Enhancement of Action Description Language for UML Activity Diagram Review

Chinnapat Kaewchinporn and Yachai Limpiyakorn

Department of Computer Engineering, Chulalongkorn University,
Bangkok 10330, Thailand
Chinnapat.K@student.chula.ac.th, Yachai.L@chula.ac.th

Abstract

The UML activity diagram is graphical presentation that describes the operational process and related causes used in each stage of the system. For understanding and accurate communication, the UML standard is required for determining the congruent and consistent format application. To detect the misconception and incorrect notation, this paper presents an automation approach to reviewing UML activity diagrams based on a domain specific language, called Action Description Language (ADL). The input is the UML activity diagram in the XMI format. Due to the variations of XMI formats, the review process starts with the standardization of the XMI source file. Next, the ADL script will be created using the information extracted from the XMI file. The ADL script will then be verified against the UML constraints defined in the UML standard 2.4.1. The inspection result will be reported. In case of valid activity diagrams, the ADL scripts will be parsed to the ADL semantic model as the final output from the system. The demonstration of the proposed method was performed via three cases. Currently, the implemented prototype can review only the activity diagrams created by ArgoUML and Modelio.

Keywords: *Domain Specific Language, UML Activity Diagram, Verification, Process Improvement*

1. Introduction

The Unified Modeling Language (UML) is becoming a standardized modeling notation for expressing the object-oriented model, and a widely used design tool in software development. UML is a visual modeling language, and it consists of a set of diagrams. Static diagrams are used to depict the static structure of a program, whereas dynamic diagrams specify how the control flows of the program should behave [1]. For quality and standardization in the design, the UML Specification has been defined by the Object Management Group (OMG) for controlling the semantics and notation of UML. OMG is a consortium, which originally aimed at setting standards for distributed object-oriented systems, and is now focusing on modeling programs, systems and business processes, as well as model-based standards [2].

However, for large and complex systems, manually creating UML diagrams with graphic notation is error-prone and may cause data and behavior inconsistency. In addition, software engineers may misunderstand the semantics and notation, resulting in the diagrams nonconformance to UML specification.

In literature, researchers proposed several methods to verify and validate the UML diagrams. Kotb and Katayama [3] proposed a novel XML semantics approach for checking

the semantic consistency of XML document using attribute grammar techniques. Shen et al. [4] implemented a toolset which could examine both static and dynamic aspects of a model. The toolset was based on the semantic model using Abstract State Machines presented in [5]. Flater, *et al.*, [6] proposed human-readable Activity Diagram Linear Form (ADLF) for describing activity diagrams in text format. Narkngam and Limpiyakorn [7-9] introduced a preventive approach to rendering valid activity diagrams with a domain specific language called Action Description Language (ADL).

In this paper, the enhancement of the Action Description Language invented in [7-9] is carried out to verify existing activity diagrams whether they conform to the UML specification version 2.4.1 [2]. Currently, the prototype developed in this work can merely inspect the activity diagrams created by ArgoUML and Modelio due to the restriction caused by the variations of the XMI format generated by different UML tools. This research work could be useful for software process improvement as the automation of reviews would lessen defects and resources consumed during software development.

2. Action Description Language (ADL)

A domain specific language (DSL) contains the syntax and semantics that model a concept at the same level of abstraction provided by the problem domain [10]. Some DSLs use a parser to populate a semantic model or an object model, as it provides a clear separation of concerns between parsing a language and the resulting semantics [11]. A semantic model can be used to generate results such as activity diagrams and XML Metadata Interchange (XMI) that can be used for documentation and data analysis, respectively.

Action Description Language (ADL) [7-9] is a domain specific language used for creating activity diagrams that conform to UML specification. The design of ADL covers four elements required to constitute a DSL: structure, constraints, representation, and behavior. The ADL metamodel (Figure 1) illustrates the language structure consisting of Element, Object, Relation, Guard, and Action. Constraints can be defined as validation and verification rules described in [9], serving the purposes of preventing data inconsistency, and fortifying conformance to UML specification, respectively. Representation can be visualized with a digraph using Graphviz as used in the research or any other graph visualization software. Behavior is accomplished by means of QVT applied for model-to-model transformation. For constructing activity diagrams, QVT or query/ view/ transformation specification [12] is used to describe the transformation from the semantic model of ADL metamodel to the semantic model of the activity diagram metamodel shown in Figure 2.

The current ADL covers the generation of intermediate activity diagrams as shown in Figure 3. The research work [7] has defined the syntax of ADL for an action, a sequence of actions, and a decision, as illustrated in Figure 4, Figure 5, and Figure 6, respectively.

The Action syntax denotes an activity in the operational process. Each action must have at least one output, but it may not have input. In addition, there may or may not exist the pre-condition or post-condition.

The syntax of a sequence of actions can be obtained from the object that is shared between the two actions. Each sequence of actions must have the direction from the source action to the destination action. There are two types of the sequence of actions: 1) sequence of actions with explicit objects, and 2) sequence of actions with implicit objects.

The syntax of iteration is not defined since a loop can be directly derived from object relations. And the controls can be automatically detected from its individual pattern as described in [9].

3. Automation of Activity Diagram Review

While the research [7-9] proposed the preventive approach to creating UML activity diagrams, this paper presents the corrective approach to reviewing the existing UML activity diagrams. Figure 7 illustrates the research method how we adapt ADL for verifying the conformance to UML specification of existing activity diagrams. The review process of UML activity diagrams consists of four main steps as briefly explained in the following subsections.

