m}_j$ and another edge from fragment node representing the end of the loop fragment $\text{loop}_i$ to message node $\text{m}_k$, then do the following: (i) add back edge from end of loop fragment $\text{loop}_i$ to the start of that loop fragment, (ii) add a loop exit edge from the start of loop fragment $\text{loop}_i$ to the message node $\text{m}_k$ with the label same as $\text{l}_c$, and (iii) delete the edge from end of the loop fragment $\text{loop}_i$ to the message $\text{m}_k$. Rules $R_2$ to $R_6$ are similar to the rule $R_1$, but only the difference is the contexts where rules are applied. For example, when loop fragment $\text{loop}_i$ has an inner fragment $f_j$ as the first interaction, then rule $R_2$ is applied. Rule $R_3$ is applied when $\text{loop}_i$ is followed by another fragment $f_k$. Rule $R_4$ is applied when $\text{loop}_i$ is contained in some other fragment $f_k$ as the last interaction. Rule $R_5$ is applied if $\text{loop}_i$ has an inner fragment $f_j$ as the first interaction and $\text{loop}_i$ is followed by the fragment $f_k$. Rule $R_6$ is applied when loop fragment $\text{loop}_i$ has an inner fragment $f_j$ as the first interaction, and $\text{loop}_i$ is contained in some other fragment $f_k$ as the last interaction.

(b) Opt Fragment. The two rules ($R_7$ and $R_8$) are for the $\text{opt}$ fragment. According to the rule $R_7$, if there is an edge labeled with $c$ from the fragment node representing the start of $\text{opt}$ fragment $\text{opt}_i$ to some message node $\text{m}_j$, then add an edge with the label same as $\text{l}_c$ from fragment node representing the start of $\text{opt}$ fragment $\text{opt}_i$ to the end of that $\text{opt}$ fragment. Rule $R_8$ is similar to the rule $R_7$, but the difference is that rule $R_8$ is applied only when the $\text{opt}$ fragment contains an inner fragment $f_j$ as the first interaction.

(c) Ref Fragment. Only one rule ($R_9$) is for the $\text{ref}$ fragment. Rule $R_9$ says that if $\text{ref}$ fragment refers to the interaction diagram ID whose graph representation is $\text{ID}(\text{S}_\text{ID}, \text{E}_\text{ID})$ with the start node $\text{S}_\text{ID}$ and end node $\text{E}_\text{ID}$, respectively, then do the following: (i) add an edge from the start of $\text{ref}$ fragment to $\text{S}_\text{ID}$, (ii) add an edge from $\text{E}_\text{ID}$ to the end of $\text{ref}$ fragment, and (iii)
delete the edge from the start of \textit{ref} fragment to the end of \textit{ref} fragment.

(d) \textbf{Break Fragment}. Next six rules (R10–R15) are for the \textit{break} fragment. Rules R10–R15 are similar to the rules R1–R6, but only the difference is that \textit{back edge} is not there in case of \textit{break} fragment, which is added for \textit{loop} fragment (see the rules from R1 to R6 and from R10 to R15).

\textbf{Illustration of Our Approach}. We illustrate our approach for conversion from XMI representation of an interaction diagram to an equivalent control flow graph with the help of a case study pertaining to a Restaurant Automation System (RAS). The RAS automates various functionalities of a restaurant such as Make Order, Process Order, and Generate Bill. Here, we focus only on a particular use case, namely, Generate Bill. In Generate Bill use case, manager of the restaurant inputs Order Number of an order whose Bill is to be generated. Depending on current status of the order (which may not even be processed or delivered) and whether Bill has already been generated for this Order or not, many scenarios can occur, which are modeled in interaction diagram as shown in Figure 6. All messages and fragments in the interaction diagram are referred to as Message Numbers and Fragment labels, respectively (see Figure 6).

