International Journal of Recent Trends in Engineering, Vol 1, No. 1, May 2009

Document Type Definition for the XMI Representation of UML2.0 Activity Diagram

Philip Samuel[1], Sunitha E.V[2]
[1]School of Engineering, [2] Department of Computer Science
Cochin University of Science And Technology, India – 682022
[1]philips@cusat.ac.in, [2]sunithaev@gmail.com

***Abstract* -- This paper describes a new Document Type Definition (DTD) to represent UML activity diagram in XMI (XML Metadata Interchange) format. DTDs are important as far as automatic code generation is concerned. Our proposed DTD considers the activity diagram as a graph. The elements in the activity diagram are nodes and edges. The old versions (1.x) of the UML DTDs describe the activity diagram as a special type of state machine. The proposed DTD is based on UML 2.0, where the activity diagram semantics is rooted in Petri Nets rather than state machines. The proposed DTD defines the tags for different types (action, decision, initial, final etc.) of nodes in the activity diagram and the attributes required for the edges. Each node and edge in the activity diagram can be mapped to the respective XMI tags using this DTD. This can be done in sequential manner. The paper also describes how this DTD is used in the conversion of activity diagram to XMI format, and an algorithm for the conversion process.**

***Index Terms* -- UML activity diagram, DTD, XMI, XML, automatic code generation**

I. INTRODUCTION

UML activity diagrams are very important to present the business process and workflow [1]. A programmer can implement it by writing suitable code in a programming language. Anyhow, to draw an activity diagram and to get its full implementation code on a button click is really amazing. This sounds like an easy task, but, indeed, it needs lots of work underneath.

Converting activity diagram to platform neutral representation is a very important step in the automatic code generation. This representation should be widely accepted. The most widely accepted platform independent representation is XMI (XML Metadata Interchange) [5]. Platform independent representation provides portability which allows activity diagram to be exchanged between different modeling tools [3]. This gives developers the freedom to use the modeling tools convenient for their work. This flexibility can be achieved when the diagrams are converted to XMI which is a platform independent format.

XMI is an application of XML (eXtensible Markup Language), which is an HTML. We can define XML according to the UML notations [2]. These definitions can be written as Document Type Definition (DTD) file. Each element in the activity diagram should be defined in this DTD. Thereby the activity diagram can be completely converted to the XMI format without any ambiguity

This paper presents a DTD to represent UML 2.0 activity diagram. The XML document will be validated against the DTD. When the modeling tools generate the XMI representation of the UML diagrams, they should use the agreed DTD. If heterogeneous modeling tools need to be interoperable then they should use the same DTD. UML 2.0 has changed the semantics of activity diagram from state machine to graph/Petri Nets [10]. The proposed DTD is following UML 2.0 and so, it can replace the old DTDs. Earlier activity diagrams were represented as a special case of statechart diagrams [4] and hence their representation is complex. Sometimes it may be related to some other objects. These complexities are solved in the proposed DTD.

This paper also presents an algorithm to convert the activity diagram to the XMI format, which make use of the proposed DTD. The algorithm gives a straight forward method for the conversion of the activity diagram to XMI format.

The remainder of this paper is organized as follows: section II discusses the related works; section III describes the proposed DTD; section IV describes our method of converting the activity diagram to XMI based on the new DTD; section V gives the evaluation of the proposed DTD; and section VI concludes the paper.

II. RELATED WORKS

The Java implementation code generation from UML diagrams is described in [6]. In this method they accept high-level UML diagrams and convert them to Java code. This method places fewer constraints on the UML construct, like generalization and association classes. In this paper the authors describe the objectives in mapping the high level UML designs, like separation of design from implementation, separation of behavior from representation etc. They are also addressing some special issues related to Java code generation from class diagrams, like how to map multiple inheritance to Java code.

Bjoraa et. al, [7] describes the generation of Java skeleton from the XMI format. This paper considers class diagrams for converting into XMI. Also, they are generating Java skeleton from the diagram. They are using template based approach for the code generation. There will be different methods to compose different elements in the skeleton. The order, in which these methods are invoked, is determined by the template [7]. The XMI file is parsed using a DOM parser and the relevant nodes are extracted and placed in lists. Vectors and hashmaps are used for lists.

