A Transformation-based Approach to Generating Scenario-oriented Test Cases from UML Activity Diagrams for Concurrent Applications

Chang-ai Sun
School of Computer and Information Technology
Beijing Jiaotong University, Beijing 100044 P.R. China
casun@bjtu.edu.cn

Abstract

Testing concurrent applications is difficult yet important. UML Activity Diagrams are widely used to model concurrent interactions among multiple objects. We present a transformation-based approach to generating scenario-oriented test cases for testing concurrent applications modeled by UML Activity Diagrams. The approach first transforms a UML activity diagram specification into an intermediate representation via a set of transformation rules. From the intermediate representation we then construct a set of test scenarios with respect to the given concurrence coverage criteria. Finally, we derive a set of test cases from the constructed test scenarios. The approach employs transformation to resolve the nonstructural problem with activity diagrams, and can generate test cases on demand to satisfy a given concurrence coverage criteria and hence the number of the resulting test cases is controllable. With the approach, testers can not only earlier schedule the software test process but also better allocate the test resource for testing concurrent applications.

1. Introduction

UML (Unified Modeling Language) [11] is the de-facto standard for modeling software systems under development. UML captures different aspects of the system and provides different UML diagrams to specify, construct, visualize and document artifacts of software-intensive systems. Based on these resulting design artifacts, the implementation of systems can be further automated by the code generation. It is important to design test cases earlier, since software engineering economics indicates that software testing should be executed at the earlier stage of the software development. In this situation, there is an increasing need of test techniques that derive tests from UML diagrams specifications.

UML Activity Diagrams are widely used to model business workflow and concurrent behavior of large-scale complex systems. The activity diagram describes how multiple objects collaborate to implement a set of specific operations or functional scenarios. The scenarios described by the activity diagram are often the business workflow in the implemented systems. This convinces us that test case generation from activity diagrams as the basis of functional testing, in particular for testing the concurrent behavior in the implemented system is more valuable and effective than those from other diagrams, such as Class Diagram [2,13], State Diagram [1, 3, 5, 7, 9, 15], Collaboration Diagram [10], and Use Case Diagrams [6].

In this paper, we present a transformation-based approach to effectively generating test cases from UML activity diagrams for testing concurrent applications. Concurrent behavior is nondeterministic; its testing is more difficult than the testing of common control flows or data flows. When activity diagrams are used to model concurrent applications under test, a challenge of generating test cases from activity diagrams relies in their non-structural properties. The approach is based on a set of transformation rules developed in our previous work [16] and in the meantime we extend them for loops. A typical system example is used to demonstrate the feasibility of the approach.

The rest of this paper is organized as follows. Section 2 provides a motivating example of activity diagram which is used through out the paper. Section 3 introduces a set of transformation rules. Section 4 presents a transformation-based approach to generating scenario-oriented test cases from activity diagrams. Section 5 discusses related work. Section 6 concludes the paper and points out future work.

2. A Motivating Example of UML Activity Diagrams

The basic elements in a UML Activity Diagram are *activities* and *transitions*. An *activity* noted as a node is a state of doing something and can be further classified into different types, such as *Start Activity*, *End Activity*, *Branch Activity*, *Merge Activity*, *Fork*

0730-3157/08 $25.00 © 2008 IEEE
DOI 10.1109/COMPSAC.2008.74

IEEE computer society

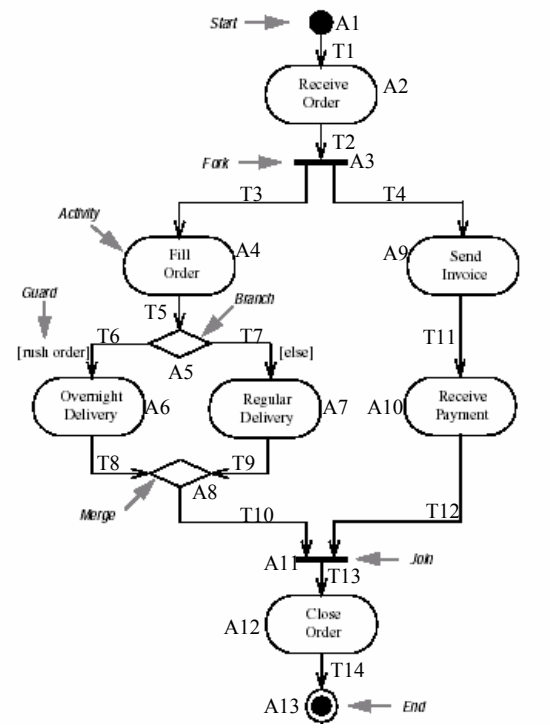

Figure 1. An example of Activity Diagrams

Activity, *Join Activity* and so on. A *transition* noted as a directed line acts as the connection between activities and can be further classified into *control flow*, *message flow* and *object flow*. Activity Diagram can be used to describe the complex sequence of activities, with support for both *conditional* and *parallel* behavior. *Conditional* behavior is delineated by a *branch* and a *merge*, and *parallel* behavior is indicated by a *fork* and a *join*. A *branch* has a single incoming transition and several guarded outgoing transitions. *Parallel* processes during the *fork* and *join* can choose the arbitrary order to execute.