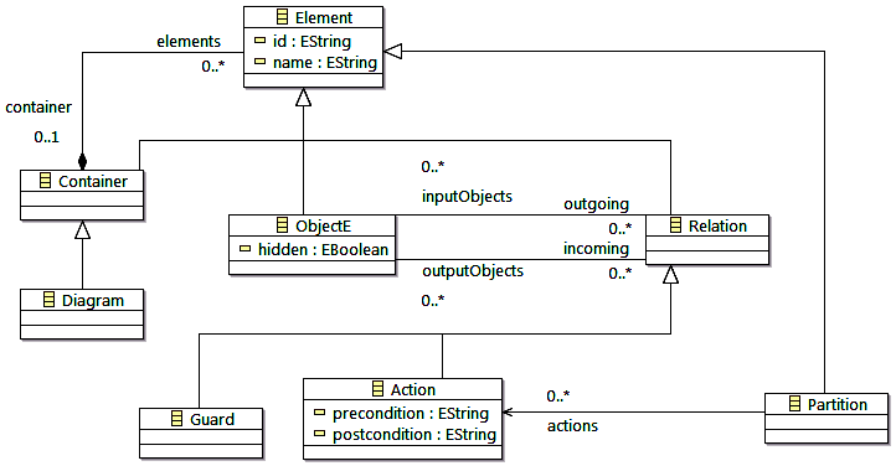

Figure 1. ADL metamodel

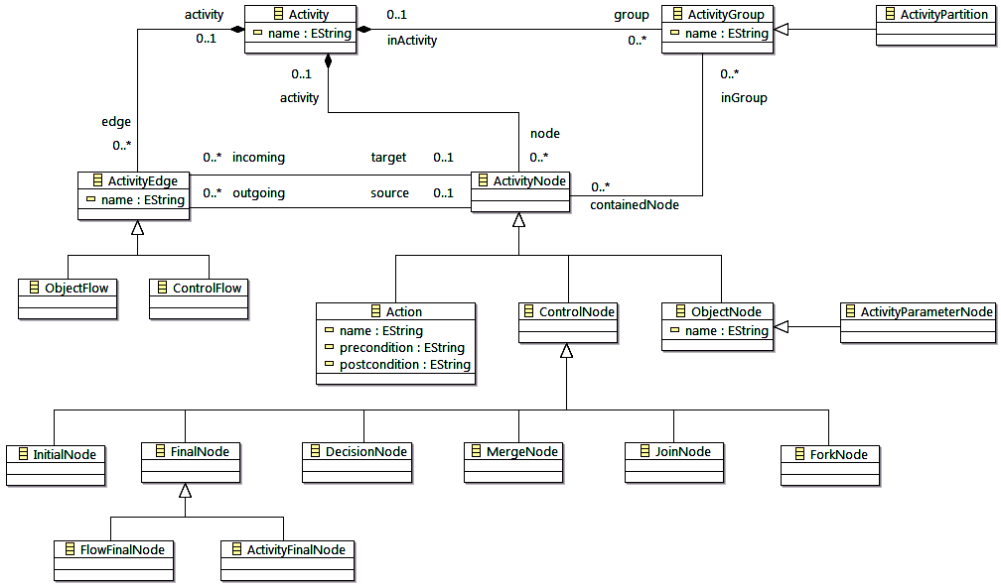

Figure 2. Intermediate activity diagram metamodel

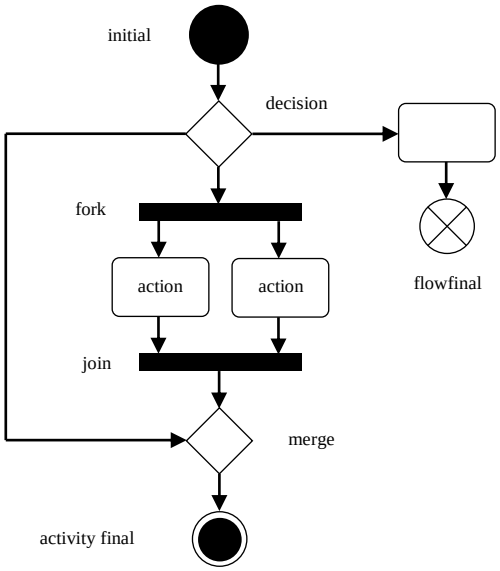

Figure 3. Components of intermediate activity diagram [9]

```
action id [[isLocallyReentrant=[true|false]]]
    [name 'string']
    [<- OID1[,OID2...[,OIDN]]] --input objects
    [-> OID3[,OID4...[,OIDN]]] --output objects
    [precondition 'string']
    [postcondition 'string']

end
```

Figure 4. ADL syntax for defining an action [9]

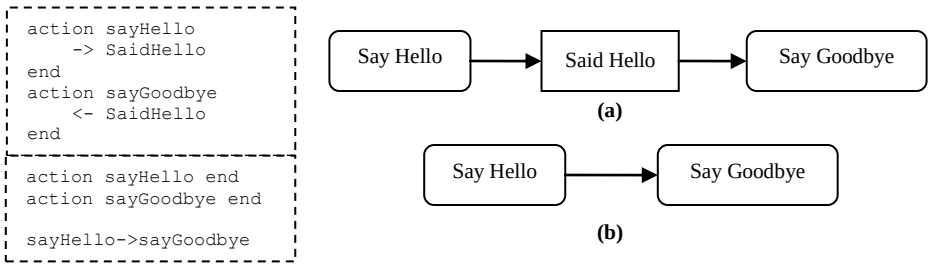

Figure 5. ADL syntax for defining a sequence of actions with: (a) explicit object; (b) implicit object [9]

```
decision ['input']
    if 'condition1' then id1
    if 'condition2' then id2
    ...

end
```

Figure 6. ADL syntax for defining a decision [9]

