Test Cases Generation from UML Activity Diagrams

Hyungchoul Kim, Sungwon Kang, Jongmoon Baik, Inyoung Ko
School of Engineering
Information and Communications University, Korea
{hckim1, kangsw, jbaik, iko}@icu.ac.kr

Abstract

UML activity diagram is a notation suitable for modeling a concurrent system in which multiple objects interact with each other. This paper proposes a method to generate test cases from UML activity diagrams that minimizes the number of test cases generated while deriving all practically useful test cases. Our method first builds an I/O explicit Activity Diagram from an ordinary UML activity diagram and then transforms it to a directed graph, from which test cases for the initial activity diagram are derived. This conversion is performed based on the single stimulus principle, which helps avoid the state explosion problem in test generation for a concurrent system.

1. Introduction

An important challenge in software testing is test cases generation. It is especially difficult when a system contains concurrently executing participants (or objects) since such a system can exhibit different responses depending on the concurrency condition. The Unified Modeling Language (UML) activity diagram is a suitable modeling language for describing the interactions between system objects since activity diagram can be conveniently used to capture business processes, workflows and interaction scenarios.

In the past, there were several approaches for generating test cases from UML activity diagrams [1-5]. But most of them do not deal with concurrency problems and tend to propose only a conceptual idea for test generation.

This paper discusses system test generation for the systems modeled with UML activity diagrams. In this paper, we propose a method that is based on an *I/O explicit Activity Diagram* (IOAD) model, which is an abstraction model obtained from the fully expanded activity diagram by exposing only external inputs and outputs. The advantage of using this intermediate model is in that the IOAD models let us focus only on observable behavior so that only and all the behavior relevant for testing is kept in the model. Then from IOAD model, our method constructs a directed graph to extract test scenarios and test cases. To increase an efficiency of system testing, we use the all-paths test coverage criterion, which has high test coverage and is frequently used in graph-based testing.

The remainder of the paper is organized as follows: In Section 2, we survey representative test generation techniques developed in the past that generates test cases from UML activity diagrams and point out their limitations. In Section 3, we describe our method by defining the IOAD model, discussing test coverage and giving an example that shows the main idea of IOAD model. In Section 4, the method is applied to an example to demonstrate its efficacy. Finally, in Section 5 we conclude our paper by discussing the contributions of the paper and the future research directions.

2. Related Work

In this section, we survey representative researches [1-3] of test cases generation based on UML activity diagrams. The method of the paper [1] generates test cases from UML activity diagrams systematically, which modifies Depth First Algorithm (DFS) for automated generation. The paper [1] does not fully handle fork-join structures. The deficiency of [1] is that any fork node has only two exit edges; evidently, these assumptions limit the applicable scope of the proposed algorithm. Another problem in the paper [1] is that *basic path* defined, *basic paths,* can be found by the DFS algorithm, while the detailed walkthrough of the proposed algorithm shows that some test scenarios are not generated, especially when the test scenarios are derived from the fork-join parts of the activity diagrams [4].

The paper [2] defines the concept of the thin-thread tree, the condition tree, and the data-object tree, as well their relationship with the UML activity diagrams [5]. The previous works dealing with test scenario generations in activity diagrams did not consider data

0-7695-2909-7/07 $25.00 © 2007 IEEE
DOI 10.1109/SNPD.2007.189

objects and input values. This paper proposes traversal algorithms for this [5]. However, the proposed algorithms are incomplete. For example, for the example in Figure 1, it generates only two test scenarios. If we regard each activity as an atomic one, it should generate ten test scenarios.

The paper [3] differs from the papers [1-2] in that it derives usage-scenarios instead of test-scenarios. Though the concept between test scenarios and usage scenarios are different, this paper does not focus on automatic generation of test cases for synchronous events cases. Table 1 compares and summarizes the three approaches.

Table 1. Testing approaches using activity diagram

	Techniques	Limitations
Wang et al [1]	- Table representation - DFS algorithm	- Simple fork-join
Liet al [2]	- Use adaptive agents (AI approach) - Introduce data-object concept	- Path explosion, - Manual
Chandler et al [3]	- Capture and store XMI	- Simple fork-join - Use case scenario generation

3. Test Cases Generation

In this section, we describe our test generation method. First, we represent the original activity diagram as an Input/Output explicit Activity Diagram (IOAD), IOAD is an activity diagram that explicitly shows external inputs to and external outputs. It is explicit in showing the external inputs and outputs in the sense that in IOAD no activity can be further decomposed into constituent activities or tasks, thereby exposing all possible external inputs and outputs.