An example of UML activity diagrams [4] is illustrated in Figure 1, which describes the business workflow of an order processing system. There is a conditional behavior and a parallel behavior, and two guarded branches with the *branch* activity. From the business point of view, we can derive the complete functional scenarios from this activity diagram, and each is a path from the *Start* to *End* activity.

3. Transformation Rules

UML Activity Diagrams provide the *fork* and *join* mechanism for modeling concurrent behavior (i.e. interleaving parallel processes) in a complex business workflow. It is difficult to generate test scenarios from such a non-structural diagram in particular when there are complex nesting of concurrent behavior. To alleviate this we have developed a set of transformation rules to translate the activity diagram specification into an intermediate representation. For each type of activity, one rule is used to transform it into an equivalent tree. As an illustration, we present transformation rules for the *fork* and *join* activities. The interested readers refer to [16] for the whole view.

Definition 1(Activity Diagram) A UML Activity Diagram AD is a tuple $< A,T,C,L >$, where A is the collection of activities; $T : A \, x \, C \rightarrow A$, where C is the constraint conditions that must be satisfied when the transition T happens; L is a label function which assigns a unique label for each activity and transition.

Definition 2 (Extended AND_OR Tree) An extended AND_OR tree ET is a tuple $< N, E, \psi >$, where $N = \overline{A} + \{BOR, MOR, FAND, JAND\}$, $\overline{A}$ is the mapped nodes from activities A in AD; *BOR* and *MOR* are extended *OR* nodes designed for the *branch* and *merge* activity respectively, while *FAND* and *JAND* are extended *AND* nodes designed for the *fork* and *join* activity respectively; E is a tuple $<N_1, N_2>$ where $N_1 \in N \wedge N_2 \in N$; ψ is a label function which assigns a unique label for each node and edge.

Transformation rule for *Fork* activities For the *fork* activity a of activity diagram AD with multiple out transitions $t_1 \in T \wedge \ldots \wedge t_m \in T$ where $m > 1 \wedge t_i$ is $(a, c_i) \rightarrow a_i \wedge 1 \le i \le m \wedge c_i \in C \wedge a_i \in A$, we create a node n in ET to refer to a, add a logic node *FAND* in ET, connect n and *FAND* by an edge labeled with NULL; in the meantime, the out-transition edges of *FAND* are set as $FAND \xrightarrow{e_1} n_1, \ldots, FAND \xrightarrow{e_m} n_m$, where $e_1, \ldots, e_m$ are the mapped edges of transitions $t_1, \ldots, t_m$ in T of AD; $n_1, \ldots, n_m$ are the mapped nodes of activities $a_1, \ldots, a_m$ in A of AD. Figure 2 shows the transformation of the *fork* activity illustrated in Figure 1. Note that the transformation of the *branch* activity is similar except that *FAND* is replaced with *BOR* in Figure 2.

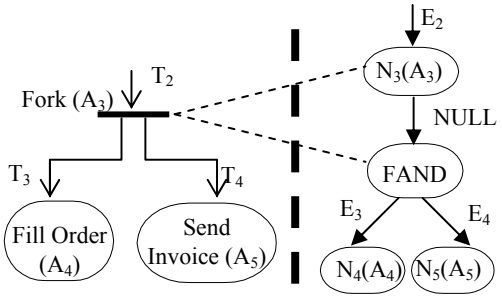

Figure 2. The transformation of the *fork* activity

Transformation Rule for *Join* Activities For the *join* activity a of activity diagram AD with multiple in-transitions $t_1 \in T \wedge \ldots \wedge t_m \in T$ where $m > 1 \wedge t_i$ is $(a_i, c_i) \rightarrow a \wedge 1 \le i \le m \wedge c_i \in C \wedge a_i \in A$, we create a node n in ET to refer to a, add a logic node *JAND* in *ET*, connect *JAND* and n by an edge labeled with NULL; in the meantime, the in-transition edges of JAND are set as $n_1 \xrightarrow{e_1} JAND, \ldots, n_n \xrightarrow{e_m} JAND$, where $e_1, \ldots, e_m$ are the mapped edges of transitions $t_1, \ldots, t_m$ in T of AD; $n_1, \ldots, n_m$ are the mapped nodes of activities $a_1, \ldots, a_m$ in A of AD. Figure 3 illustrates the transformation of the *join* activity illustrated in Figure 1. Note that the transformation of *merge* activity is similar except that *JAND* is replaced with *MOR* in Figure 3.

