View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Nottingham ePrints

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Sun, Chang-ai and Zhao, Yan and Pan, Lin and He, Xiao and Towey, Dave (2016) A transformation-based approach to testing concurrent programs using UML activity diagrams. Software: Practice and Experience, 46 (4). pp. 551-576. ISSN 0038-0644

Access from the University of Nottingham repository:
http://eprints.nottingham.ac.uk/51790/1/SPEpaper2014.CA.20141215.pdf

Copyright and reuse:

The Nottingham ePrints service makes this work by researchers of the University of Nottingham available open access under the following conditions.

This article is made available under the University of Nottingham End User licence and may be reused according to the conditions of the licence. For more details see: http://eprints.nottingham.ac.uk/end_user_agreement.pdf

A note on versions:

The version presented here may differ from the published version or from the version of record. If you wish to cite this item you are advised to consult the publisher's version. Please see the repository url above for details on accessing the published version and note that access may require a subscription.

For more information, please contact eprints@nottingham.ac.uk

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **00**:1–26
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

A Transformation-based Approach to Testing Concurrent Programs using UML Activity Diagrams

Chang-ai Sun[1]*, Yan Zhao[1], Lin Pan[1], Xiao He[1], and Dave Towey[2]

[1]*School of Computer and Communication Engineering, University of Science and Technology Beijing, China*
[2]*School of Computer Science, The University of Nottingham Ningbo China, China*

SUMMARY

UML activity diagrams are widely used to model concurrent interaction among multiple objects. In this paper, we propose a transformation-based approach to generating scenario-oriented test cases for applications modeled by UML activity diagrams. Using a set of transformation rules, the proposed approach first transforms a UML activity diagram specification into an intermediate representation, from which it then constructs test scenarios with respect to the given concurrency coverage criteria. The approach then finally derives a set of test cases for the constructed test scenarios. The approach resolves the difficulties associated with fork and join concurrency in the UML activity diagram, and enables control over the number of the resulting test cases. We further implemented a tool to automate the proposed approach, and studied its feasibility and effectiveness using a case study. Experimental results show that the approach can generate test cases on demand to satisfy a given concurrency coverage criterion, and can detect up to 76.5% of seeded faults when a weak coverage criterion is used. With the approach, testers can not only schedule the software test process earlier, but can also better allocate the testing resources for testing concurrent applications. Copyright © 2014 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Software Testing; Scenario-oriented Testing; Concurrent Programs; UML Activity Diagrams

1. INTRODUCTION

The Unified Modeling Language (UML) [28] is the de-facto standard for modeling software systems. UML captures different aspects of the system and provides different diagrams to specify, construct, visualize and document artifacts of software-intensive systems. UML 2.0 [8] defines 13 diagrams types, which can be classified into structure, behavior and interaction categories. Based on these, implementation of systems can be further automated using code generation. Since software engineering economics indicates that software testing should be performed in earlier stages of the software development process, it is important to design test cases as early as possible, leading to an increasing demand for testing techniques which derive tests directly from UML diagram specifications.

In recent years, much research has looked at test case generation from various UML diagrams, such as Class Diagrams [5][32], State Diagrams [3][7][13][17][19][26][36][40],

*Correspondence to: School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China. E-mail:casun@ustb.edu.cn
†A preliminary version of this paper was presented at the 32nd Annual IEEE International Computer Software and Application Conference (COMPSAC 2008) [33].

Sequence/Collaboration Diagrams [27][38], Use Case Diagramss [14], Activity Diagrams [33][34], and the combination of Use Cases and State charts [30].

UML Activity Diagrams (UADs) are widely used to model business workflow and the concurrent behavior of large-scale complex systems, and therefore many domains and applications are most easily rendered by such flow-based descriptions. They describe how multiple objects collaborate to implement a set of specific operations or functional scenarios, and the scenarios described correspond to the business workflow in the implemented systems. Because of this, we believe that test cases generated from activity diagrams as the basis of functional testing, especially for testing concurrent behavior, are both appropriate and well-suited for validating the correctness of the entire system [33].

When generating tests from UADs, we first need to construct a set of test scenarios, which represent a sequence of performed operations in a software system. When the system to be tested is complex, the number of test scenarios may be huge, and therefore one challenging task is how to derive a comprehensive test scenario suite from UADs. Clearly, automatic test scenario generation would be particularly desirable, but automatic UAD-based test case generation faces the following difficulties [33][34]:

(1) Some characteristics of the UADs may cause challenges for the generation of test scenarios. In particular, the fork and join activities are commonly used to model concurrent applications, but they have some special semantics which are different to the common branch structures, including that only one branch will be executed when its corresponding condition holds, but all parallel activities between a fork and its corresponding join must be executed.
(2) Exhaustive coverage of UAD concurrency and branch features could lead to a huge number of test scenarios, not all of which could be tested, because some infeasible test scenarios correspond to unreachable paths.