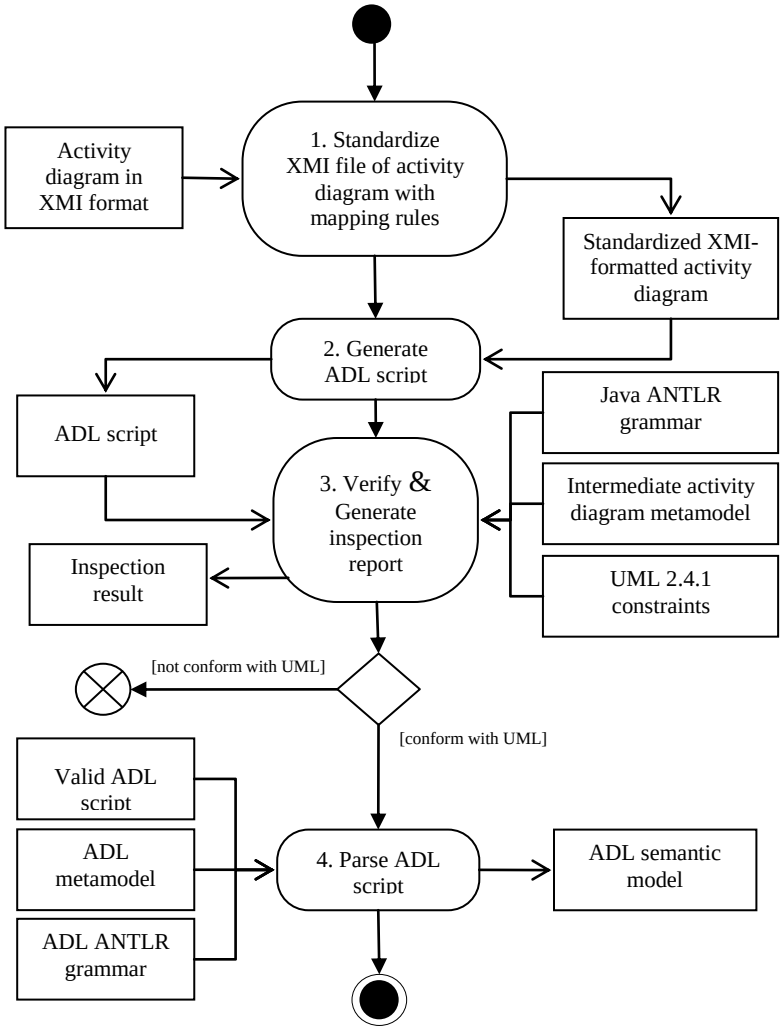

Figure 7. Review process of UML activity diagram

3.1. Standardize XMI file of Activity Diagram with Mapping Rules

The input of the system is the XMI file of the UML activity diagram to be reviewed. Since different UML tools support different XMI formats and the UML notations may vary due to enhanced version, the XMI standard converter is therefore developed in this work. The component is responsible for converting the XMI format of the input activity diagram to the defined XMI format. The converter is developed as Eclipse Plug-in [13] and it uses mapping rules for standardization. The mapping rules are particularly defined for converting the XMI format of ArgoUML and Modelio as shown in Figure 8 and Figure 9, respectively.

3.1.1. Data extraction from ArgoUML activity diagrams: ArgoUML is developed by Jason E. Robbins using Java language. It is open-source software under Eclipse Publish License 1.0. ArgoUML supports XMI standard version 1.2 and UML standard version 1.4. In

order to generate ADL script in the next step, some important information must be detected and extracted as the followings:

1. The "UML: Model" is a tag that provides description of the activity diagram: identity of diagram, diagram name, and other properties.
2. The "UML:CompositeState.subvertex" is a tag that collects all objects in the activity diagram. The objects of activity diagram include action, object, and control nodes of which the tag name is an identity of object type. Example tag names are: UML:ActionState, UML:ObjectFlowState, UML:Pseudostate, and UML:FinalState. Inside the tags, UML:StateVertex.outgoing and UML:StateVertex.incoming are defined to provide incoming edge tag and outgoing edge tag.
3. The "UML:StateMachine.transitions" is a tag that collects all edges in the activity diagram. The property of edge is 1-1 relationship between a node and another node. The source and target tags are labeled as "UML:Transition.source" and "UML:Transition.target".

3.1.2. Data extraction from Modelio activity diagrams: Modelio is developed by ModelioSoft. It is open-source software under GPLv3 License. Modelio supports XMI standard version 2.1 and UML Standard version 2.0. In order to generate ADL script in the next step, some important information must be detected and extracted as the followings:

1. The "packagedElement" is a tag that provides description of the activity diagram: diagram type, identity of diagram, and diagram name.
2. The "node" is a tag that provides description of a node in the activity diagram: node type, identity of node, node name, identity of incoming edges, and identity of outgoing edges. Inside incoming edge and outgoing edge, if it has more than one edge, each edge will be separated with the space character.
3. The "edge" is a tag that provides description of an edge in the activity diagram: edge type, identity of edge, edge name, identity of source node, and identity of target node.

3.2. Generate ADL Script

The second step is to transform the XMI standard document obtained from the previous step into the ADL script. The method is to reverse the approach presented in [7, 8, 9]. If the resulting ADL script fails during the verification in the next step, it can be accessed and revised.

The initial step of ADL script generation is to parse the XMI standard document obtained from the previous step to node and edge array lists. Next, the ADL script generator will inspect node/edge and generate the script based on each item. The following three steps are carried out during the script generation.

1. Generate actions.
2. Generate decisions.
3. Generate sequence of actions.

3.2.1. Generate actions: The syntax of actions is determined by the object type associated with an action. There are two types of objects: 1) explicit object, and 2) implicit object. The tool will generate syntax for actions with implicit objects.

3.2.2. Generate decisions: The syntax of decisions is recursively generated. The principles of decision syntax generation are as follows:

i. Decision syntax will be generated when a decision node is connected to more than one edge.

ii. The result of conditions is in the form of an action or fork node only.

iii. If the result of decision is another decision node then create nested syntax inside the decision syntax.