For the transition, it is necessary to specify each action in the activity diagram owing to interleaving events that they contain. Each action can be divided into *accept event* and *send signal*. In the *IOAD*, model, we suppress non-external input and output. By applying this *IOAD model*, we can solve the state explosion problem that would otherwise occur.

Secondly, we will show how to extract test cases from the *IOAD model*. Based on all-paths test coverage criterion, we will traverse all nodes without revisiting the same node.

3.1 Two Utilitarian Principles for Test Cases Generation

To generate test cases from an *IOAD* model, we adopted two principles that minimizes the number of test cases generated by focusing only on the test cases that are controllable and can be practiced by the testers.. The first of them is *the principle of black-box testing*. Black-box testing describes testing based on external specifications. It observes the operations to be executed. Therefore, input data determine the appropriate action status of the program, and an output data sequentially executed after invoking an input data. As a guide for preparing the software analyst to test the software system, input data and the accuracy of model output data enable the analyst to assure the existence of the data necessary to execute the model in order to ascertain the accuracy of the data generated by the model. We should identify all categories of input data and any special analytical techniques required to obtain those data [6]. If the sources of input data include output from other models, sufficient details should be provided to enable the analyst to assess the appropriateness of those data in solving his problem [6]. Output data should provide the analyst with a methodology for assessing the accuracy of model output data [6]. Since the accuracy of the output values will be judged in relation to the method used to derive them, a review of the algorithms used to compute those output values may be necessary at this point [6].

The second one is *the single stimulus principle* [7]. The single stimulus principle, which prohibits multiple stimuli at stable states and stimuli during transitions in testing, can be used to delimit test purpose or focal points of individual test cases [7].

The overall procedure for generating test cases is as below:

- Derive a system of activity diagram from given specifications
- Derive *IOAD Diagram Model Activity Diagram Model* can be presented via specification writers and implementers)
  - A. Delete data objects and use them as input data
  - B. Delete implicit operations (e.g. read action and write action)
  - C. Leave *send signal* and *accept event actions*
- Based on two principles, construct a graph from *IOAD*. We can focus on the interrelation of sub-systems from a stable state of a system to a stable state

- Traverse nodes based on *all-paths test coverage criterion*
- Generate test scenarios

In this paper, we use the term test scenario and test case interchangeably.

3.2 I/O Explicit Activity Diagram

UML activity diagrams are used typically for workflow representations, the realization of the operation of the design phase, and refinement or sequence ordering and concurrency. In contrast to the UML activity diagram, *IOAD* is a model that suppresses non-external inputs and outputs in the UML activity diagram. Activities are categorized into two external elements: *send signal* and *accept event*. To illustrate this feature, all activities are translated into *send signal* and *accept event* notations. Meanwhile data objects such as *invoice* and *order* are dropped out because these objects are implicit tasks.

There are three characteristics in this model. This model alters the fully specified original activity diagram into an *IOAD*. We represent the model by external inputs and external outputs. In an activity diagram, action can be divided into three categories: input action, output action, and internal action. Internal action is less important to testers whereas a result from function behavior is considered to be a more substantial result.

Based on this model, sequence, decision, and loop cases are all simple constructs. Wherever *send signal* and *accept event* are located, sequential c, Boolean cases (true/false), and loop-non loop cases are simply redesigned. In addition, the fork-join construct should be considered.

3.3 Test Coverage

When analyzing graphs, we choose the basic path and used the first search algorithm. These two principles have been proved in graph testing to generate proper test scenarios. When calculating a path of an activity diagram, if each activity in the path occurs only once, we call such a path a basic path of the activity diagram [4]. The diagram composed by all the basic paths of an activity diagram is called *a basic activity diagram of the activity diagram* [4]. DFS algorithm is used to search all nodes, from the first node to the end node, to calculate all the paths of an activity diagram. When we traverse an activity diagram from the initial activity state to the final activity state by DFS algorithm, we ensure that the loops are executed one at a time and that all action states and transitions are covered. Thus we arrive at a number of basic paths. This number of basic paths is generally acceptable in practice. So we define these basic path based coverage criteria as the test completion criteria.

An intuitive approach is to require a test suite to cover all possible paths in a flow graph. But it is not a practical approach because many flow graphs contain a huge number or even an infinite number of paths [8]. To adopt a practical path-coverage criterion, we need to select a representative subset of all-paths to retest and the selection needs to remove redundant information in a path [8]. There are two ways: The first is to remove redundant nodes, and the second is to remove redundant edges [8].

In graph theory, an *elementary path* is a path with no repeat occurrences of any node, and a *simple path* is a path with no repeat occurrence of any edge [8]. With these restrictions, given a flow graph, there are usually a very limited number of elementary and simple paths [8].