In our previous work [42], we developed a three-layer framework for generating test cases based on specifications in the form of UADs. An important contribution of this work was a set of transformation rules. Since UADs often contain elements associated with fork and join branches and concurrent flows, the approach first converts the UAD into a set of Extended AND-OR Binary Trees (EBTs) — a standardized intermediate structure. However, how to handle loops in UADs and how to generate test scenarios from the transformed specification (i.e. EBTs) was not addressed. In [34], we proposed a recursive algorithm to directly generate test scenarios from UADs with respect to a basic path coverage criterion, and also developed a tool to automate the algorithm. Note that this algorithm was not based on the transformed specification.

In this paper, we propose a transformation-based scenario-oriented approach to testing programs whose behavior is described using UADs. All fork and join elements are eliminated during the transformation, with concurrent structures represented as binary trees to avoid the path explosion problem during test case generation. Next, the approach generates test scenarios from the EBTs according to a coverage criterion for concurrent flows. In this paper, we present a transformation-based scenario-oriented testing framework, develop a prototype tool to automate this framework, and present a case study to examine the applicability and effectiveness of the proposed approach. The main contributions of this paper, together with its preliminary version [33], are fourfold:

(i) We propose a transformation-based scenario-oriented testing framework and develop transformation rules for converting a UAD to a standardized intermediate structure;
(ii) We propose concurrency coverage criteria for testing concurrent behavior, and develop a set of algorithms for generating test scenarios with respect to a specific criterion;
(iii) We develop a tool to automate the proposed framework and to interact with UAD-supported tools; and
(iv) We conduct a case study to validate the feasibility and effectiveness of the proposed approach.

The rest of the paper is organized as follows: Section 2 introduces some of the concepts of UADs, model-driven testing, and mutation analysis; Section 3 explains the proposed transformation-based testing approach; Section 4 presents the tool developed to implement the proposed approach; Section

5 describes a case study to validate the feasibility and effectiveness of the proposed approach; Section 6 discusses some related work; and Section 7 presents the conclusion and some discussion about future work.

2. BACKGROUND

In this section, we introduce some of the underlying concepts of UADs, model-driven testing, and mutation analysis.

2.1. *UML Activity Diagrams*

The Unified Modeling Language (UML) provides a set of diagrams which enable developers to specify a system from different views, one of which is the UML Activity Diagram (UAD), which is widely used for behavior modeling. The UAD can be used to depict the control flows of a certain operation in the system, or for the entire system, supporting all control structures, such as sequences, loops, branches, and concurrency.

A UAD usually consists of activities, transitions, decision points, guards, parallel activities, swimlane guidelines and action-object guidelines. Figure 1 shows an example of a UAD [1] which models the workflow of a university enrollment system. A *start* point and an *end* point are modeled as a filled circle and a filled circle with a border around it, respectively. A *transition* describes the data-flow or control-flow between two activities. A *decision point* (modeled as a diamond) concerns selection of the following activities based on those preceding the decision point. The *decision* point is sometimes called a *branch* activity or *merge* activity. A *guard* (depicted using the format [· · ·] on the transitions) is a condition that must be true in order to traverse a transition. For example, the decision point in Figure 1 has three exiting transitions, each of which has a guard condition. To traverse the direct transition from the decision point to the activity "Enroll in University", the guard condition [trivial problems] must be true. *Parallel* activities (depicted using two parallel bars) are used to show that activities can occur in parallel. The first bar (e.g. the left one in Figure 1) and the second one (e.g. the right one in Figure 1) are called *fork* and *join* activities, respectively. In some situations, called *loop*s, an activity repeats until a specific condition is satisfied — these structures have two incoming and one outgoing transition. Each activity in a UAD, therefore, is either an *action*, a *start*, an *end*, a *fork*, a *join*, a *branch*, a *merge*, or a *loop* type.