R1: If tag is "ActivityGraph" then store activity name in variable.
R2: If tag is "StateMachine.transitions" then change the status of stored edge to *ready*.
R3: If status of stored edge is *ready* and tag is "Transition" then do the followings:
  R3-1: create temporary edge,
  R3-2: define new identity of node from order the node is found,
  R3-3: store identity of node from attribute of tag,
  R3-4: store edge name (optional).
R4: If tag is "Transition.source" then change the status of stored source of edge to *ready*.
R5: If tag is "Transition.target" then change the status of stored target of edge to *ready*.
R6: If status of stored source of edge is *ready* and tag is "ActionState", "ObjectFlowState", or "Pseudostate" then store identity of source node to temporary edge.
R7: If status of stored target of edge is *ready* and tag is "ActionState", "ObjectFlowState", "Pseudostate", or "FinalState" then store identity of target node to temporary edge.
R8: If tag is "CompositeState.subvertex" then change the status of stored node to *ready*.
R9: If status of stored node is *ready* and tag is "ActionState" then do the followings:
  R9-1: create temporary node,
  R9-2: define new identity of node from order the node is found,
  R9-3: store identity of node from attribute of tag,
  R9-4: store node name,
  R9-5: define type of node "Action",
  R9-6: change the status of considered node to *ready*.
R10: If status of stored node is *ready* and tag is "ObjectFlowState" then do the followings:
  R10-1: create temporary node,
  R10-2: define new identity of node from order the node is found,
  R10-3: store identity of node from attribute of tag,
  R10-4: store node name,
  R10-5: define type of node "Object",
  R10-6: change the status of considered node to *ready*.
R11: If status of stored node is *ready* and tag is "Pseudostate" and tag type is "initial" then do the followings:
  R11-1: create temporary node,
  R11-2: define new identity of node from order the node is found,
  R11-3: store identity of node from attribute of tag,
  R11-4: store node name,
  R11-5: define type of node "InitialNode",
  R11-6: change the status of considered node to *ready*.
R12: If status of stored node is *ready* and tag is "Pseudostate" then do the followings:
  R12-1: create temporary node,
  R12-2: define new identity of node from order the node isfound,
  R12-3: store identity of node from attribute of tag,
  R12-4: store node name,
  R12-5: define node type with tag type,
  R12-6: change the status of considered node to *ready*.
R13: If status of stored node is *ready* and tag is "FinalState" then do the followings:
  R13-1: create temporary node,
  R13-2: define new identity of node from order the node is found,
  R13-3: store identity of node from attribute of tag,
  R13-4: store node name,
  R13-5: define type of node "ActivityFinalFlow",
  R13-6: change the status of considered node to *ready*.
R14: If status of considered node is *ready* and tag is "StateVertex.outgoing" then change the status of considered outgoing edge to *ready*.
R15: If status of considered node is *ready* and tag is "StateVertex.incoming" then change the status of considered incoming edge to *ready*.
R16: If status of considered outgoing edge is *ready* and tag is "Transition" then store identity of edge to temporary node in

outgoing variable.
R17: If status of considered incoming edge is *ready* and tag is "Transition" then store identity of edge to temporary node in incoming variable.
R18: If status of stored edge is *ready* and closed tag is "Transition" then add temporary edge to edge array list.
R19: If status of stored node is *ready* and closed tag is "ActionState", "ObjectFlowState", and "Pseudostate" then add temporary node to node array list.
R20: If found any closed tag then change status to *not ready*.

Figure 8. ArgoUML mapping rules

R1: If tag is "packagedElement" then store activity name in variable.
R2: If tag is "node" then do the followings:
- R2-1: create temporary node ,
- R2-2: define new identity of node from order the node is found,
- R2-3: store identity of node from attribute of tag,
- R2-4: consider incoming edge and store in array variable,
- R2-5: consider outgoing edge and store in array variable,
- R2-6: define type of node from attribute of tag,
- R2-7: consider temporary node; if type of node is initial node or activity final node then store temporary node in temporary object variable and add it to node array list after parsing has been completed, else add temporary node to node array list.

R3: If tag is "edge" then do the followings:
- R3-1: create temporary edge,
- R3-2: define new identity of edge from order the edge is found,
- R3-3: store identity of edge from attribute of tag,
- R3-4: store edge name. (optional)

Figure 9. Modelio mapping rules

3.2.3. Generate sequence of actions: The syntax of a sequence of actions is recursively generated. The principles of sequence of actions syntax generation are as follows:

i. Sequence of actions starts with an action node only.

ii. If the next node in a sequence is "ActivityFinalFlow", "Decision", or "Fork" then stop finding and return the sequence of actions.

iii. If the next node in a sequence is "Merge", "Join", or "Object" then skip the node and continue finding the sequence by considering the next node.

iv. If the next node in the sequence is "FlowFinal" then return "break" to concatenate with previous syntax and return the sequence of actions.

v. If the next node in the sequence is "Action" then return activity name of node to concatenate with previous syntax and continue finding the sequence by considering the next node.

3.3. Verify and Generate Inspection Result

The third step is to examine the ADL script generated from the previous step. Java ANTLR grammar (Figure 10) is developed for the inspection process and report generation. The Intermediate activity diagram metamodel (Figure 2) is used to indicate what elements will be examined against the constraints of UML 2.4.1. The elements of Intermediate activity diagrams that need be examined include: Activity, ActivityGroup, ActivityPartition, ActivityEdge, ActivityNode, ObjectFlow, ControlFlow, Action, ActivityParameterNode, InitialNode, FinalNode, DecisionNode, MergeNode, JoinNode, ForkNode, FinalFlowNode, and ActivityFinalNode. The review process ensures that the syntax of each element

comprising the model conforms to the standards and guidelines specified by OMG [2]. All the constraints checking are listed in the following subsections.