\textbf{Identifying the Fragments and Their Message Sets}. Considering XMI representation of interaction diagram as input (see Figure 6), we first parse it using SAX parser and then obtain a set of lists of objects of following classes: (a) \textit{EMessage}, (b) \textit{EMessageEvent}, (c) \textit{ECallEvent}, (d) \textit{EOperation}, (e) \textit{EFragment}, (f) \textit{EOperand}, (g) \textit{EObject}, (h) \textit{EClass}, and (i) \textit{ELifeline} (see Table 2). The values of the instance variables of \textit{aMessage} object in the list of \textit{EMessage} objects are set with the values from the objects of other classes considering class relationships as discussed in Step 1. We then obtain the following.

(a) From the list of \textit{EMessage} objects, we find set of messages $M_{\text{seq}} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19\}$. On the other hand, the set of fragments $F_{\text{seq}}$ is obtained from the list of \textit{EFragment} objects as $F_{\text{seq}} = \{\text{loop}_1, \text{alt}_1, \text{alt}_2, \text{break}_1, \text{loop}_2, \text{break}_2\}$ (see Figure 6).

(b) We then identify set of messages for a fragment $f_i \in F_{\text{seq}}$ with considering the \textit{EFragment} object corresponding to $f_i$ as well as the set of \textit{EOperand} objects associated with the \textit{EFragment} object (see Figure 6).
We determine the nodes of control flow graph from \( M_{\text{seq}} \) and \( F_{\text{seq}} \) as follows.

(a) We add two fragment nodes into the set of nodes \( N \) of control flow graph for each fragment \( f_i \in F_{\text{seq}} \) and thus obtain \( N \) as \{\( \text{loop}_1^s, \text{loop}_1^p, \text{alt}_1^t, \text{alt}_2^t, \text{alt}_3^t, \text{alt}_4^t; \text{break}_1^t, \text{break}_2^t, \text{loop}_2^t, \text{break}_3^t, \text{break}_4^t \)\} (see Figure 7).

(b) Adding a message node \( m_i \) for each \( i \in M_{\text{seq}} \), the set of nodes \( N \) becomes \( N = N \cup \{m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9, m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, m_{15}, m_{16}, m_{17}, m_{18}, m_{19}\} \). Here, the filled nodes represent fragment nodes, and empty nodes represent message nodes (see control flow graph in Figure 7).

(c) We store the values of tuple \( \langle T, \text{ID}, B, \text{RO}, M, \text{PR}, \text{RVar} \rangle \) for all nodes in \( N \) of CFG \( \langle N, E \rangle \) in a table named as Node table (see Table 4). For message node \( m_i \in N \), we use the values of instance variables of EMessage object corresponding to the message \( i \in M_{\text{seq}} \). We obtain the information for the fragment nodes from the EFragment objects (see Table 4). Note that for message node corresponding to a reply message, PR would be same as the return value if any, and both \( M \) and \( \text{RVar} \) would be empty.

Determining Fragment Structure. In this step, we first find the outermost fragments formed by the set of messages \( M_{\text{seq}} \). 

![Figure 7: Graph representation and control flow graph of interaction diagram for Generate Bill use case.](image-url)
We then determine the inner fragments contained in each fragment to find the hierarchy structure of the fragments.

(a) We need to determine the minimum number of fragments $F'$ that together correspond to the largest subset of $M_{seq}$. The $F'$ is initialized as the set of all fragments of the interaction diagram, that is, $F' = F_{seq} = \{loop_1, alt_1, alt_2, break_1, loop_2, break_2\}$. We observe that $M_{break_2} \subset M_{alt_2}$, $M_{loop_2} \subset M_{alt_2}$, and $M_{break_2} \subset M_{loop_2}$. As each message set of three fragments $break_1$, $loop_2$, and $break_2$ is subset of message set of $alt_2$, we exclude these three fragments from $F'$, which reduces $F' = \{loop_1, alt_1, alt_2\}$. That implies that fragments $loop_1$, $alt_1$, and $alt_2$ together are sufficient to replace the largest subset $\{3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18\}$ of $M_{seq}$ and are the outermost fragments formed by the messages in $M_{seq}$. Replacing the largest subset of $M_{seq}$ by the fragments in $F'$, we obtain $M_{seq} = \{1, 2, loop_1, alt_1, alt_2, 19\}$.