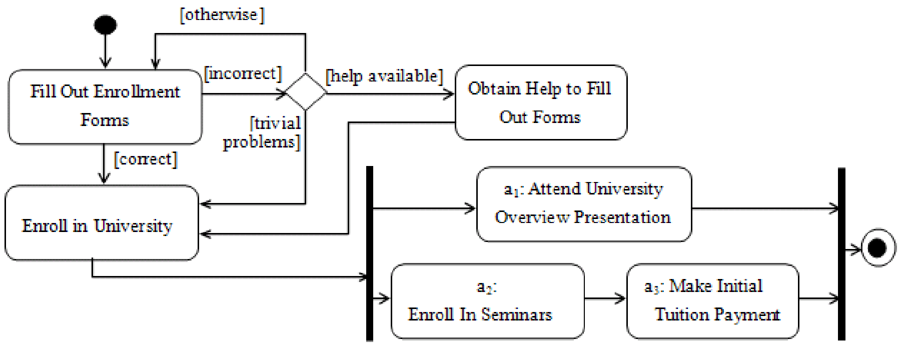

[otherwise]
[incorrect]
[help available]
Fill Out Enrollment Forms
Obtain Help to Fill Out Forms
[trivial problems]
[correct]
a1: Attend University Overview Presentation
Enroll in University
a2: Enroll In Seminars
a3: Make Initial Tuition Payment

Figure 1. An example of a UAD

Parallel activities and decision points require more attention when deriving test scenarios from UADs. On the one hand, fork and join activities represent concurrent behavior, and it is necessary that all parallel activities be executed when testing the system. For instance, each test scenario derived from the UAD in Figure 1 should contain the activities "Enroll in Seminar(s)", "Make Initial Tuition Payment", and "Attend University Overview Presentation". However, the order of these activities may vary. On the other hand, branch and merge activities represent optional behavior, requiring only that one of the optional activities be executed when testing the system. Due to the semantics of parallel activities and decision points, automatic derivation of such test scenarios from a UAD is a difficult task, and when the system under development is large and complex, the combination or nesting of optional behavior and concurrent behavior may make test scenario generation even more difficult.

2.2. *Model-driven Testing*

Modeling is an important and necessary step in the development of large-scale software-intensive systems: it usually captures the important system aspects, and thus enables exploration and analysis of key points and properties. MDT (Model-Driven Testing) [9] is a testing method based on the modeling artifacts. As illustrated in Figure 2, MDT usually consists of the following steps: (i) Modeling a system; (ii) Deriving abstract tests from the model; (iii) Implementing a system from the model; (iv) Refining the abstract tests into executable tests; and (v) Running the executable tests against the system and comparing the actual results and expected ones. MDT makes the testing (steps ii and iv) and coding (step iii) of a system parallel jobs, thus enabling testers to start designing tests at the modeling phase rather than waiting until the implementation phase, something particularly useful when the schedule is tight and/or testing resources are limited.

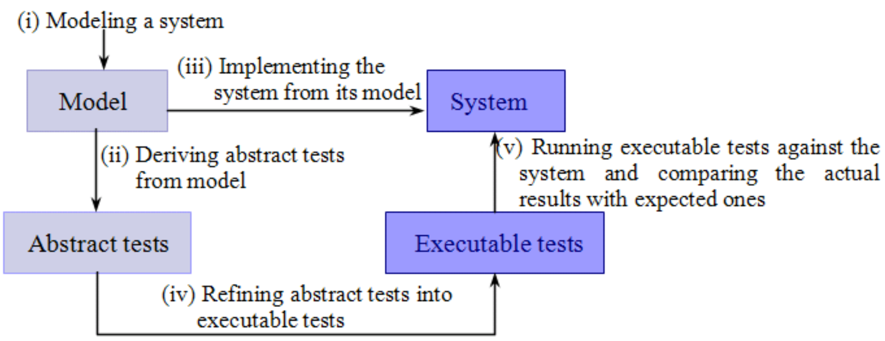

(i) Modeling a system
Model
(iii) Implementing the system from its model
System
(ii) Deriving abstract tests from model
(v) Running executable tests against the system and comparing the actual results with expected ones
Abstract tests
Executable tests
(iv) Refining abstract tests into executable tests

Figure 2. The main steps of MDT

Given the widespread use of UML, an interesting question is that of how to test a system based on its UML diagrams. In this paper, we investigate how to effectively and efficiently test a system with concurrent behavior based on its UAD. A key issue is to generate test scenarios from the UAD, and use these scenarios to help discover defects earlier.