3.3.1. Constraints of Activity

i. The nodes of the activity must include one ActivityParameterNode [2] for each parameter.

ii. An activity cannot be autonomous and have a classifier or behavioral feature context at the same time.

iii. The groups of an activity have no super groups.

```
prog: block;
scope {
        String diagramName;
        ArrayList actionArray = new ArrayList<ActionNode>();
        ArrayList decisionArray = new ArrayList<DecisionNode>();
        ArrayList sequenceArray = new ArrayList<SequencePath>();
}
@init {
        $block::actionArray = new ArrayList();
        $block::decisionArray = new ArrayList();
        $block::sequenceArray = new ArrayList();
}
Diagram:
   $block::diagramName = STRING;

Element:
   Action | Decision | Sequence;

Action:
   $block::actionArray.add(new ActionNode(ACTIONID, Action_Opts*));

Action_Opts:
   'name' name=STRING |
   ('<-' | 'inputs') inputObjects=MultiOID |
   ('->' | 'outputs') outputObjects=MultiOID |
   'precondition' precondition=STRING |
   'postcondition' postcondition=STRING;

Decision:
   $block::decisionArray.add(new DecisionNode(ACTIONID, Guard*));

Guard:
   'if' condition=STRING 'then' (then+=Expression)+
   ('else' (els+=Expression)+)*
   'endif';

Expression:
   MultiAID | Guard;

Sequence:
   $block::actionArray.add(new SequencePath(ACTIONID (('->' | 'then') next+=SequenceRight)+));

SequenceRight:
   id=(MultiAID) ('[' expression=STRING ']')?;
```

Figure 10. Java ANTLR grammar

3.3.2. Constraints of ActivityEdge

i. The source and target of an edge must be in the same activity as the edge.

ii. Activity edges may be owned only by activities or groups.

3.3.3. Constraint of ActivityNode

i. Activity nodes can only be owned by activities or groups.

3.3.4. Constraints of ObjectFlow

i. Object flows may not have actions at either end.

ii. Object nodes connected by an object flow, with optionally intervening control nodes, must have compatible types. In particular, the downstream object node type must be the same or a super type of the upstream object node type.

iii. Object nodes connected by an object flow, with optionally intervening control nodes, must have the same upper bounds.

3.3.5. Constraint of ControlFlow

i. Control flows may not have object nodes at either end, except for object nodes with control type.

3.3.6. Constraint of ObjectNode

i. All edges coming into or going out of object nodes must be object flow edges.

3.3.7. Constraints of InitialNode

i. An initial node has no incoming edges.

ii. Only control edges can have initial nodes as source.

3.3.8. Constraint of FinalNode

i. A final node has no outgoing edges.

3.3.9. Constraints of DecisionNode

i. A decision node has one or two incoming edges and at least one outgoing edge.

ii. The edges coming into and out of a decision node, other than the decision input flow (if any), must be either all object flows or all control flows.

iii. The decisionInputFlow [2] of a decision node must be an incoming edge of the decision node.

iv. A decision input behavior has no output parameters, no in-out parameters and one return parameter.

v. If the decision node has no decision input flow and an incoming control flow, then a decision input behavior has zero input parameters.

vi. If the decision node has no decision input flow and an incoming object flow, then a decision input behavior has one input parameter whose type is the same as or a super type of the type of object tokens offered on the incoming edge.

vii. If the decision node has a decision input flow and an incoming control flow, then a decision input behavior has one input parameter whose type is the same as or a super type of the type of object tokens offered on the decision input flow.

viii. If the decision node has a decision input flow and a second incoming object flow, then a decision input behavior has two input parameters, the first of which has a type that is the same as or a super type of the type of the type of object tokens offered on the non-decision input flow and the second of which has a type that is the same as or a super type of the type of object tokens offered on the decision input flow.

3.3.10. Constraints of MergeNode

i. A merge node has one outgoing edge.

ii. The edges coming into and out of a merge node must be either all object flows or all control flows.

3.3.11. Constraints of JoinNode

i. A join node has one outgoing edge.
self.outgoing->size() = 1

ii. If a join node has an incoming object flow, it must have an outgoing object flow, otherwise, it must have an outgoing control flow.
(self.incoming.select(e | e.isTypeOf(ObjectFlow)->notEmpty() implies self.outgoing.isTypeOf(ObjectFlow)) and (self.incoming.select(e | e.isTypeOf(ObjectFlow)->empty() implies self.outgoing.isTypeOf(ControlFlow))

3.3.12. Constraints of ForkNode

i. A fork node has one incoming edge.

ii. The edges coming into and out of a fork node must be either all object flows or all control flows.

3.4. Parse ADL Script

The final step is to generate the ADL semantic model by parsing the validated ADL script. The ADL ANTLR grammar, accompanied with the ADL metamodel (Figure 3), is used to transform the ADL syntax into the semantic model. The ADL ANTLR grammar is a combined grammar containing both grammars and lexer. It has been constructed upon ANTLR v.3 [14] which is a parser generator used to create a parser for compiling ADL scripts.

The resulting ADL semantic model consists of nodes, object evidence, guard condition objects, and relationships. The semantic model is useful for further applications such as generating: test cases, design document, Java source code, and *etc*.

4. Demonstration

Three activity diagrams were selected as the cases to demonstrate how the proposed approach works. The first case is the valid case, while the other two represent the invalid cases. The first case represents the trouble ticket scenario of which the activity diagrams are created by ArgoUML and Modelio. Figure 11 illustrates the activity diagram of trouble ticket scenario created by ArgoUML whereas that created by Modelio is depicted in Figure 12.

Both diagrams were transformed to XMI format using the export command provided within the tools. Initially, the XMI files of the input activity diagrams were converted into standard XMI format defined in this work. Excerpt of the standardized XMI format is illustrated in Figure 13.

The next step is to generate the ADL script from the standardized XMI files. All actions (Figure 14), sequences of actions (Figure 15), decisions (Figure 16), and their information were detected and extracted from the XMI files for being used to create the ADL script as shown in Figure 17. The ADL script is correct compared to that generated in [7-9] using the same activity diagram.