(b) To determine the hierarchy structure of the fragments in $F_{seq}$, let us first consider the fragment $loop_2$. Only one fragment $break_2$ is sufficient to replace the largest subset $\{10\}$ of $M_{loop_2} = \{9, 10\}$. After replacing the largest subset of $M_{loop_2}$ by $break_2$, we obtain $M_{loop_2} = \{9, break_2\}$. Therefore, $loop_2$ contains only one inner fragment $break_2$. Now consider another fragment $alt_2$. In this case, minimal set of fragments $F'' = \{break_1, loop_2\}$ is sufficient enough to replace together the largest subset $\{7, 9, 10\}$ of $M_{alt_2} = \{6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18\}$. Therefore, after replacing corresponding subset of $M_{alt_2}$ by each fragment in $F''$, we obtain $M_{alt_2} = \{6, break_1, 8,
loop2, 11, 12, 13, 14, 15, 16, 17, 18). It implies that fragment alt2 contains inner fragments break1 and loop2. Note that Mloop1, Malts1, Mbreak1, and Mbreak2 remain unchanged as there is no fragment in Fseq that can replace the subset of message sets of fragments loop1, alt1, break1, and break2.

Determining the Edges. In this step, we first apply precedence relations on Mseq. Depending on the operand of the precedence relation, we determine the edges. (a) Applying < on Mseq, we obtain P(Mseq) = \{1 < 2, 2 < loop1, loop1 < alt1, alt1 < alt2, alt2 < 19\} ∪ P(Mloop1) ∪ P(Malt1) ∪ P(Malt2). These unions are for the fragment operands loop1, alt1, and alt2 present in the precedence relations of Mseq. To compute the unions, we need to have the sets of precedence relations for each fragment operand: loop1, alt1, and alt2. They are computed as below:

(i) P(Mloop1) = \{Λ < 3, 3 < Λ\}.
(ii) P(Malt1) = \{Λ < 4, 4 < Λ, Λ < 5, 5 < Λ\}.
(iii) P(Malt2) = \{Λ < 6, 6 < break1, break1 < 8, 8 < loop2, loop2 < 11, 11 < 12, 12 < 13, 13 < 14, 14 < 15, 15 < 16, 16 < 17, 17 < Λ, Λ < 18, 18 < Λ\} ∪ P(Mbreak1).
(iv) P(Mbreak1) = \{Λ < 7, 7 < Λ\}.
(v) P(Mloop2) = \{Λ < 9, 9 < break2, break2 < Λ\} ∪ P(Mbreak2).
(vi) P(Mbreak2) = \{Λ < 10, 10 < Λ\}.

Therefore, P(Mseq) = \{1 < 2, 2 < loop1, loop1 < alt1, alt1 < alt2, alt2 < 19, Λ < 3, 3 < Λ, Λ < 4, 4 < Λ, Λ < 5, 5 < Λ, Λ < 6, 6 < break1, break1 < 8, 8 < loop2, loop2 < 11, 11 < 12, 12 < 13, 13 < 14, 14 < 15, 15 < 16, 16 < 17, 17 < Λ, Λ < 18, 18 < Λ, Λ < 7, 7 < Λ, Λ < 9, 9 < break2, break2 < Λ, Λ < 10, 10 < Λ\}.

(b) Depending on the operand of a precedence relation (Ix < Iy) in P(Mseq), we draw an edge in control flow graph as follows:

(i) In our example, we have seven precedence relations 1 < 2, 11 < 12, 12 < 13, 13 < 14, 14 < 15, 15 < 16, and 16 < 17 satisfying the condition (Ix, Iy ∈ Mseq) and thus, we draw seven edges (m1, m2), (m11, m12), (m12, m13), (m13, m14), (m14, m15), (m15, m16), and (m16, m17) in control flow graph.
(ii) The condition (Ix ∈ Mseq, Iy ∈ Fseq) is satisfied by the precedence relations: 2 < loop1, 6 < break1, 8 < loop2, and 9 < break2, and they correspond to four edges (m2, loop1), (m6, break1), (m8, loop2), and (m9, break2).
(iii) There are three precedence relations alt2 < 19, break1 < 8, and loop2 < 11 satisfying the condition (Ix ∈ Fseq, Iy ∈ Mseq). Therefore, we draw three edges (alt1, m19), (break1, m8), and (loop1, m11).