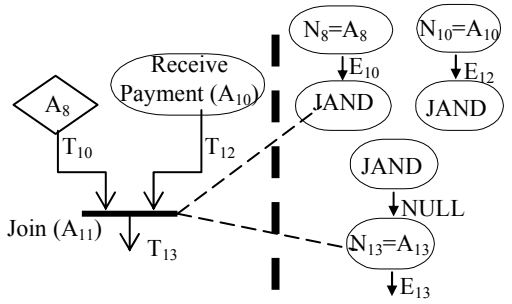

Figure 3. The transformation of the *join* activity

4.Scenario-oriented Test Case Generation from UML Activity Diagrams

Generating test cases from UML activity diagrams is a kind of model-driven testing technique which is more challenging than black-box testing or white–box testing because the incomplete information is provided in the design model and there are multi-design aspects which are usually separate. We assume that enough information has been provided in the activity diagram specifications before our approach is applied and that there is not inconsistency in the UML Activity Diagrams under processing before our approach is applied. These two assumptions can be alleviated since UML-based software development process puts more emphases on modeling the system's structure and behavior in order to support the automatic code generation. This means that enough detailed information is provided in the design model and some tool executes the inconsistency checking before the execution of testing. This greatly enhances the feasibility of our approach in practice.

To deal with the nonstructural property of activity diagrams due to the *fork* and *join* activity, we propose a transformation-based approach to generating test cases for testing concurrent applications. The approach first transforms activity diagram specifications into an intermediate representation, from which we can develop algorithms to derive a set of test scenarios. For each test scenario, test cases are derived through selecting the corresponding decisions in the test scenario path. Thus, the intermediate representation, in some sense, is a kind of general test specification language on which we can develop algorithms to satisfy different concurrent coverage criteria. We use the example discussed in Section 2 to demonstrate the proposed approach, and the discussion concentrates on testing concurrent applications modeled by UML activity diagrams.

4.1 Transformation

When UML activity diagram is used to model business workflow, the resulting artifacts are *activity diagram specifications*. We use two separate tables to store the activities and transitions in the activity diagram specifications. Table 1 illustrates the activities extracted from the activity diagram specification example in Section 2. A tuple <*I*, *T*, *P*, *IT*, *OT*> is used to specify each activity where

(1) *I* is a unique identity number of an activity.
(2) *T* identifies the activity type, such as *Start*, *End*, *Branch*, *Merge*, *Fork*, *Join* and so on.
(3) *P* refers to other detailed attributes of activities.
(4) *IT* records the incoming transitions. Both *join* and *merge* have more than one incoming transitions.
(5) *OT* records the outing transitions. Both *branch* and *fork* have more than one outing transitions.

Table 1. Extracted activities from the Activity Diagram Specification

I	*T*	*P*	*IT*	*OT*
A1	Start	N/A	N/A	T1
A2	Common	Receive Order	T1	T2
A3	Fork	Concurrency Begin	T2	T3, T4
A4	Common	Fill Order	T3	T5
A5	Branch	Branch Begin	T5	T6, T7
A6	Common	Overnight Delivery	T6	T8
A7	Common	Regular Delivery	T7	T9
A8	Merge	Branch End	T8,T9	T10
A9	Common	Send Invoice	T4	T11
A10	Common	Receive Payment	T11	T12
A11	Join	Concurrency End	T10, T12	T13
A12	Common	Close Order	T13	T14
A13	End	N/A	T14	N/A

For each transition we assign its ID and record its transition type, pre-activities, post-activities and other properties, such as the guard conditions. With the transformation rules, we can transform the extracted activities and transitions into an intermediate representation, where the extended AND nodes and OR nodes are introduced, namely *BOR*, *MOR*, *FAND* and *JAND*. These nodes are equipped with special semantics, which helps to generate test scenarios in the subsequent steps.

We also use the table to represent the transformed intermediate representation. For each transformed node and edge, we maintain their relationship with original activities and transitions in the activity diagram specifications. Table 2 illustrates the transformed intermediate representation for the activity diagram specification example in Section 2.