III. PROPOSED DOCUMENT TYPE DEFINITION (DTD)

Document Type Definition (DTD) is used to define markup languages (ie, tagged languages like HTML). It is a schema giving the structure of specific XML markup language instances. An XML document may be validated against a particular DTD to ensure that it conforms to it. Typically the tags defined in the DTD would be particular to the domain addressed by the document and parties within that domain would agree on a common DTD. The interoperability between heterogeneous tools is achieved by the use of common UML DTD in these tools. If the tools are using same DTD they can easily exchange UML models. The DTD that describes the UML 2.0 Activity Diagram is shown in Fig. 1. The activity diagram is referred as 'activity graph' to show the semantic changes of activity diagram in UML 2.0. An activity graph is described as a set of nodes and edges.

AG = { N, E }

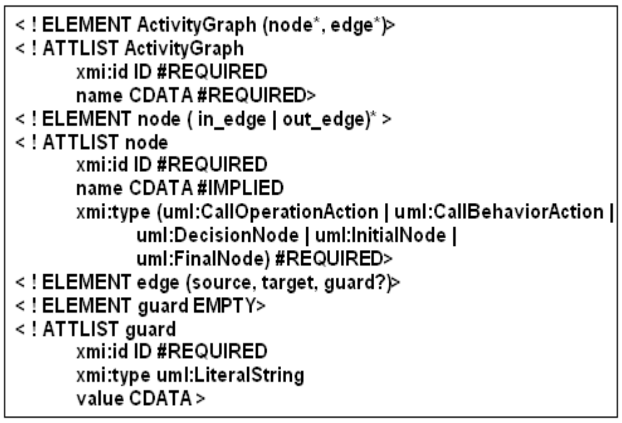
where,

AG - Activity Graph, N- Nodes, and E- edges

The activity graph is composed of two main components; nodes and edges. Nodes are connected through edges. The activity graph contains two special types of nodes called initial node and final node, which represent the start and stop end of the graph respectively. Each node in the activity graph should be an element in the path between the initial and final node. Each node should have at least one incoming and one outgoing edges, except for the initial and final nodes.

The edges show the data/control flow. A node can be any of the four types; ActionNode, DecisionNode, InitialNode and FinalNode. ActionNode represents a node which is doing some action. UML 2.0 is specifying around 30 different type of actions, like CallOperationAction, CallBehaviorAction etc. [10]. DecisionNode represents the decision making point in the graph. InitialNode and FinalNode represent the start and stop nodes of the activity graph.

In our DTD we are defining different tags to represent different components of activity diagram. We define the tag <*ActivityGraph*> as the root element of the XMI representation of the activity diagram. It has two components; <*node*> and <*edge*>. The tag <*node*> has two sub elements; <*in_edge*> and <*out_edge*>. <*in_edge*> has an attribute xmi:idref, which keeps a reference to the incoming edge of the corresponding node. <*out_edge*> has the attribute xmi:idref, which stores reference to the outgoing edge from the respective node. Apart from these sub elements, each <*node*> element has its own attributes, like xmi:id, xmi:type and name. The xmi:id is the unique identifier for each <*node*>



```
< ! ELEMENT ActivityGraph (node*, edge*)>
< ! ATTLIST ActivityGraph
        xmi:id ID #REQUIRED
        name CDATA #REQUIRED>
< ! ELEMENT node ( in_edge | out_edge)* >
< ! ATTLIST node
        xmi:id ID #REQUIRED
        name CDATA #IMPLIED
        xmi:type (uml:CallOperationAction | uml:CallBehaviorAction |
                uml:DecisionNode | uml:InitialNode |
                uml:FinalNode) #REQUIRED>
< ! ELEMENT edge (source, target, guard?)>
< ! ELEMENT guard EMPTY>
< ! ATTLIST guard
        xmi:id ID #REQUIRED
        xmi:type uml:LiteralString
        value CDATA >
```

Fig. 1 Part of DTD for Activity Diagram

element in the activity diagram. The 'name' is the meaningful name that the designer has given for the element <*node*>. The xmi:type is the attribute which helps us to distinguish different nodes in the diagram. It can be an ActionNode, DecisionNode, InitialNode or FinalNode. Just for simplicity we consider these four actions first. The proposed DTD is shown in Fig. 1.