(iv) The two precedence relations: loop1 < alt1 and alt1 < alt2 satisfy the condition (Ix, Iy ∈ Fseq). Considering these two precedence relations, we draw two edges (loop1, alt1) and (alt1, alt2).

(c) In our example, P(Mseq) (as obtained in step (a)) has the following left null precedence relations: Λ < 3, Λ < 4, Λ < 5, Λ < 6, Λ < 7, Λ < 9, Λ < 10, and Λ < 18. Here, the fragment loop1 contains 3; alt1 contains 4, 5; alt2 contains 6, 18; break1 contains 7; loop2 contains 9; break2 contains 10 (see the message sets Mloop1, Malts1, Mbreak1, Mloop2, and Mbreak2). For these eight left precedence relations, we draw the following edges (loop1, alt1), (alt1, alt2), (alt2, m18), (break1, m7), (loop2, m9), and (break2, m10).

(d) There are eight right null precedence relations: 3 < Λ, 4 < Λ, 5 < Λ, 7 < Λ, 8 < Λ, 10 < Λ, 12 < Λ, and 13 < Λ. Among them, only one break2 < Λ has fragment operand and the fragment loop2 contains break2. (see the Mloop1 and Mbreak2 after replacement in the step [determine fragment structure]). For this right null precedence relation, we draw an edge (break2, loop2). For remaining seven right null precedence relations, we draw seven edges (m3, loop1), (m4, alt1), (m5, alt2), (m17, alt2), (m18, alt2), (m2, break1), and (m10, break2).

Identifying the Labels of Edges. We label the edges in the control flow graph corresponding to the left null precedence relations. Let us now consider a left null precedence relation Λ < 7. Here, 7 ∈ Mbreak, this means that fragment break1 contains the message 7. The only one operand of the break1 that contains the message 7 is associated with the condition Status ≠ “Delivered”. Therefore, the edge (break1, m7) corresponding to (Λ < 7) is labeled with the constraint: Status ≠ “Delivered”. Similarly, the edges corresponding to other left null precedence relations are labeled as shown in Figure 7. At the end of this step, the control flow graph for Generate Bill interaction diagram is the same as the graph representation as shown in Figure 7(a).

Applying Control Flow Rules. In our example interaction diagram, we have two loop fragments and two break fragments. So, we need to apply the control flow rules for them. We observe that first loop fragment (loop1) does not contain an inner fragment and is followed by another fragment alt1, and thus we apply the rule R3. Applying the rule R3, we add two edges (loop1, alt1) and (loop1, alt2) and delete the edge (loop1, alt2). For the fragment break1, we apply the rule R10 as break1 does not contain an inner fragment and is not followed by another fragment. Applying the rule, we delete the edge (break1, m8) and add the edge (break1, m8). Applying rule R13 for the fragment break2, we delete the edge (break2, loop2) and add the edge (break2, loop2). Here, the loop fragment loop2 does not contain a fragment as first interaction and is not followed by a fragment. So, we apply the rule R1 and add edges (loop2, alt2).
5. XMI2CFG: A Prototype Tool

We have developed a prototype tool named as XMI2CFG (XMI of interaction diagram to control flow graph) following our approach. We have implemented XMI2CFG in Java language (Java 2) using NetBeans IDE 6.1 [23]. Input of XMI2CFG is the XMI representation of UML 2.x interaction diagram. We have used MagicDraw 16.0 [21] to draw interaction diagram and subsequently exported this diagram in the form of XMI representation. XMI2CFG visualizes the control flow graph as the output. XMI2CFG consists of two main components: CFGConstructionUnit and VisualizationUnit. CFGConstructionUnit first parses the XMI of interaction diagram and then converts it into control flow graph. Taking control flow graph as the input, VisualizationUnit changes it into the DOT language format [24] and produces an image to visualize the control flow graph. Two components: CFGConstructionUnit and VisualizationUnit are described below.