Table 2. Transformed Nodes in the Intermediate Representation

Node	*Type*	*Pre-Edge*	*Post-Edge*	*Original Activities*
N1	START	Φ	E1	A1
N2	Normal	E1	E2	A2
N3	NULL	E2	E3	A3
N4	FAND	E3	E4, E5	A3
N5	Normal	E4	E6	A4
N6	Normal	E5	E10	A9
N7	NULL	E6	E7	A5
N8	BOR	E7	E8,E9	A5
N9	Normal	E8	E11	A6
N10	Normal	E9	E12	A7
N11	Normal	E10	E15	A10
N12	MOR	E11, E12	E13	A8
N13	NULL	E13	E14	A8
N14	JAND	E14,E15	E16	A11
N15	NULL	E16	E17	A11
N16	Normal	E17	E18	A12
N17	END	E18	Φ	A13

4.2 Deriving Test Scenarios

The intermediate representation is a set of trees. It is more convenient to drive test scenarios from a normalized tree than from the common one. Therefore, we propose the binary tree to represent the test specification on which we can further develop algorithms to generate test scenarios on demand.

An activity diagram specification may have multiple parallel behaviors or conditional behaviors. As illustrated in Figure 4, a *fork* activity with more than two out-transitions will result in more than two branches in the intermediate representation. In order to keep to the binary tree, we need to add extra *FAND* nodes, illustrated in Figure 5. For those *join* activities that may have more than one in-transition, the constructed binary tree from the intermediate

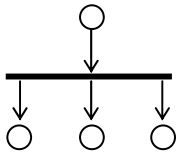

Figure 4. The activities with more than two out-transitions

representation is a graph rather than a set of trees. To avoid the occurrence of graph during the construction of binary trees, we stop the construction process when the current leaf is *MOR* node, *JAND* node or *End* node. Thus, the resulting test specification is a forest of binary trees, and for each binary tree, its root is one of *MOR* node, *JAND* node or *Start* node and it leaf is one of *MOR* node, *JAND* node or *End* node. We call such a test specification as the *Binary Extended AND_OR Tree* (briefly *BET*), since the extended *AND* and *OR* nodes are contained.

We define the semantics relationship among left child *LChild*, right child *RChild* and parent node *Father* as follows: *Father* must execute just before the *LChild* and the *RChild*; *LChild* can execute in parallel with *RChild* without the specific order. According to the convention, only *BOR* nodes and *FAND* nodes have the *RChild*s. We propose Algorithm 1 to construct a set of *BETs* from the intermediate representation. The algorithm first looks for a *Start*/ *MOR* /*JAND* node and uses it as the tree root. Then a depth-first binary tree traversal procedure continues until all its leaves of the binary tree under construction are *MOR*/*JAND*/*END* nodes. The algorithm generates a set of trees. By applying Algorithm 1 to the transformed intermediate representation, we obtain a set of *BETs* as illustrated in Figure 6.

A path from the *Start Activity* to the *End Activity* is factually a functional scenario. Concurrent behavior is very common in activity diagram specifications. Assume that there are n parallel processes $p_1, p_2, ..., p_n$ between a pair of *fork* and *join* activity, and P_i consists of m_i sequential activities, denoted as $a_{i,1} \to a_{i,2} \to ... \to a_{i,m_i}$ where $1 \le i \le n$ and $m_i \ge 1$, then

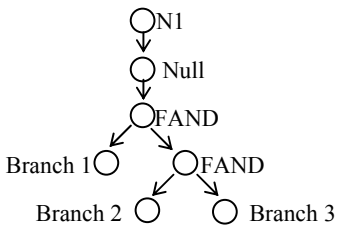

Figure 5. The binary tree for multiple parallel behaviors

Algorithm 1. Constructing *BETs* from the intermediate representation

INPUT: ***NodeT*** is a node table and ***EdgeT*** is an edge table.
OUTPUT: ***TestSpec*** is a set of BETs
PROCEDURE
1. Initialize the flag of all nodes in *NodeT* to UNPROCESSED, initialize the *TestSepc*
2. Look up and return a root node *r* from *NodeT* where *r.PreEdge=Φ*
3. If all not nodes in *NodeT* are processed, then repeat steps 4~7; Otherwise go to step 8
4. Set the flag of node *r* as PROCESSED.
5. Return a tree *bet* from the remaining unprocessed nodes to satisfy that the root of *bet* is *r* and all its leaves are one of *MAND, JAND*, and *END* nodes. If a node *n* is included into bet, then set the flag of *n* as PROCESSED.
6. Add the *bet* to ***TestSpec***
7. Look up and return a root node *r* from the *NodeT* where *r.type=MOR* or *r.type=JAND,* goto Step 3
8. Return the ***TestSpec***