The element <*edge*> has three sub elements; <*source*>, <*target*> and <*guard*>. The <*source*> and <*target*> elements represent the respective source and target <*node*> elements. The element <*edge*> has attribute xmi:id, which uniquely identifies the edge in an activity diagram. The structure of the basic tags is shown in Fig. 2. Activity diagram semantics are based on graphs (mainly Petri Nets) in UML 2.0 [8]. In our DTD, we tried to follow this graph semantics. We consider activity diagram as a separate diagram which has its own attributes and properties, rather than considering it as a type of state machine. This helps us to directly map each element in activity diagram to the tagged representation in XMI. Hence, the semantic change reduces the complexity of the activity diagram representation.

Consider a workflow that gets name from the user and then prints it. The activity diagram for the same is shown in Fig. 3. The workflow contains two main actions; *getName* and *printName*. *getName* get the name from the user and *printName* prints the name. These two actions are the high level representations. This can again be divided into sub actions. In this example we are not going into such details.

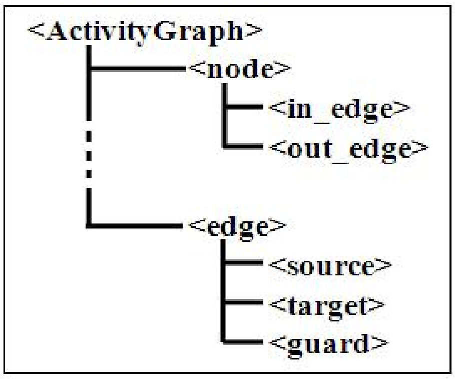
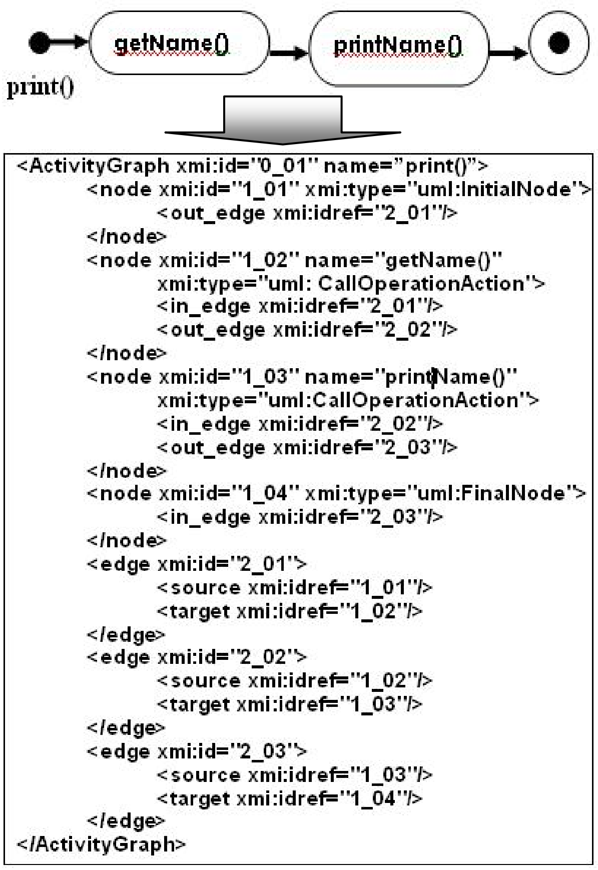


Fig. 2 Tag structure in the proposed DTD



```
<ActivityGraph xmi:id="0_01" name="print()">
        <node xmi:id="1_01" xmi:type="uml:InitialNode">
                <out_edge xmi:idref="2_01"/>
        </node>
        <node xmi:id="1_02" name="getName()"
                xmi:type="uml: CallOperationAction">
                <in_edge xmi:idref="2_01"/>
                <out_edge xmi:idref="2_02"/>
        </node>
        <node xmi:id="1_03" name="printName()"
                xmi:type="uml:CallOperationAction">
                <in_edge xmi:idref="2_02"/>
                <out_edge xmi:idref="2_03"/>
        </node>
        <node xmi:id="1_04" xmi:type="uml:FinalNode">
                <in_edge xmi:idref="2_03"/>
        </node>
        <edge xmi:id="2_01">
                <source xmi:idref="1_01"/>
                <target xmi:idref="1_02"/>
        </edge>
        <edge xmi:id="2_02">
                <source xmi:idref="1_02"/>
                <target xmi:idref="1_03"/>
        </edge>
        <edge xmi:id="2_03">
                <source xmi:idref="1_03"/>
                <target xmi:idref="1_04"/>
        </edge>
</ActivityGraph>
```