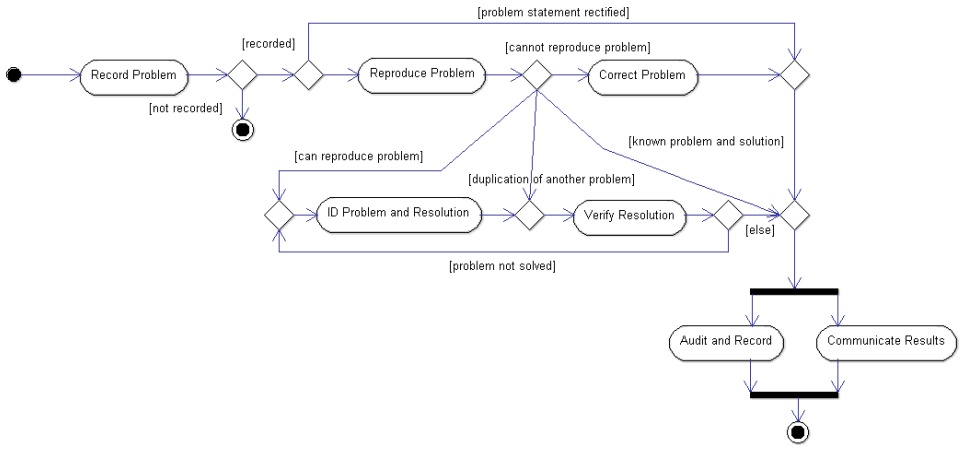

Figure 11. First case of examined activity diagram created by ArgoUML

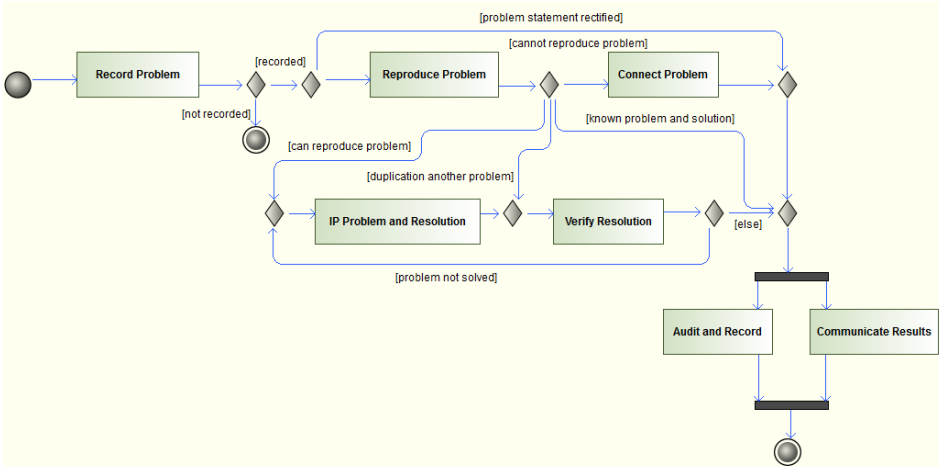

Figure 12. First case of examined activity diagram created by Modelio

```
<?xml version="1.0" encoding="ASCII"?>
<intermediatead:Activity xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:intermediatead="http://th.ac.chula.eng.cp.adl.emf/intermediatead/1.0" name="ex6">
<edge xsi:type="intermediatead:ObjectFlow" source="//@node.17" target="//@node.0"/>
<edge xsi:type="intermediatead:ObjectFlow" source="//@node.0" target="//@node.1"/>
<edge xsi:type="intermediatead:ObjectFlow" source="//@node.1" target="//@node.2" name="recorded"/>
<node xsi:type="intermediatead:Action" incoming="//@edge.0" outgoing="//@edge.1" name="Record
        Problem"/>
<node xsi:type="intermediatead:DecisionNode" incoming="//@edge.1" outgoing="//@edge.2 //@edge.3"/>
<node xsi:type="intermediatead:DecisionNode" incoming="//@edge.2" outgoing="//@edge.4 //@edge.8"/>
<node xsi:type="intermediatead:Action" incoming="//@edge.4" outgoing="//@edge.5" name="Reproduce
        Problem"/>
<node xsi:type="intermediatead:DecisionNode" incoming="//@edge.5" outgoing="//@edge.6 //@edge.9
        //@edge.22 //@edge.23"/>
</intermediatead:Activity>
```

Figure 13. Excerpt of standardized XMI format

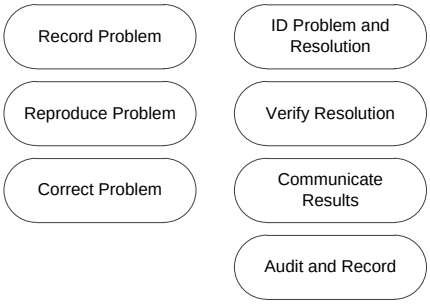

Figure 14. Extraction of all actions in activity diagram of first case

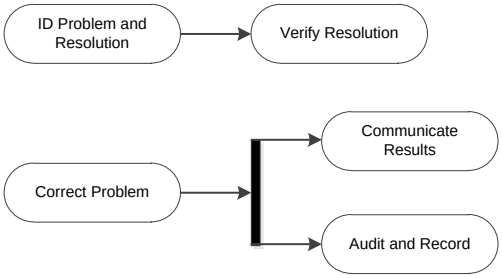

Figure 15. Extraction of all action sequences in first case activity diagram

Figure 18 illustrates the inspection result of the activity diagram of trouble ticket scenario without any error reported.

The activity diagram of the second case (Figure 19) and that of the third case (Figure 21) represent the invalid examples, *i.e.*, nonconformance to UML 2.4.1 standard. The inspection results with the errors reported are shown in Figure 20 and Figure 22, respectively.