2.3. *Mutation Analysis*

Mutation analysis is a fault-based testing technique which hypothesizes certain types of faults that may be injected by programmers, and then designs test cases targeted at uncovering such faults [10]. Mutation analysis has also been widely employed to evaluate the effectiveness of various software testing techniques [35]. It applies mutation operators (each describing a simple syntactic change to the code) to seed faults into the program under test, thereby generating a set of faulty versions, called mutants. After creating the mutants, a set of test cases is executed on them. When a test case results in a mutant producing different (incorrect) output compared with the original program, then we say that that mutant is "*killed*". The mutation score *(MS)* measures the adequacy of a set of test cases. It is defined as follows:

$$MS(p,t) = \frac{N_k}{N_m - N_e} \tag{1}$$

where p refers to the program being mutated, t is the test suite, N_k is the number of killed mutants, N_m is the total number of mutants, and N_e is the number of equivalent mutants. An equivalent mutant is one whose behavior is the same as that of p, for all test cases. The automatically generated mutants can be very similar to real-life faults [2], making *MS* a good indicator of the effectiveness of a testing technique [35]. We therefore use the *MS* to evaluate our proposed approach.

3. TRANSFORMATION-BASED SCENARIO-ORIENTED TEST CASE GENERATION

In this section, we present a transformation-based scenario-oriented test case generation method based on UADs.

3.1. Overview

Generating test cases from UADs is a kind of model-driven testing technique which can be more challenging than traditional black-box or white-box testing due to the fact that there are multi-design aspects which are usually separate. We assume that sufficient information has been provided in the UAD specification, and that it contains no inconsistencies. These two assumptions are justifiable because UML-based software development emphasizes modeling the system's structure and behavior in order to support automatic code generation. This means that sufficiently detailed information is provided in the design model, and inconsistency checking is performed before testing begins.

To deal with the challenges in activity diagrams due to *fork* and *join* activities, we propose a transformation-based approach to generating test cases, focusing mainly on the testing of concurrent activities (illustrated in Figure 3). The approach includes the following four major steps:

1. *Preprocessing*: Parse the UAD specification (often represented in an XML Metadata Interchange (XMI) file) to produce a UAD graph model, which not only contains a set of activity and transition entries, but also has essential information for the test case generation.
2. *Transformation*: Based on a set of transformation rules, convert the UAD graph into an intermediate representation called an Extended AND-OR Binary Tree (EBT). This results in fork and join elements being eliminated, and other branches (including loops) and concurrent flows all being represented as EBTs. Using such a standard representation, test scenario generation algorithms can easily be developed.
3. *Generating test scenarios*: Traverse the derived EBTs to generate a set of test scenarios with respect to the given concurrency coverage criteria. The EBTs are a general test specification language based on which we can develop algorithms satisfying different concurrency coverage criteria.
4. *Test data generation*: For each test scenario, derive test data by selecting the corresponding decisions along the test scenario. To do this, we can first randomly generate a test data pool, and then select the appropriate data as test cases. In our future work, we will use constraint solver techniques to automatically generate the test data for each test scenario.

We next discuss in detail the main steps of the proposed approach.

3.2. Preprocessing

Most existing UML tools (such as IBM Rational Rose[†] and ArgoUML[‡]) support UAD modeling. In order to facilitate the exchange of modeling artifacts, UML specifications are normally stored as XMI (XML Metadata Interchange) files. To generate tests from the UAD specifications, we need to extract the elements or properties that are essential for testing. Such preprocessing has two implications: (1) We can skip those elements or properties unrelated to testing; and (2) Transformation rules and test scenario generation algorithms can be developed without being restricted to a specific input format — thus they are reusable.

For each activity a, we extract an entry <*ID*, *resposeID*, *noOutTransitions*, *type*, *name*>, where

- *ID* is the unique label of a. All activities are labeled as follows:
  (1) Starting from zero and labeling in order the activities from *start* to *end*.

[†] http://www-03.ibm.com/software/products/en/ratirosefami/
[‡] http://argouml.tigris.org/