there is a maximum of $(\sum_{i=1}^{n} m_i)! \Big/ \prod_{i=1}^{n} m_i!$ possible test scenarios between the *fork* and *join* activity. For a large-scale system, functional scenarios derived from activity diagram specifications may be numerous. It is impossible to test all these function scenarios due to the limited testing resource in practice. Therefore, we propose the following concurrence coverage criteria that decide to what extent the combination of concurrent elements should be tested.

(1) *Weak concurrency coverage* Test scenarios are derived to cover only one feasible sequence of parallel processes between a pair of *fork* and *join* activity, without considering the interleaving of activities between parallel processes.

(2) *Moderate concurrency coverage* Test scenarios are derived to cover all the feasible sequences of parallel processes between a pair of *fork* and *join* activity, without considering the interleaving of activities between parallel processes.

(3) *Strong concurrency coverage* Test scenarios are derived to cover all feasible sequences of activities and parallel processes between a pair of *fork* and *join* activity.

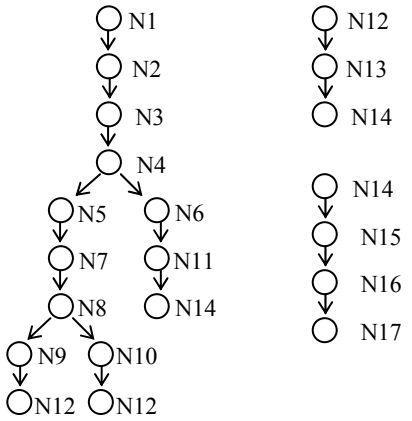

Figure 6. The constructed BETs from the intermediate representation

All these concurrence coverage criteria require the derived test scenarios to cover each parallel process at least once. Both *weak concurrence coverage* and *moderate concurrence coverage* test the activities and transitions within a parallel process in a sequential way; while *strong concurrency coverage* considers the crossing of activities and transitions from parallel processes.

Algorithms for deriving test scenarios from the test specifications (namely the *BETs)* can be developed to satisfy the above concurrence coverage criteria. As an illustration, Algorithm 2 schematically derives the weak concurrency coverage test scenarios from the *BETs*. The algorithm consists of three passes. During the first pass, *GenerateSequence* recursively generates the initial node sequences from the *TestSpec* in a depth-first traversal way; during the second one, *Concatenate* concatenates all initial node sequences into the complete node sequences by repeating the substitute operations; during the third one, *GenerateValidTestScenarios* merges the parallel complete node sequences as the final test scenario based on the weak concurrency coverage criteria. The generated test scenarios are represented by a sequence of nodes. For simplicity, edges are omitted since they can be figured out from the transformed nodes and edges table. These test scenarios can be further transformed to the sequence of activities and transitions, since our approach retains the mapping between transformed nodes and edges in the intermediate representation and original activities and transitions in activity diagram specifications.

We now execute Algorithm 2 on the *BETs* as illustrated in Figure 6. Tables 3 and 4 illustrate the collection of node sequences after the procedures *GenerateSequence* and *Concatenation* in Algorithm 2, respectively. Table 5 shows a set of the derived test scenarios which satisfy weak concurrency coverage criteria. The *Test Scenarios (N-E-N)* column represents test scenarios in the form of "node to edge to node" which is from the intermediate representation, while the *Test Scenarios (A-T-A)* column represents test scenarios in the form of "activity to transition to activity" which is from the original activity diagram specifications.

Algorithm 2. Deriving test scenarios from *BETs*

INPUT: ***TestSpec*** is a collection of *BETs*
OUTPUT: ***TestScen*** is a collection of test scenarios that are composed of the sequence of nodes.
PROCEDURE
1.Initialize ***INSs*** and ***CNSs*** which are used to record the collections of initial node sequences and complete node sequences, respectively.
2.Repeat the steps 3-5 until all bets in ***TestSpec*** are processed
3. Get a bet ***currentBet*** from ***TestSpec, TestSpec=TestSpec \ {currentBet}***
4. ***currentNode= currentBet.Root***
5.***NSs***= ***INSs***∪*GenerateSequence(****currentNode****)*
6. ***CNSs***=*Concatenate (****INSs****)*
7.***TestScen***= *GenerateValidTestScenarios*(***CNSs***)
8. Return ***TestScen***