3.4 An Example

The fragment of an activity diagram in Figure 1 has a simple fork-join structure.

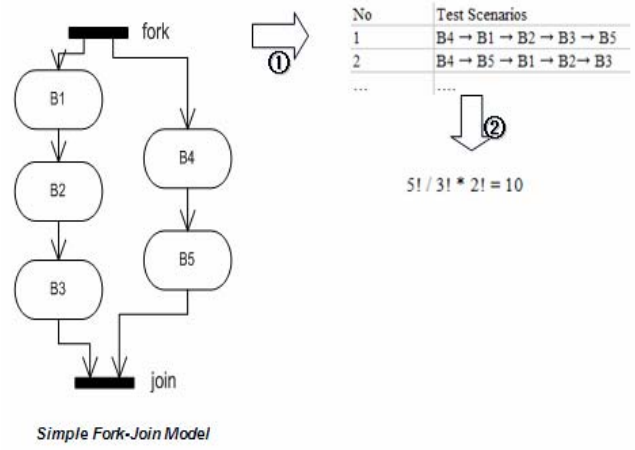
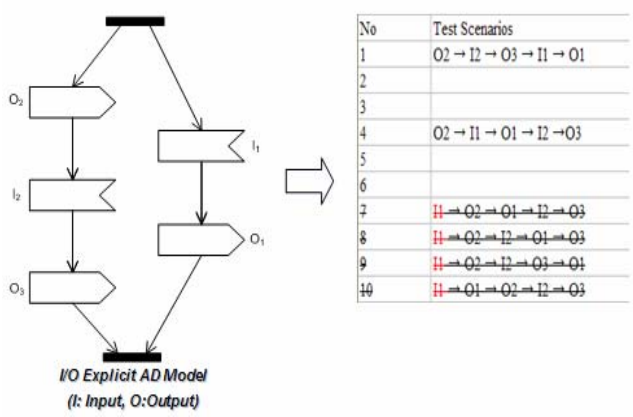


Fig 1. A simple fork-join structure

As can be seen in the example in Figure 1, the original activity diagram generates ten test scenarios (5! /3!*2! = 10). But, this example is different from *IOAD*. It means that starting point as I1 is unimportant because the flow of thread occurs in a few seconds and therefore the output value O2 should be inspected quickly. If the output value O2 is not examined, it can be taken for granted that this system has erroneous parts. The test case generation which considers I1 becomes redundant work and is expensive.

How many test scenarios will be generated from the fragment of an activity diagram in Figure 1? If we apply the same example to the *IOAD*, we can derive two test cases.

In this concurrent case of Figure 2, two flows execute concurrently: One starting with O2 and the other starting with I1. However, O2 is an external output action by the system, which cannot be controlled by the tester. On the other hand, I1 is an external input and the system waits until the tester injects input. So we can assume that if the tester waits long enough, then O2 will eventually execute. After that, the tester can inject I1 to initiate the right-hand side flow of the fork-join structure in Figure 2. Therefore we can eliminate from the test cases derived the interleaved sequences of actions that start with I1 as shown in the table of Figure 2..



No	Test Scenarios
1	O2 → I2 → O3 → I1 → O1
2	
3	
4	O2 → I1 → O1 → I2 →O3
5	
6	
~~7~~	~~I1 → O2 → O1 → I2 → O3~~
~~8~~	~~I1 → O2 → I2 → O1 → O3~~
~~9~~	~~I1 → O2 → I2 → O3 → O1~~
~~10~~	~~I1 → O1 → O2 → I2 → O3~~

Fig 2. A sample IOAD model for the fork-join structure in Figure 1 and test scenarios for the IOAD model