Fig. 3 Example for XMI representation using new DTD

We gave the name *print()* for the activity diagram. Its corresponding XMI representation is shown in Fig. 3. Only the content part of the XMI document is shown here. In the example, we can see that each type of element has a separate sequence of ids.

IV. XMI Representation Based On The Proposed DTD

The XMI document will contain two main parts; header and content. Header specifies the exporter, the XMI version we are using etc. The content part contains the actual activity diagram, i.e., XMI.content part in the XMI document.

In this section we propose an algorithm to generate the XMI representation of the UML2.0 activity diagram based on the proposed DTD. This algorithm uses the traditional Graph traversal algorithms to explore the nodes in the activity graph. First, makes the list of the nodes and edges by traversing the activity graph. Assign a unique id for all elements. Record all the attributes of the elements. For a node, its type and name should be noted. For an edge, its source and destination nodes should be known. These details will be represented as the attributes of the node and edge elements. After completing the list of nodes and edges, it can be mapped to the XMI format by referring the DTD.

The first step in our method is to explore all nodes in the activity graph. Then convert them to their respective XMI tags. The entire conversion process can be explained as follows. First, do the Breadth First Traversal or Depth First Traversal to explore all the nodes and edges in the activity diagram. In the example, we have used Depth First Traversal. The start vertex is the initial node. Each node/edge should be assigned with a unique id. In the proposed method, explore each node of the graph and add to a list of the nodes, say *Nodes*, including their attributes.

Each flow/edge is added to a list of edges, say *Edges*, including their source, target and guard (if any). After the traversal, we will get all the nodes and edges in the lists *Node* and *Edges* respectively. Algorithm to convert the activity diagram to XMI is given below.

For each activity diagram,

1. Do Breadth First Traversal/ Depth First Traversal by keeping initial node as the start vertex
    - ▪ Each node is added to the list *Nodes* along with their id, type and other attributes
    - Each flow/ edge is added to the list *Edges* along with their id, and guard condition (if any)
    - The source and target can be identified with the id of the respective nodes
2. Map each element in the *Nodes* list to its corresponding XMI tag.
3. Map each element in the *Edges* list to its corresponding XMI tag

Consider the activity diagram in Fig. 3. After the first step we will get two tables which contain the list of nodes and the list of edges as shown in Table I. The table *Node* contains the nodes. For each node we have to record the ID, name, type, in-edge and out-edge. ID is the unique id given to each element in the activity graph. For different types of elements (node, edge, entire graph), we give a different series of IDs. For activity Diagram, we can give the id as 0_01. The prefix number (the number preceding the underscore) can be used to distinguish the element type. We are using the prefix number 0 for activity diagram, 1 for nodes, and 2 for edges. Name of the nodes represents the name given by the user for each action. It is an optional one, because for the nodes like decision nodes name is not necessary.

TABLE I
Tables Created After Graph Traversal

Node				
Xmi :ID	**Name**	**Xmi:Type**	**In_edge**	**Out_edge**
1_01		Uml:InitialNode		2_01
1_02	getName ()	Uml:CallOperationAction	2_01	2_02
1_03	printName()	Uml:CallOperationAction	2_02	2_03
1_04		Uml:FinalNode	2_03	

Edge			
Xmi:ID	**Source**	**Target**	**Guard**
2_01	1_01	1_02	
2_02	1_02	1_03	
2_03	1_03	1_04	

Type of the nodes is the action types defined by UML [9]. As the second step we can map the element of the above table to their tagged representations. We can use a template for this conversion. It is shown below.

```
<node  xmi:id= _____ xmi:type= ____  name= ____ >
        <in_edge  xmi:idref= ______  />
        <out_edge  xmi:idref= _____ / >
</node>
<edge   xmi:id= _____   >
        <source xmi:idref = ______  />
        <target xmi:idref = _______  />
        <guard xmi:id =___ xmi:type=__value= ____/>
</edge>
```