Table 3. Immediate Node Sequences after the first pass of Algorithm 2 on *BETs* in Figure 6

No	Immediate Node Sequences
1	N1 → N2 → N3 → N4 → N5 → N7 → N8 → N9 → N12
2	N1 → N2 → N3 → N4 → N5 → N7 → N8 → N10 → N12
3	N1 → N2 → N3 → N4 → N6 → N11 → N14
4	N12 → N13 → N14
5	N14 → N15 → N16 → N17

Table 4. Complete Node Sequences after the second pass of Algorithm 2 on *BETs* in Figure 6

No	Complete Node Sequences
1	N1 → N2 → N3 → N4 → N5 → N7 → N8 → N9 → N12 → N13 → N14 → N15 → N16 → N17
2	N1 → N2 → N3 → N4 → N5 → N7 → N8 → N10 → N12 → N13 → N14 → N15 → N16 → N17
3	N1 → N2 → N3 → N4 → N6 → N11 → N14 → N15 → N16 → N17

Table 5. Weak Concurrency Coverage Test Scenarios Generated from *BETs* in Figure 6

No	Test Scenarios (N-E-N)	Test Scenarios (A-T-A)
1.	N1-N2-N3-N4-N5-N7-N8-N9-N12-N13-N6-N11-N14-N15-N16-N17	A1-A2-A3-A4-A5-A6-A8-A9-A10-A11-A12-A13
2.	N1-N2-N3-N4-N5-N7-N8-N10-N12-N13-N6-N11-N14-N15-N16-N17	A1-A2-A3-A4-A5-A7-A8-A9-A10-A11-A12-A13

4.3 Test Case Generation

Category Partition Method (CPM) [12] is a kind of method for effectively creating functional test suites for the system specifications. In CPM, a *category* is a major property or characteristic of a parameter or environment, *choices* are distinct possible values within a category, a *test frame* is a set of choices. We leverage the terms used by CPM, while adapt them to the context of test scenarios. Each complete and feasible *test scenario* is a complete *test frame*. A *test case* with respect to a *test scenario* corresponds to a set of *choices* whose values can result in the execution of the *test scenario*. Hence, our approach identifies the *categories* and *choices* by processing conditions in the branch activities (decision guards), and identifies the dependency relationships between different choices by judging whether these choices occur in the same scenario paths. Finally, we generate test cases by filling up the values for those guards and inputs required in each activity along the scenario path.

For the test scenarios in Table 5, we construct two feasible test cases as follows:A1->A2->A3->A4->A5 {Rush Order/Order Type} ->A7 ->A8- >A9-> A10-> A11->A12->A13, and A1->A2->A3->A4-> A5 {Else /Order Type}->A7->A8-> A9->A10->A11->A12->A13. These test cases should contain all necessary inputs required by the activities and guards in transitions along the test scenario path. For simplicity, the inputs for each activity are omitted.

In some situation, activities in the business workflow repeat until the specific conditions are satisfied. As a necessary extension, we provide a scheme for processing loops in the activity diagrams, including the identification, the transformation rule and the generation of test scenarios for loops.

Definition 3 (Post-activities of an activity) Given an activity *a* in the UML activity diagram *ad*. We define the *post-activities* of the activity *a* as follows:

$Post\text{-}activities(a) = a^1 \cup a^2 \cup ... \cup a^n$ where $a^1 = post-activities(out-transitions(a))$, …, $a^n = post-activities(out-transitions(a^{n-1}))$ and *out-transitions* (*a*) is a set of out-transitions of the activity *a*, *post-activities* (*t*) is a set of post-activities of transition *t*.

There are loops in the activity diagram only when

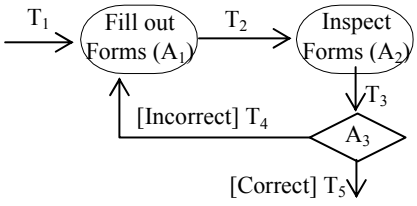

Figure 7. An Example of the loop-causing common activity

there exists an activity that is subsumed in its *post-activities*. Ten transformation rules are developed to process various type of activities [16]. It is omitted how to transform a common activity with two or more in-transitions, which is illustrated as the common activity *Fill out Forms* A_1 in Figure 7.

To solve this problem, we propose to convert these common activities with two or more out-transitions into the combination of one *merge* activity $A_1^{'}$ and original common activity, which is illustrated in Figure 8. Comparing the activity diagram before transformation and the one after transformation, one *fake merge* activity $A_1^{'}$ and one *fake transition* T' are introduced into the latter. They do not cause extra elements from the perspective of test scenarios. However, one problem with the transformed specification is the mismatch between the fake *merge* activity and *branch* activity; the fake *merge* activity precedes the *branch* activity. This will result in the failure of our algorithm for deriving test scenarios from the test specification.