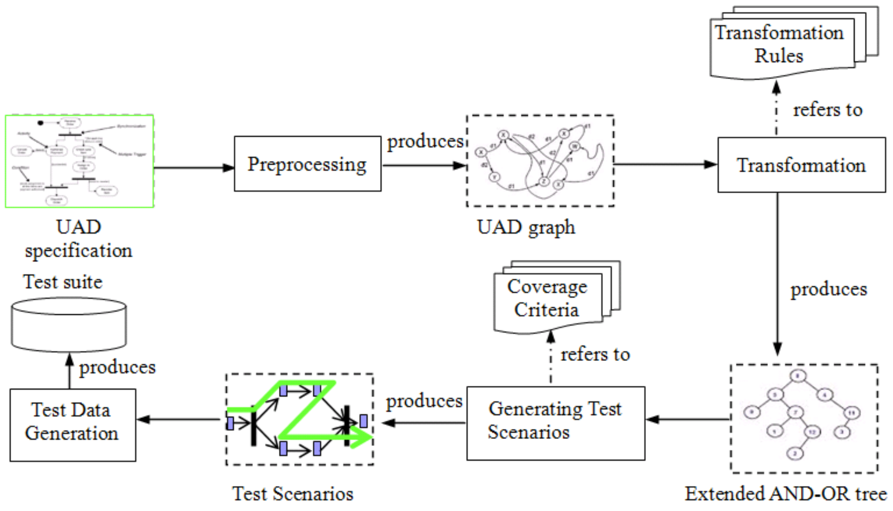


Figure 3. An illustration of the transformation-based framework

(2) When a *branch* or *fork* activity is encountered, labeling the activities in order along the path until all activities have been processed.
(3) When an activity whose *noOutTransitions* is greater than two is encountered, if the ID of the following activity has been assigned, then skipping over this path; otherwise, switching to another following activity and repeating steps (1), (2), and (3).

- *responseID* is a specially designed label which is used to identify the hierarchy of the activity. It is defined as follows:
  - If the type is *start*, *action*, or *loop*, its *responseID* is the ID of next activity whose type is not *action*;
  - If the type is *branch*, its *responseID* is the ID of its corresponding *merge* activity;
  - If the type is *fork*, its *responseID* is the ID of its corresponding *join* activity;
  - If the type is *merge* or *join*, its *responseID* is equal to its ID plus 1.
  - If the type is *end*, its *responseID* is its own ID.
- *noOutTransitions* is the number of transitions leading to activities from a.
- *type* is the type of a — either $start$, end, $action$, $fork$, $join$, $branch$, $merge$, or $loop$.
- *name* is the name of a.

For each transition t, we extract an entry <*iID*, *oID*>, where

- *iID* is the ID of the incoming activity of t.
- *oID* is the ID of the outgoing activity of t.

We store the extracted activities and transitions in a UAD graph structure, which is defined as follows.

Definition 1 (UAD Graph): A UAD graph is a 4-tuple $< A, T, C, L >$, where $T : A \times C \rightarrow A$, and where: A is the collection of activity entries; T is the set of transitions; C is the set of constraint conditions that must be satisfied when transitions in T happen; and L is a label function which assigns a unique label to each activity and transition.

3.3. *Transformation*

In the next step, our approach converts the UAD graph into an EBT using a set of transformation rules. We next provide a formal definition of an EBT, and then present the transformation rules.

Definition 2 (EBT): An EBT is a tuple $< N, E, \psi >$, where $N = \overrightarrow{A}$ + {*BOR*, *MOR*, *FAND*, *JAND*, *CYCLE*}, with $\overrightarrow{A}$ being the nodes mapped from activities A in the UAD; *BOR* and *MOR* are extended *OR* nodes designed for *branch* and *merge* activities, respectively; *FAND* and *JAND* are extended *AND* nodes designed for *fork* and *join* activities, respectively; *CYCLE* is designed for *loop* activities; E is a set of two-tuples $< n_1, n_2 >$ where $n_1 \in N$ and $n_2 \in N$; and ψ is a function which assigns a unique label for each node and edge.

Rule 1 (Transformation rule for an *action* activity): A common UAD *action* activity does not require special treatment, and is converted into a $normal$ node, and stored as a left child of the EBT. The *start* activity should be converted into a root node of the EBT, and the *end* activity should be converted into a leaf node.

Rule 2 (Transformation rule for a *branch* or *fork* activity): For a UAD *fork* activity with two outgoing transitions $t_1, t_2 \in T$, (where t_i is $(a, c_i) \to a_i$, $1 \le i \le 2$, $c_i \in C$, $a_i \in A$) generate a node *n* to replace *a*, and add a logical node $FAND$. Connect *n* and $FAND$ with an edge labeled *NULL*, and set the outgoing transition edges of $FAND$ to $FAND \longrightarrow^{\mathrm{e}_1} n_1$ and $FAND \longrightarrow^{\mathrm{e}_2} n_2$, where e_1 and e_2 are the mapped edges of transitions t_1 and t_2 in the UAD's T; and n_1 and n_2 are the mapped nodes of activities a_1 and a_2 in the UAD's A.

The transformation of a *branch* activity is similar to that for *fork*, but replacing *FAND* with *BOR*.