CFGConstructionUnit. This component parses the XMI representation of interaction diagram using SAX parser. The class diagram of this component is shown in Figure 8. This component comprises of two main classes: MyParser, and CFGConstructor, and other auxiliary classes: EMessage, ELifeline, EClass, ECallEvent, EMessageEvent, EFragment, EOoperand, EOperation, EFragment, Node, and Edge. Dependency among these classes is also depicted in Figure 8.

The MyParser class implements the event-handlers startElement(), endElement(), characters(), and endDocument() to interface with SAX parser. For this, we have used the library of Apache Xerces available in the web portal [22]. In the event-handler startElement(), we process the tagged elements starting with the names same as “ownedAttribute”, “lifeline”, “fragment”, “operand”, “guard”, “specification”, “argument”, “body”, “ownedParameter”, “message” and “packagedElement”, “ownedOperation”, “ownedBehavior”, “guard”, and “ownedParameter”. Depending on type of tagged elements, we categorize them as “MessageEvent”, “Fragment”, “CallEvent”, “Object”, “Class”, “Lifeline”, “Operand”, “Message”, “Operation”, “Parameter”, “SequenceDiagram”, and “Guard”. When multiple tagged elements start with the same name, then we consider the value of the attribute “XMI type”. For example, tagged elements specifying the class name and call event start with the same name “packagedElement”. In order to distinguish them, we have considered whether the attribute “XMI type” has the value as “uml:CallEvent” or “uml:Class”. For each processed tagged element, we retrieve the associated metainformation of interaction diagram from SAX parser and store them by means of instance variables: Id, Name, ClassId, ObjectId, FragmentType, Guard, SendEventId, ReceiveEventId, CallEventId, MessageType, OperationId, MessageId, and LifelineId of the class named MyParser. Note that after processing of a tagged element, only relevant variables would have meaningful values, and the rest would have the null value.

Once metainformation of interaction diagram is available through instance variables of MyParser object, we instantiate metaobject of particular type as mentioned in metaclass diagram of interaction diagram (see Figure 5). We then pass the meta-object to CFGConstructor object via parameter of its method RegisterSAXEvent(). After all necessary metaobjects are passed to CFGConstructor, then CFGConstructor object would have a set of arraylists of objects: EMessageList, EFragmentList, EOoperandList, EOperationList, ECallEventList, EMessageEventList, ELifelineList, EObjectList, and EClassList. These arraylists store meta object of types EMessage, EFragment, EOoperand, EOperation, ECallEvent, EMessageEvent, ELifeline, EObject, and EClass, respectively. To keep the track of the hierarchy structure of fragments as well as operands of a fragment, we use two arraylists: FragmentIDList and OperandIDList. Note that FragmentIDList and OperandIDList keep the IDs of fragments and operands whose processing is still remaining. In order to update these two arraylists, we add the ID into corresponding arraylist within startElement() when SAX parser notifies about the opening of fragment/operand tag. Similarly, we remove the ID of fragment/operand from the corresponding list within endElement(), when SAX parser notifies about the closing of the fragment/operand. We also use different flags such as OperationParameterFlag, and GuardFlag to keep tracking the end of processing of parameters of recent operation and guard associated with fragment operand.

The conversion logic is encapsulated in different methods such as ComputeMessageSet(), ComputeFragmentSet(), DetermineFragmentStructure(), DetermineCFGNodes(), ComputePrecedenceRelations(), DetermineCFGEdges(), DetermineEdgeLabel(), and ApplyControlFlowRules() of CFGConstructor class. Two methods ComputeMessageSet() and ComputeFragmentSet() perform the first step of our conversion approach, that is, computing the message set and fragment set. Two arraylists EMessageList and EFragmentList are used for this purpose. DetermineCFGNodes() performs the second step of conversion. Two Node objects for each fragment (specifying the start and end of the fragment) and one Node object for each message are instantiated. Node information (i.e., the values of instance variables of Node object) is obtained from arraylists EMessageList and EFragmentList. Once a Node object is instantiated, it is added into arraylist named CFGNodeList. As per our conversion approach, DetermineFragmentStructure() first identifies the minimum number of outermost fragments that can replace largest subset of message set of interaction diagram and replaces that subset by IDs of outermost fragments. After that, for message set of each fragment, a set of inner fragment IDs is identified, and the subset of message set of the fragment is replaced by IDs of inner fragments (see details in the third step of our conversion procedure). ComputePrecedenceRelations() computes the precedence
relations using SeqNo. of interaction (message/fragment) in message set of interaction diagram. If a fragment operand is found in some precedence relation, then precedence relations among messages of the fragment also need to be computed. Based on type of operand in precedence relation, DetermineCFGEdges() determines the edges between message and message, message and fragment start, fragment end and message, and fragment end and fragment start. In addition to this, DetermineCFGEdges() also finds the edges corresponding to null precedence relations. These edges are between fragment start and message, fragment start and fragment start, message and fragment end, and fragment end and fragment end. For each edge, we identify a pair of Node objects which correspond to the end nodes of the