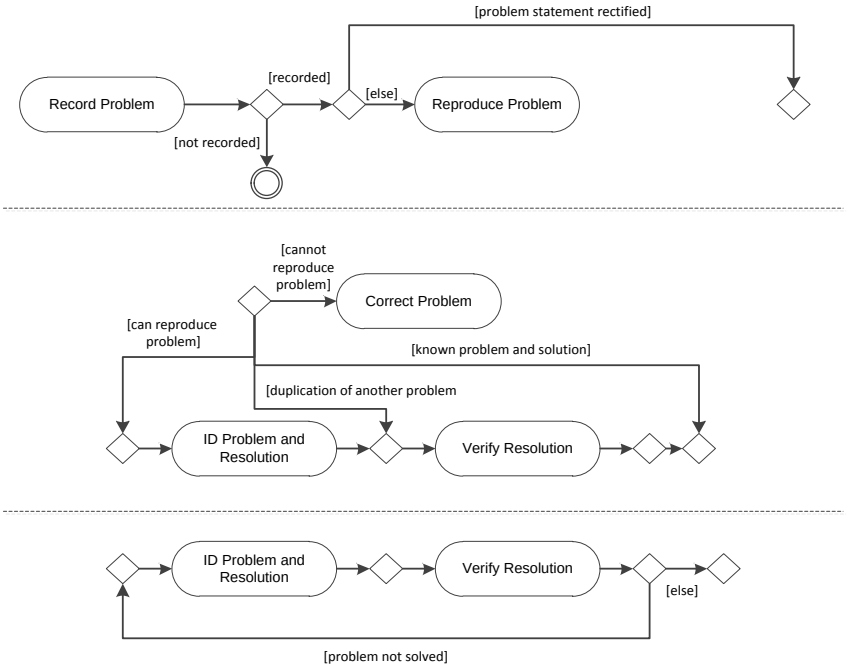

Figure 16. Extraction of all decisions in activity diagram of first case

```
diagram 'ticket trouble scenario'
    action Record Problem end
    action Reproduce Problem end
    action Correct Problem end
    action Id Problem And Resolution end
    action Verify Resolution end
    action Audit And Record end
    action Communicate Result end
    decision from Record Problem
        if 'recorded' then
            if 'problem statement rectified' then Audit And Record and
                Communicate Result
            else if 'else' then Reproduce Problem
                endif
            endif
        else
            if 'not recorded' then break
            endif
        endif
    end
    decision from Reproduce Problem
        if 'cannot reproduce problem' then Correct Problem
        else if 'can reproduce problem' then Id Problem And Resolution
            else if 'duplication of another problem' then Verify Resolution
                else if 'known problem and solution' then Audit And Record and
                        Communicate Result
                    endif
                endif
            endif
        endif
    end
```

```
    decision from Verify Resolution
        if 'problem not solved' then Id Problem And Resolution
        else if 'else' then Audit And Record and Communicate Result
            endif
        endif
    end
    Id Problem And Resolution->Verify Resolution
    Correct Problem->Audit And Record and Communicate Result
end
```