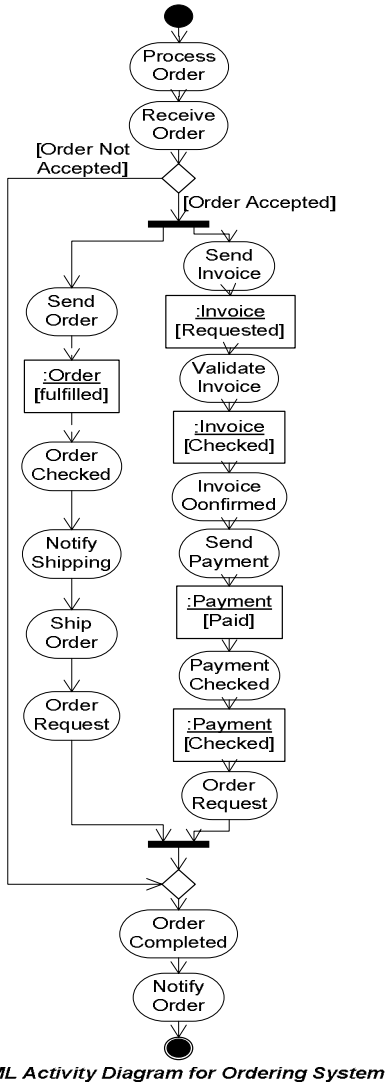


Fig 3. UML activity diagram for the order processing activity

4. An Application Example

Figure 3 shows an activity diagram for an order processing activity. We can convert this diagram into an *IOAD* as in Figure 4. Based on the two principles introduced in Section 3, activities can be changed into *accept event* and *send signal*. Activities *Process Order* and *Receive Order* are transformed into an accept event *Order Received*. The object *Invoice* and a performing activity *Send Invoice*, *Validate Order,* and *Invoice Confirmed* are converted to a send signal *Send Invoice* and to an accept event *Invoice Confirmed,* respectively. Likewise, the activities *Send Order, Order Checked, Notify Shipping, Ship Order, and Order Request* are altered to a *send signal* and an *accept event* notation.

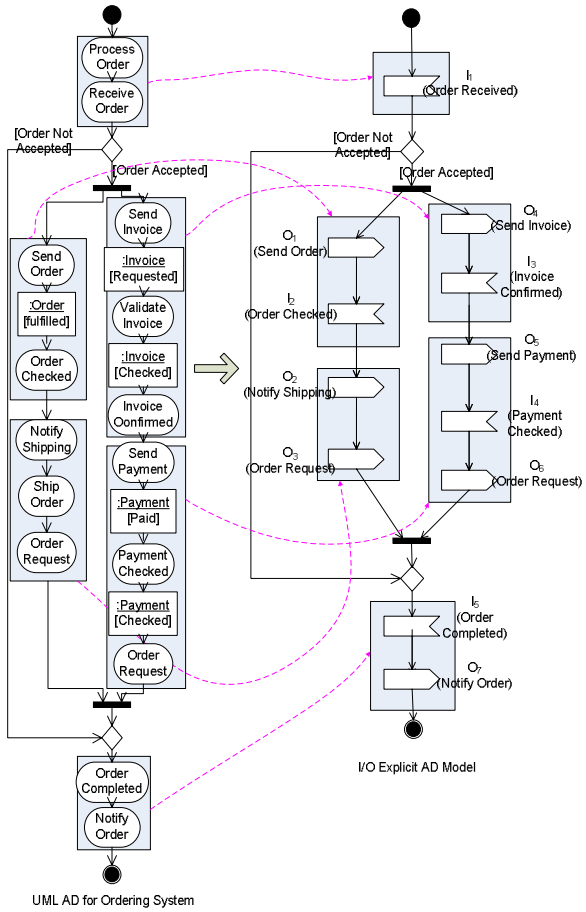


Fig 4. Building IOAD from the original activity diagram for the order processing activity

Figure 5 presents test cases generation process. We do not need to consider sequential cases. For example, we should consider the starting point from the fork as two cases (O1 and O4) because two starting points are concurrent cases. However, in our model, we regard testers' viewpoints as significant ones. Therefore, we just wait for the output event (O1 and O4, or vice versa). In addition to this, output event is followed by input event. For instance, if the tester input I3, then he can expect O5 output. Based on these procedures, we can build *Test Scenario Graph* as in Figure 5. As a coverage model, we adopt all-paths coverage criteria. There are only two elementary paths:

P1 = {I1, O1, O4, I3, O5, I2, O2, O3, I4, O6, I5, O7}
P2 = {I1, I5, O7}

With an additional path

P3 = {I1, O1, O4, I2, O2, O3, I3, O5, I4, O6, I5, O7}
P4 = {I1, O1, O4, I3, O5, I4, O6, I2, O2, O3, I5, O7}

are all simple paths. We can define elementary paths as independent paths because P3 and P4 are redundant cases when traversing nodes as indicated in P1.

Table 2 shows the result of test derivation.