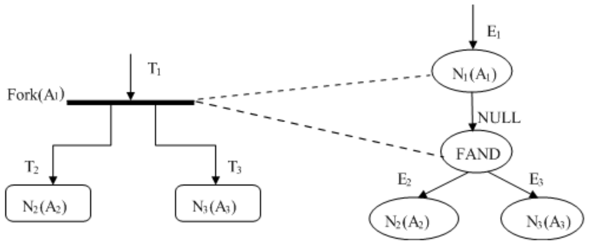


Figure 4. Transformation of a *fork* activity

With the above transformation rule, we can convert a fork activity into an EBT by first creating a corresponding node and then adding a $FAND$ node as its left child; we then treat the two following parallel activities as the left and right child nodes of the $FAND$ node. Figure 4 illustrates this transformation of a *fork* activity. For a *branch* activity, we first create a corresponding node, add a *BOR* node as its left child, then treat the two following branch activities as the left and right child nodes of the *BOR* node.

Rule 3 (Transformation rule for a *join* or *merge* activity): For a UAD *join* activity *a* with multiple incoming transitions $t_1, \ldots, t_m \in T$ (where $m \ge 2$, t_i is $(a_i, c_i) \to a$, $1 \le i \le m$, $c_i \in C$, and $a_i \in A$), generate a node *n* to replace *a* and add a logical node $JAND$. Connect *n* and $JAND$ with an edge labeled $NULL$; and set the incoming transition edges of $JAND$ to $n_1 \longrightarrow^{\mathrm{e}_1} JAND, \ldots, n_m \longrightarrow^{\mathrm{e}_\mathrm{m}} JAND$, where $e_1, \ldots, e_m$ are the mapped edges of transitions $t_1, \ldots, t_m$ in the UAD's T; and $n_1, \ldots, n_m$ are the mapped nodes of activities $a_1, \ldots, a_m$ in the UAD's A.

The transformation of a *merge* activity is similar to that for *join*, but replacing $JAND$ with MOR.

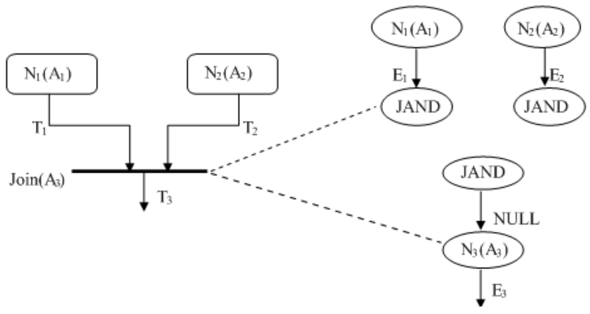


Figure 5. The transformation of a *join* activity

With the above transformation rule, we can convert a *join* activity into a set of EBTs by first creating a corresponding node, and then adding a $JAND$ node as its parent, and then treating each preceding parallel activity as a parent of the $JAND$ node. Figure 5 illustrates this transformation of a *join* activity. Similarly, for a *merge* activity, we first create a corresponding node, add a MOR node as its parent, and then treat each preceding branch activity as a parent of the MOR node.

Rule 4 (Transformation rule for a *multi-fork* or *multi-branch* activity): For a multi-fork activity a with more than two outgoing transitions $t_1, \ldots, t_m \in T$, where $m > 2$, the transformation is done as follows: (i) create a corresponding $NULL$ node a' to replace a, and add a logical $FAND$ node f as its left child; (ii) randomly select an unprocessed following parallel activity as the left child of f (using Rule 2), and add a logical node $FAND$ f' as the right child; (iii) if the number of the following unprocessed parallel activities is more than one, go to step (ii); otherwise, set the following unprocessed parallel activity as the right child of f'.

Figure 6 illustrates the transformation of a multi-fork activity. The transformation of a multi-branch activity is similar to that for multi-fork, but replacing *FAND* with *BOR*.

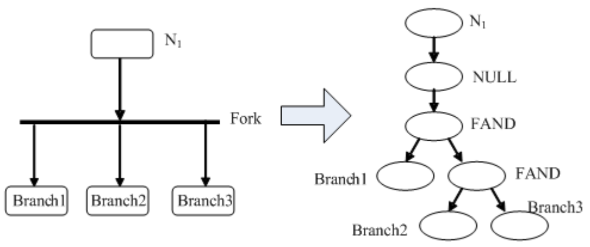


Figure 6. The transformation of a *multi-fork* activity

Unlike a common action activity, a loop activity has two incoming and one outgoing transition, as illustrated in the left part of Figure 7. We identify loop activities using the concept of post-activities, which is defined next. An activity *a* is a *loop* activity if and only if *a* is subsumed in its post-activities.