The text in the template will be printed as such and the blank lines represent the changing data in the XMI representation. These blank spaces should be filled when a node or edge is encountered. The template shows that, for each node the variable values are the id, type and name of the node. Moreover, the child element of the node is referring to the incoming and outgoing edges. For each node we can use the above template with the corresponding values of the variable terms. Similarly, the second part of the template shows edges. The final representation of the activity diagram is shown in the Fig. 3.

New generation UML tools, like UModel, MagicDraw etc., follow UML2.0 by defining their own XMI DTDs. These DTDs are not yet standardized.

V. Evaluation Of The Proposed DTD

The UML DTD presented in this paper is using the semantics of graph to represent the activity diagram. It makes the UML activity diagram representation simple. Moreover, it follows the UML 2.0 specification. Hence, this will be useful in new generation UML tools.

A comparison with the old DTD will help us to understand the advantages of the proposed one. As per UML 1.x, activity diagram is following the semantics of state machines [4]. Each node in the activity diagram needs to be considered as a state and the edges are considered as transitions. This concept is far from the reality. In the case of some nodes, like decision making nodes, the node cannot be considered as a normal state, because it has no state. Hence, such nodes are to be considered as a dummy state.

Fig. 4 shows a part of the DTD of the activity graph (for UML 1.x). We can see that the features of the activity graph are considered as a type of state machine features, like state machine context, state machine top, transitions, sub machines etc. The attributes of the activity graph are defined on the basis of state machine attributes, like context, sub states etc. many of these details, like Composite State, Sub Machine State etc., are not relevant for an activity diagram or they don’t match with the actual meaning of the activity diagram.

For example, state machine top defines different type of states, like Simple State, Composite state etc., that can appear in a state machine and for activity diagram as well. Anyhow, it is not defining or distinguishing different types of actions that can occur in an activity diagram, like Call Operation Action, Call Behavior Action etc. Here the only consideration is for states, not for actions. From this DTD itself we can find the complexity of the representation of activity diagram. UML 2.0 tries to reduce this complexity by changing the semantics of the activity diagram.

The semantics of activity diagram is like a graph in UML 2.0. It makes activity diagrams more consistent and easier to understand by moving from semantics rooted in state machines toward semantics rooted more in *Petri nets* [8]. Petri Nets are used to describe the state changes in a system with transitions. Petri Nets consist of two types of node; *places* and *transitions*.

In the proposed DTD we are following the UML 2.0 specifications. It basically represents the activity diagram as a set of nodes and edges like Petri Nets. Fig. 1 and Fig. 2 show the structure of the activity diagram representation. UML 2.0 has defined some predefined actions that can appear in an activity diagram and other UML diagrams. It is included as the xmi:type attribute of the node in the proposed DTD. So, each node can be distinguished (action, decision, initial, final etc.) using this attribute. Edges can have some values like conditions. This can be written as the *guard* attribute of the edge. The control flows (or, token flows) can be identified with the help of edges.

We can compare the representation of the activity diagram using the old DTD and the proposed DTD. The Fig. 5 shows an example using the old DTD. This example is to get the PIN number from the user and to verify whether it's a valid PIN or not. In this example, the activity diagram is represented in terms of state machine. The representation includes two parts; StateMachine.top which includes all nodes in the activity diagram, and StateMachine.transitions which includes all edges in the diagram.Each node in the activity diagram is considered as a *state* here. Decision node, initial node etc. are considered as pseudo states, because they are not doing an action.

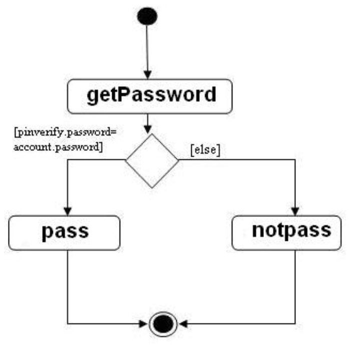
The entire activity diagram is considered as a composite state. Composite state is a state which is composed of sub states. This representation corresponds to the concept of *activity* in UML 1.x. UML2.0 considers *actions*, directly executable (atomic) actions, but not activities. The elements like, StateMachine.top, CompositeState, CompositeState. subvertex etc. are overheads in the XMI representation of the activity diagram. In the proposed DTD, we avoid these overheads by considering the activity diagram as a set of nodes and edges. It does not include any state machine specific elements in the activity diagram representation. Hence reduces the complexity of XMI representation.