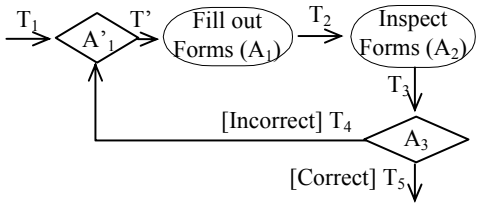

Figure 8. The transformation of the loop-causing common activity

The execution time of the loop body in the activity diagram specification can only be decided at run-time. Each conditional branch of the loops should be covered at least once from the testing point of view. Keeping this in mind, the transformation from Figure 8 to Figure 9 solves the problem. The resulting test scenarios from Figure 7 and Figure 9 are $\{ \xrightarrow{T_1}$ Fill Out Forms $(A_1) \xrightarrow{T_2}$ Inspect Forms $(A_2) \xrightarrow{T_3} A_3 \xrightarrow{[Correct]T_5}$; $\xrightarrow{T_1}$ Fill Out Forms $(A_1) \xrightarrow{T_2}$ Inspect Forms $(A_2) \xrightarrow{T_3} A_3 \xrightarrow{[Incorrect]T_4}$ Fill Out Forms $(A_1) \xrightarrow{T_2}$ Inspect Forms $(A_2) \xrightarrow{T_3} A_3 \xrightarrow{[Correct]T_5} \}$ and $\{ \xrightarrow{T_1}$ Fill Out Forms $(A_1) \xrightarrow{T_2}$ Inspect Forms $(A_2) \xrightarrow{T_3} A_3 \xrightarrow{[Correct]T_5} A_1^{'} \xrightarrow{[Correct]T_5}$; $\xrightarrow{T_1}$ Fill Out Forms $(A_1) \xrightarrow{T_2}$ Inspect Forms $(A_2) \xrightarrow{T_3} A_3 \xrightarrow{[Incorrect]T_4}$ Fill Out Forms $(A_1) \xrightarrow{T_2}$ Inspect Forms $(A_2) \xrightarrow{T'} A_1^{'} \xrightarrow{[Correct]T_5} \}$, respectively. Since $A_1^{'}$ and T' are a fake activity and transition, respectively, test scenarios derived from Figure 9 (after the transformation) equal to those derived from Figure 7 (before the transformation) under the consumption that each conditional branch is executed once.

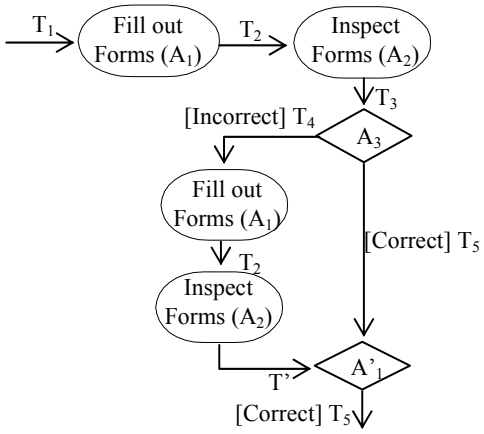

Figure 9. The normalized transformation of the loop-causing common activity

5. Related Work

Many test techniques are developed to generate tests from different UML diagram specifications [1, 3, 5, 6, 7, 9, 10, 15]. We introduce most related work on test case generation from UML Activity Diagrams.

In our previous work [16], we developed a three-layer framework for automated test case generation from UML activity diagram specifications. An important contribution is a set of transformation rules for each type of activities, which provides a sound and convenient basis for the development of test case generation algorithms. We developed a tool, TCaseUML which extracts the UML activity diagram specifications from Rational Rose and generates test cases in terms of each activity. However, it is not well investigated how test cases can be effectively generated for functional scenarios from the transformed test outline model. The work presented in this paper employs the transformation rules in [24], and in the meantime supplements the previous work by extending the transformation rules for processing the loops in the activity diagrams.

Wang et al. [14] propose a gray box-based approach to generating test scenarios from the activity diagrams and generating test cases by extracting the information from each test scenario. Test scenarios are generated in terms of basic paths and their algorithm for test scenario generation is developed based on the Petri net model. Since there are many parallel and conditional behaviors in activity diagrams, it is difficult to directly define the

basic path from the original activity diagram. The algorithm presented does not touch the concurrency issue, an essential component in UML activity diagrams. Our approach generates test cases from UML activity diagram specifications via transformation and provides sound testing for concurrent applications.