Definition 3 (Post-Activities of an activity): Given an activity *a* in the UAD, the post-activities of *a* are defined as $postActivities(a^+) = a^1 \cup a^2, ..., \cup a^n$, where $a^1 = postActivities(outTransitions(a)), \cdots, a^n = postActivities(outTransitions(a^{n-1}))$; $outTransitions(a)$ is a set of outgoing transitions of *a*; and $postActivities(t)$ is a set of target activities of transition *t*.

We convert a *loop* activity into the combination of one *merge* activity and the original activity, as illustrated in the right part of Figure 7. After the transformation, one fake *merge* activity A' and one fake transition T' are introduced into the converted UAD. The transformation does not cause additional elements from the point of view of testing, but it does have problems: the *merge* and *branch* nodes do not match; and the *merge* node is ahead of the *branch* node, which will cause the test scenario derivation algorithm (to be described later) to fail. Considering that each conditional branch of the loops should be covered at least once from the point of view of testing, we next propose a transformation rule for a loop activity.

Rule 5 (Transformation rule for a *loop* activity): For a UAD *loop* activity with two incoming transitions $t_1, t_2 \in T$ (where t_1 is $(a_x, c_1) \rightarrow a$; t_2 is $(a_y, c_2) \rightarrow a$; $a_x, a_y \in A$; and $a_y \in postActivities(a^+)$), generate a *cycle* node *n* to replace *a*, and generate a sequence of nodes

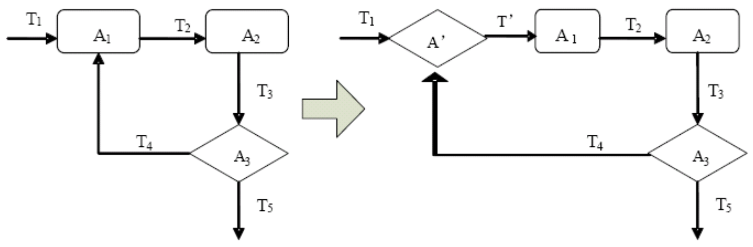


Figure 7. An illustration of a *loop* activity

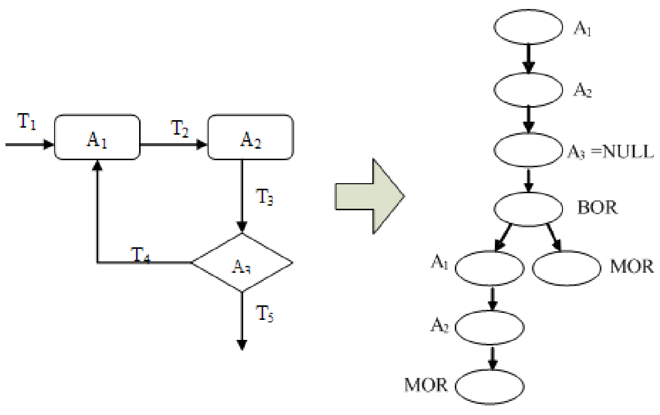


Figure 8. The transformation of a *loop* activity

n_i to replace activities in $postActivities(a^+) \setminus a_y$($n$ and nodes n_i are referred to as the *loop body*). Add a logical node *BOR* and one logical node *MOR*, and connect them; add another logical node *MOR*, connect *BOR* to the *loop body*, and connect the *loop body* to *MOR*.

With the above transformation rule, we can convert a *loop* activity into a set of EBTs. We first create a sequence of nodes to represent the loop body, then create two branches as leaves of the *BOR* node: one is the loop body (as the left child), and the other is $empty$ (as its right child); both of them end with a *MOR* node. Figure 8 shows this transformation of the *loop* activity illustrated in Figure 7.

We developed the algorithm shown in Figure 9 to implement the automatic transformation. The algorithm reads in a UAD graph model and outputs a set of EBTs. It traverses all activities in the graph model and handles each activity according to its type. The conversions are conducted based on the transformation rules above: (i) if the current activity is an *action* activity, convert it to a *NORMAL* node and add it to the EBT; (ii) if the current activity is a *fork* or *branch* activity, create a *FAND* or *BOR* node and add it to the EBT, then get the child activities of the current activity and convert them recursively; (iii) if the current activity is a *join* or *merge* activity, create a *JAND* or *MOR* node and add it to the EBT, then get the child activities of the current activity and convert them recursively; (iv) if the current activity is a *loop* activity, create a *CYCLE* node and add it to the EBT, and then recursively get the child activities and convert them.