---

**Figure 8:** The class diagram of CFGConstructionUnit.
edge, and then instantiate an Edge object using references of the associated Node objects. Once an Edge object is instantiated, it is added into arraylist named CFGEdgeList. To construct the edge label, DetermineEdgeLabel() first identifies Edge object corresponding to that edge and hence, EOperand object corresponding to an operand which contains the fragment/message referenced by the second operand of the precedence relation for that edge. After that, the Edge label is set as same as the instance variable Guard of the corresponding EOperand object. The ApplyControlFlowRules() adds and deletes the edges depending on control flow rules for the fragments: opt, break, loop, and so forth. Final control flow graph is captured in two arraylists: CFGNodeList and CFGEdgeList.

Visualization Unit. The main task of this component is to visualize the control flow graph in an image form. VisualizationUnit comprises of two main classes: DotConverter and GraphViz. We have used the code of GraphViz available in [25]. Taking two arraylists CFGNodeList and CFGEdgeList as the input, the DotConverter object creates a .dot file by using node and edge specification as per DOT language [24]. To specify them in DOT language, we have used different methods such as start_digraph(), addln(), and end_graph() of GraphViz class. Once .dot file is created, an image is produced by calling the methods getDotSource(), getGraph(), and writeGraphToFile() of GraphViz.

XM12CFG supports different menu options for selecting an XMI file, displaying XMI file, parsing selected XMI file, starting conversion, and display control flow graph in both DOT language and image form. One typical usage scenario of XM12CFG is depicted in Figure 9. In this usage scenario, we first select and display the XMI file in upper left panel. We then select the option to start parsing and subsequently construct the control flow graph. For this, we invoke CFGConstructionUnit component. Once control flow graph construction is complete, we visualize it in both DOT language format and image form in lower left and right panel, respectively.

6. Comparison with Related Work

Control flow analysis has been investigated widely in the context of program analysis and compiler design [26].
However, control flow analysis in the context of XMI representation of UML interaction diagram (it is referred as sequence diagram in earlier version of UML 2.x) has scarcely been reported [2, 4, 5, 11, 17, 18]. Garousi et al. [17, 18] propose a methodology to analyze control flow of UML 2.x sequence diagrams using formal annotations in object constraint language (OCL). For control flow analysis, they have considered concurrent systems, which subsumes nonconcurrent systems. For activity diagram having concurrency constructs (fork-join), they derive an extended activity diagram metamodel, called as concurrent control flow graph (CCFG) based on UML 2.x activity diagrams. Garousi et al. [17, 18] propose formal OCL-based mapping rules to map an instance of sequence diagram into an instance of CCFG metamodel. A formal representation of concurrency control flow path is also discussed in their work. The differences between Garousi et al.’s approach [17, 18] and our approach is as follows. First, Garousi et al.’s approach formalizes the control flow analysis using metamodel information and object constraint language (OCL) annotations. Our approach on the other hand, achieves conversion by processing XMI representation of sequence diagram. For this, we have considered the inherent structure of XMI representation of sequence diagram as well as metamodel of sequence diagram as specified by OMG. Second, their approach considers the mapping rules for construction of edges in control flow graph, whereas our approach uses message sets of fragments and precedence relations among interactions. Third, their control flow analysis considers both the synchronous and asynchronous messages but our approach considers synchronous messages only as most method calls in object-oriented programming languages are synchronous. Note that asynchronous call generally occurs in multithreaded applications, which is not considered in this work. Fourth, our approach is directly applicable in an industry scenario where XMI representation of UML model is exported from UML modeling tools for further processing, which is the standard practice to process UML models. Sarma et al. [5] use control flow graph for test case synthesis from UML sequence diagrams. First, all operation scenarios are identified from UML sequence diagram, where an operation scenario represents a sequence of messages. The operation scenarios are used to form a directed graph called sequence diagram graph (SDG), where a node represents a message and an edge represents control flow. Sarma et al. augment the nodes of SDG with different information like attributes of participating objects, parameters of the method, predicate (if any), range of attribute values, and so forth necessary to compose test vectors. This information is obtained from use case templates, class diagrams, and data dictionary. With the node information, SDG is used to generate test cases. The differences between Sarma’s work and our work is as follows. First, Sarma’s work uses UML 1.x sequence diagram that does not support fragments whereas our work considers fragments supported by UML 2.x sequence diagram. In fact, handling sequence diagram containing nested fragments for conversion is a nontrivial problem, which our work has addressed. Second, Sarma’s work has not reported how to obtain SDG from XMI representation of sequence diagram but our approach includes a detailed conversion process from XMI of UML 2.x sequence diagram into control flow graph.