In short the new DTD gives an unambiguous and independent representation of activity diagrams. It clearly defines the actions, rather than states, in the activity diagram.

VI. CONCLUSION

This paper introduces a new DTD for XMI representation of UML activity diagram. To the best of our knowledge, there are no DTDs published for representing UML2.0 activity diagrams. When we convert the activity diagram to XMI, the DTD is very important. They are highly essential for successful automatic code generation. The proposed DTD gives an easy way to represent activity diagram in XMI format. This reduces the complexity of XMI representation because this DTD represents the activity diagrams as a set of nodes and edges. This DTD will lead to portability of the UML diagrams among heterogeneous modeling tools. This DTD is conformed to UML 2.0.

This paper also introduces an algorithm to convert the UML activity diagram to the XMI format based on our proposed DTD. In this algorithm we can use already available efficient graph traversal algorithms which will ensure the overall efficiency of the conversion algorithm. The efficiency can be again improved by the use of hash tables to implement the *Node* and *Edge* lists in the algorithm.



```
<UML:ActivityGraph xmi.id='0_01' name='AD1'>
 <UML:StateMachine.top>
  <UML:CompositeState xmi.id='1_01'>
  <UML:CompositeState.subvertex>
  <UML:Pseudostate xmi.id='2_01' kind='initial'/>
  <UML:ActionState xmi.id='3_01'>
  <UML:State.entry>
  <UML:SendAction xmi.id='3_02' name='getpassword'>
  <UML:Action.target>                    <UML:ObjectSetExpression
xmi.id='3_03'/> </UML:Action.target>
  <UML:Action.script><UML:ActionExpression
xmi.id=''3_04''/></UML:Action.script>
  </UML:SendAction> </UML:State.entry> </UML:ActionState>
  .
  .
 </UML:CompositeState> </UML:StateMachine.top>
 <UML:StateMachine.transitions>
  <UML:Transition xmi.id='4_03' source='3_04' target='3_05'>
  <UML:Transition.guard>
  <UML:Guard xmi.id='4_04'>
  <UML:Guard.expression>
  <UML:BooleanExpressionxmi.id='4_05'
body='pinverify.password=account.password'/>
   .
   .
</UML:StateMachine.transitions>
</UML:ActivityGraph>
```

Fig. 5 Example using old DTD

REFERENCES

[1] OMG *Unified Modeling Language* (OMG UML), Infrastructure,V2.1.2 , www.omg.org, 2007.

[2] Benoit Marchal, *Design XML vocabularies with UML tools*, http://www.ibm.com/ developer works/xml/library/x-wxxm23/, 2004.

[3] Deepak Vohra, *Developing UML Diagrams from XMI in Oracle*, 2004. http://www.oracle.com/technology.

[4]OMG- *UML DTD Specification*, www.omg.org.

[5] OMG - *MOF 2.0/XMI Mapping Specification*, version 2.1, formal/05-09-01, 2005.

[6] William Harrison, Charles Barton, Mukund Raghavachari, *Mapping UML Designs to Java*, IBM T.J Watson Research Center, New York, OOPSLA Conference, ACM, 2000.

[7] Eivind Bjoraa, Torgeir Myhre, Espen Westlye Straapa, *Generating Java Skeleton From XMI*, Paper presented on Open Distributed Systems, Agder University College.

[8] John Deacon, *Object-Oriented Analysis and Design*, Addison-Wesley, ISBN 0-321-26317-0, 2005.

[9] John Deacon, *Object-Oriented Analysis and Design: A Pragmatic Approach*, Published by Addison-Wesley; Pearson Education, ISBN 0321263170, 9780321263179, 2004.

[10] OMG *Unified Modeling Language* (OMG UML), Superstructure,V2.1.2,2007.