Kim et al. [8] propose a transformation-based method to generate test cases from UML activity diagrams. Their 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. The work is similar to our approach in that both methods employ the transformation, while they are different in that our transformation rules are developed based on generic type of activities instead of the instance I/O flows. In our approach test cases generated are scenario oriented which are controllable and satisfy the special concurrence coverage criteria, while their approach is based on the single stimulus principle which is used to deduce the number of test cases.

6. Conclusion and Future Work

We presented a transformation-based approach to generating scenario-oriented test cases from UML activity diagram specifications, with an emphasis on testing concurrent applications modeled using UML activity diagrams. The method employs and extends a set of transformation rules to convert an UML activity diagram specification into a well-formed intermediate representation and hence resolves non-structural properties with UML Activity Diagrams. With the approach, testers can start test design in the design stage instead of in the programming stage. Additionally, test resources are often limited in practice, which consequently requires the complexity and number of tests should be controllable. The approach supports this by generating different sets of test scenarios to satisfy different concurrence coverage criteria. Therefore, the proposed approach is in particular useful for effectively testing concurrent applications.

We are going to extend the tool TCaseUML with the extended transformation rules and test scenario generation algorithms, evaluate the approach in real-life concurrent applications such as multi-threaded Java programs modeled using UML activity diagrams, and measure the fault detection capability of concurrence coverage criteria and their costs.

Acknowledgement

The author thanks Professor T.Y. Chen and Professor Dieter Hammer for invaluable comments. The research is partially supported by the Science and Technology Foundation of Beijing Jiaotong University (Grant No. 2007RC099) and the Development Grant for Computer Application Technology Subject jointly Sponsored by Beijing Municipal Commission of Education and Beijing Jiaotong University (Grant No. XK100040519).

References

[1] L. C. Briand, J. Cui, Y. Laboche, Towards automated support for deriving test data from UML Statecharts, Proceedings of UML 2003, LNCS 2863, pp249-264.

[2] H.Y. Chen, An approach for OO cluster-level tests based on UML, Proceedings of the SMC2003, IEEE Computer Society, 2003, pp1064-1068.

[3] P. Chevalley, P.T. Fosse, Automated Generation of Statistical Test Cases from UML State Diagrams, Proceedings of COMPSAC 2001, pp205-214.

[4] M. Fowler, K. Scott, Activity Diagrams, http://www.sts.tu-harburg.de/projects/UML/Activity Diagrams.pdf, pp151-164.

[5] J. Hartmann, C. Imoberdof , M. Meisenger, UML-Based Integration Testing, Proceedings of ISSTA 2000, pp60-70.

[6] IBM Center for Software Engineering, Use Case Based Testing, http://www.research.ibm.com/softeng/testing /ucbt.htm

[7] Y.G. Kim, H.S. Hong, S.M. Cho, D.H. Bae, S.D.Cha, Test case generation from UML state diagrams, IEEE Software, 1999, 46(4):187-192.

[8] H. Kim, S. Kang, J. Baik, I. Ko, Test Cases Generation from UML Activity Diagrams, Proceedings of SNPD 2007, pp556 – 561.

[9] A.J. Offutt, A. Abdurazik, Generating tests from UML specifications, Proceedings of UML'99, pp416-429.

[10] A.J. Offutt, A. Abdurazik, Using UML Collaboration Diagrams for Static Checking and Test Generation, Proceedings of UML'00, pp383-395.

[11] Object Management Group, UML Specification (v1.5), http://www.omg.org/uml, March 2003

[12] T. J. Ostrand, M.J. Blacer, The category-partition method for specifying and generating functional tests, Communications of the ACM, 1988, 31(6):676-686.

[13] M. Scheetz, A. Mayrhauser, R. France, et al. Generating Test Cases from an OO Model with an AI Planning System, Proceedings of ISSRE99, pp250-259.

[14] L. Wang, J. Yuan, X. Yu, et al., Generating Test Cases from UML Activity Diagram based on Gray-Box Method, Proceedings of APSEC2004, 2004, pp284-291.

[15] J. Yan, J. Wang, H.W. Chen, Deriving Software Statistical Testing Model from UML Model, Proceedings of QSIC2003, 2003, pp343-351.

[16] M. Zhang, C. Liu, C.A. Sun, Automated Test Case Generation Based on UML Activity Diagram Model, Journal of Beijing University of Aeronautics and Astronautics, 2001, 27(4):433-437.