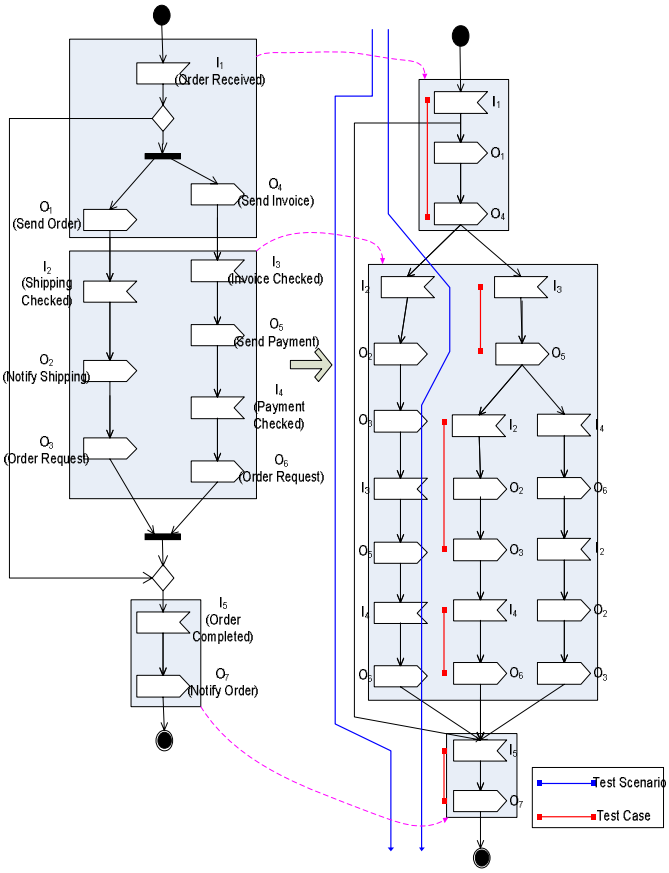


Fig 5. Test cases derivation

Table 2. Test derivation result

Item	*Test Case ID*	*Input Value*	*Expected Outcome Sequence*	*Test Summary*	*Notes*
1	OA1	I1	O1->O4	Initial section of test case	1) Order Accepted (abbr. OA)
		I3	O5	The first case of concurrent environment	
		I2	O2->O3	The second case of concurrent environment	
		I4	O6	The third case of concurrent environment	
		I5	O7	Last section of test case	
2	ONA1			The case which order is not approved	2) Order Not Accepted (abbr. ONA)

5. Conclusion and Future Work

In this paper, we presented a new method of generating test cases from UML activity diagrams. To derive test cases from UML activity diagrams, we introduced the *IOAD* model, which is subsequently converted to a directed graph for immediate extraction of test scenarios and test cases from the *IOAD*. This conversion is performed based on the *single stimulus principle* [7]. This method avoids the state explosion problem that can occur when trying to derive a set of test cases with thorough coverage for concurrent system. By reducing the number of test cases for concurrent system testing while keeping all practically useful test cases, we can save cost and time in software development without compromising quality of the developed system.

In the future, we plan to generalize our method so that it can accommodate various test coverage criteria within the same test derivation framework. We also plan to develop an automated tool for our method. There are mainly two parts to be automated. When constructing the *IOAD*, some notations of UML activity diagrams (data objects and implicit event triggers) need to be omitted. Next, it should be automated to build the directed graph and parse test cases from it. Then, we can generate test scenarios and test cases automatically. Many UML tools which support UML 2.0 [10] can generate XMI (Xml Metadata Interchange) file. Based on this XMI file, it would be easy to develop a tool which can generate *IOAD* and generate test cases.

6. References

[1] Wang. L., Yuan, J., Yu, X., , Hu, J., Li , X., Zheng G., "Generating Test Cases from UML Activity Diagram based on Gray-Box Method," National Natural Science Foundation of China, 2005.

[2] Li, H., Lam, C. P., "Using Anti-Ant-like Agents to Generate Test Threads from the UML Diagrams," TestCom 2005, LNCS 3502, pp. 69 – 80, 2005.

[3] Chandler, R., Lam, C. P., Li, H., "An Automated Approach to Generating Usage Scenarios from UML Activity Diagrams," Proceedings of the 12th Asia-Pacific Software Engineering Conference, 2005.

[4] Chen, M., Qiu, X., Li, X., "Automatic Test Case Generation for UML Activity Diagrams," National Natural Science Foundation of China, AST'06, 2006.

[5] Xu, D., Li, H., Lam, C.P., "Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams," Proceedings of the 12th Asia-Pacific Software Engineering Conference, 2005.

[6] Andriole, S. J., *Software Validation, Verification, Testing and Documentation*, Petrocelli Books, 1986.

[7] Kang, S., Shin, J., Kim, M., "Interoperability test suite derivation for communication protocols," Computer Networks Journal, Vol. 32, No. 3, 2000.

[8] Gao, J., Taso, H. S. J., Wu, Y., *Testing and Quality Assurance for Component-based Software*, Artech House Inc., 2003.

[10] UML 2.0 Superstructure, http://www.omg.org/cgi-bin/doc?formal/05-07-04