Figure 17. ADL script of trouble ticket scenario

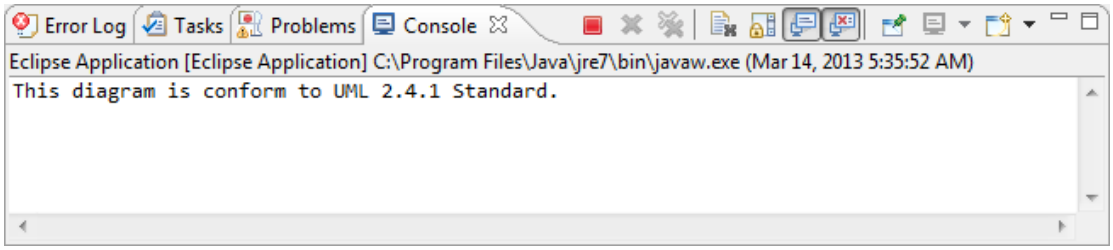

Figure 18. Inspection result of trouble ticket scenario

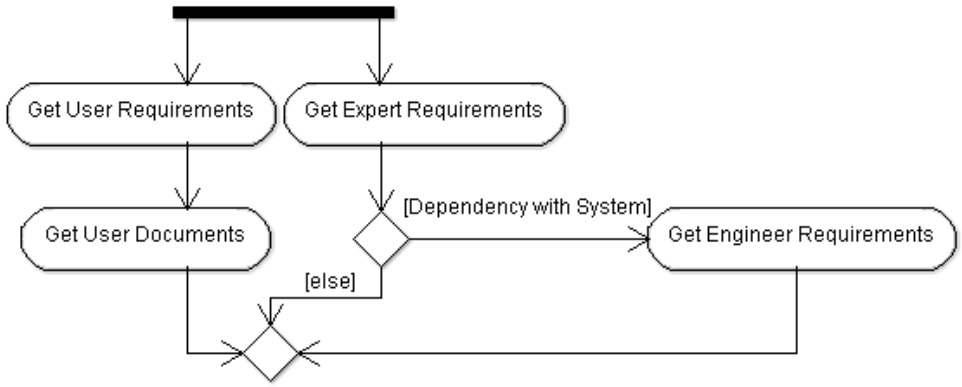

Figure 19. Second case of examined activity diagram

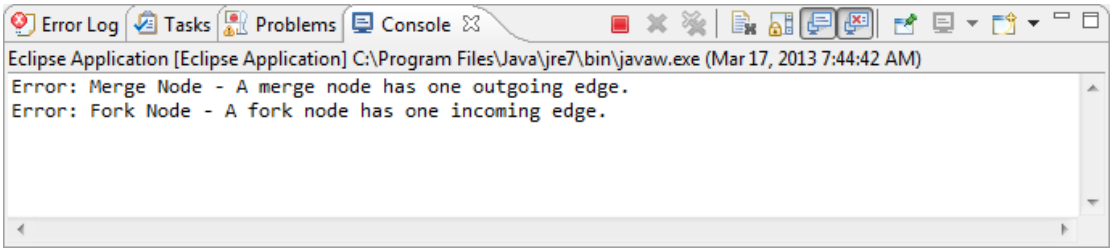

Figure 20. Inspection result of second case examined

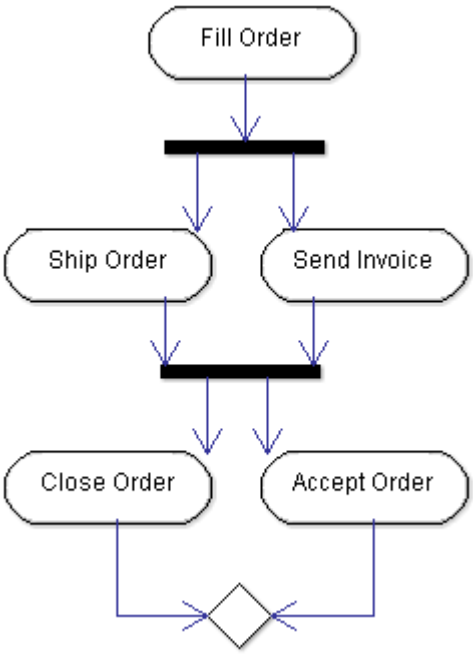

Figure 21. Third case of examined activity diagram

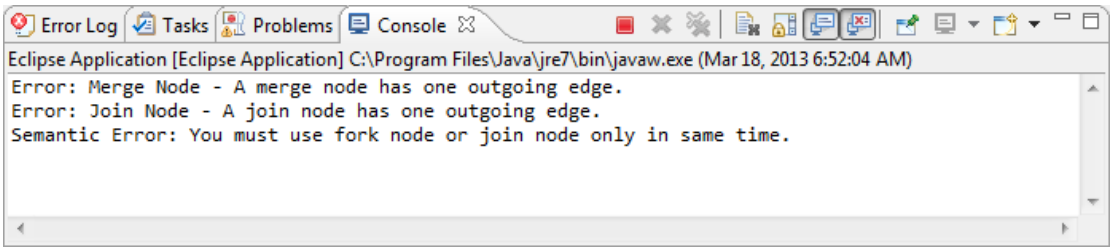

Figure 22. Inspection result of third case examined

5. Conclusion

UML Activity diagrams are widely used as the blueprints for describing procedural logic, business processes and workflows. In mature software development processes, it is suggested to detect and remove defects at the phase they were injected in order to reduce the cost of rework and promote the quality of product. This paper thus presents an automation approach to reviewing the UML activity diagrams. The method deploys a domain specific language called Action Description Language (ADL) created in [7-9]. In this work, we have enhanced ADL for reviewing the existing activity diagrams whether they conform to UML specification v.2.4.1. The proposed method can be considered as the reverse of the research work [7-9], which generates UML activity diagrams from ADL scripts. Conversely, this research generates ADL scripts from existing activity diagrams. However, the final output obtained from the proposed approach is the ADL semantic model, which is useful for further applications, namely the automated generation of test cases, design blueprints, and source

code. Moreover, the ADL scripts obtained from this work can be used for later modification and generation of UML activity diagrams as being processed in the preventive approach presented in [7-9]. Future research work could be the enhancement of mapping rules to the framework capable to support the standardization of various XMI formats.

References

[1] M. Fowler, "UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Ed)". Addison-Wesley Professional, Boston, **(2003)**.
[2] Object Management Group, "Unified Modeling Language: Superstructure version 2.4.1", Object Management Group, Inc., (**2011)**.
[3] Y. Kotb and T. Katayama, "Consistency Checking of UML Model Diagrams Using the XML Semantics Approach", 14th International Conference on World Wide Web, **(2005)** May 10-14, Chiba, Japan.
[4] W. Shen, K. Compton and J. Huggins, "A Toolset for Supporting UML Static and Dynamic Model Checking", 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment, IEEE Computer Society, **(2002)** August 26-29, Oxford, English.
[5] Y. Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms", ACM Transactions on Computational Logic, vol. 1, no.1, **(2000)**, pp. 77-111.
[6] D. Flater, P. A. Martin and M. L. Crane, "Rendering UML Activity Diagrams as Human-Readable Text", Proceedings of the 2009 International Conference on Information and Knowledge Engineering, **(2009)** July 13-16, Las Vegas, United States.
[7] C. Narkngam and Y. Limpiyakorn, "Rendering UML Activity Diagrams as a Domain Specific Language - ADL", 24th International Conference on Software Engineering and Knowledge Engineering, **(2012)** July 1-3; San Francisco, USA.
[8] C. Narkngam and Y. Limpiyakorn, "Designing a Domain Specific Language for UML Activity Diagram", 4th International Conference on Computer Engineering and Technology, **(2012)** May 12-13, Bangkok, Thailand.
[9] C. Narkngam and Y. Limpiyakorn, "Domain Specific Language for Activity Diagram", Ramkhamhaeng Journal of Engineering, vol. 1, **(2012)**.
[10] D. Ghosh. "DSLs in Action", Manning Publications Co., United States, **(2011)**.
[11] M. Fowler, "Domain-Specific Languages", Addison-Wesley Professional, Boston, **(2010)**.
[12] Object Management Group, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1", Object Management Group, Inc., **(2011)**.
[13] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick and T. J. Grose, "Eclipse Modeling Framework", Addison-Wesley Professional, Boston, **(2003)**.
[14] ANTLR Parser Generator, http://www.antlr.org (accessed on January **2012**).

Author

Chinnapat Kaewchinporn received his bachelor degree in Computer Science from King Mongkut's Institute of Technology Ladkrabang in 2011. Currently, he is pursuing the master degree in Software Engineering at Department of Computer Engineering, Chulalongkorn University, Bangkok, Thailand.