Nayak and Samanta [4] propose an approach to synthesize test cases from UML 2.0 interaction diagrams. They convert interaction diagrams into a control flow graph called scenario graph encapsulating control flow and fragment structure. The scenario graph consists of different types of nodes: block node, decision node, merge node, fork node, and join node. Note that block node represents a set of messages that execute consecutively. Subsequently, scenario graph is processed to build an intermediate testable model called ITM. To build an ITM, minimum regions (a region has single entry and single exit node and is not contained in other region) are identified in scenario graph and then replaced by composite nodes successively until scenario graph becomes a single chain of nodes, called ITM. The ITM is used to generate test scenarios by expanding each composite node into its internal paths in successive iterations. The test scenarios are further processed to synthesize test cases. But Nayak et al. have not reported how to apply proposed mapping rules to XMI representation of an interaction diagram for obtaining ITM, which is indeed a nontrivial problem.

Dinh-Trong et al. [11] propose an approach to generate inputs to test UML design models. Dinh-Trong et al. consider the information encapsulated in class and sequence models and combine them to build a graph structure called variable assignment graph (VAG). A VAG is a control flow graph annotated with data flow information. The graph contains two types of nodes: message node representing a message of sequence diagram and control node showing alternate execution paths of messages from the combined fragments. Edges between two nodes in VAG represent the control flow. Dinh-Trong et al. select a path from the VAG based on some structural coverage criteria like node coverage, edge coverage, path coverage, and generate test inputs by solving the constraints along the selected path. Extending this work, Bandyopadhyay and Ghosh [2] augment the VAG with information derived from the state machines of the participating objects to build a testable model called the extended variable assignment graph (EVAG). Once EVAG is built, they solve the path constraints to generate test inputs for each EVAG path. But they [2, 11] have not reported how mapping rules can be applied to XMI representation of a sequence diagram to obtain a VAG like Sarma et al. [5] and Nayak and Samanta [4].

7. Conclusions

Control flow graph is an important data structure used in many software engineering processes. This paper proposes an approach how a control flow graph can be obtained from XMI representation of UML 2.x interaction diagram. The proposed approach overcomes the difficulties of attributing all information from XMI tagged elements and then obtains its equivalent graph representation. From the reported work, it is not understood how actually a UML interaction diagram
is processed, if automatically. We give a way out to this hindrance. The prototype tool XMI2CFG based on the proposed approach has been tested with a large number of interaction diagrams in real-life applications. A thorough investigation on these applications substantiates the correctness of the proposed approach. Indeed the proposed approach bridges the gap between theory and practice of converting (XMI representation of) UML 2.x interaction diagram to control flow graph. To add more, the methodology can be extended to other UML diagrams such as activity diagram and interaction overview diagram with a minor enhancement in each case.

References