Although a theoretical proof of the transformation's correctness would be ideal, it is well known that the UML is a visual formal modeling language with a limited formalism, and the semantics of UADs is provided in UML standard manual. Given the difficulty in proving the transformation correctness, we instead turn to testing to validate the transformations.

3.4. *Generating Test Scenarios*

A UAD describes the important business scenarios of a system being modeled, with a path leading from a *start* activity to an *end* activity forming a test scenario. The question of generating test scenarios from UADs, therefore, is reduced to one of deriving executable paths from a directed graph. With the proposed transformation rules, we can now convert UADs containing fork and join concurrency into extended binary trees (EBTs). Here, we discuss how to derive test scenarios from the EBTs.

Definition 4 (Test scenario): A test scenario TS, derived from a UAD, is a sequence of activities $\{a_1, a_2, \cdots, a_n\}$, where a_i $(i = 1..n) \in A$ of the UAD (the type of each a_i $(i \neq 1 \wedge i \neq n)$ is one of *action*, *fork*, *join*, *branch*, *merge*, or *loop*; and a_1 and a_n are the *start* and *end* nodes of the UAD, respectively). There must be a transition between any a_i and a_{i+1} (i=1..n-1), or a_i and a_{i+1} must come from different parallel activities.

```
Input: G is a UAD graph;
Output: T is a set of EBTs;
PROCEDURE Generate(G, T)
1.  $nodeN$ ← the number of activities of G, $T \leftarrow \emptyset$;
2.  FOR i = 1 TO $nodeN$ DO
3.     $node$ ← getActivity(G, i);
4.     IF($node.type == “begin'' \; OR \; node.type == “action''$) THEN
5.        $node$.type ← “NORMAL”;
6.        $node$.lChild ← $node$.afterNodes.get(0);
7.        $T \leftarrow T \cup \{node\}$;
8.     END IF
9.     ELSE IF($node.type == “fork'' \; OR \; node.type == “branch''$) THEN
10.       $node$.type ← “NULL”;
11.       create a new node $newNodeA$ and add $newNodeA$ to $node$.lchild;
12.       $newNodeA$.type ← “FAND” or “BOR”;
13.       $n$ ← node.afterNodes.size();
14.       FOR $j = 1$ TO ($n$-2) DO
15.          $newNode$.lChild ← $node$.afterNodes.get(j);
16.          create a new node $newNodeB$ and add $newNodeB$ to $node$.rchild;
17.          $newNodeB$.type ← “FAND” or “BOR”;
18.          $newNodeB$.lChild ← $node$.afterNodes.get($n$-1);
19.          $newNodeB$.rChild ← $node$.afterNodes.get($n$);
20.       END FOR
21.       $T \leftarrow T \cup \{node\}$;
22.    ELSE IF ($node.type == “merge'' \; OR \; node.type == “join''$) THEN
23.       $node$.type ← “NULL”;
24.       create a new node $newNode$;
25.       $newNode$.type ← “MOR” or “JAND”;
26.       startNodes.add(newNode); //add $newNode$ to startNodes;
27.       $node$.lChild ← $node$.afterNodes.get(0);
28.       FOR $j = 1$ TO $nodeN$ DO
29.          tmpNode ← getNode(G, j);
30.          IF (tmpNode.afterNodes.get(0) == tmpNode) THEN
31.            add $newNode$ to tmpNode.lChild;
32.          ENDIF
33.       END FOR
34.       $T \leftarrow T \cup \{node\} \cup \{newNode\}$;
35.    END IF
36.    ELSE IF($node.type == “loop''$) THEN
37.       $node$.type ← “CYCLE”;
38.       create a new node $newNodeA$ and $newNodeA$.type ← “BOR”;
39.       $newNodeA$.lchild ← $node$.afterNodes.get(0);
40.       $node$.lchild ← $newNodeA$;
41.       create a new node $newNodeB$ and $newNodeB$.type ← “BOR”;
42.       $newNodeA$.rchild ← $newNodeB$;
43.       $T \leftarrow T \cup \{node\}$;
44.    END IF
45.    ELSE IF ($node.type == “end''$) THEN
46.       $node$.type ← “END”;
47.       $T \leftarrow T \cup \{node\}$;
48.    END IF
49.  END FOR
50.  RETURN $T$;
END PROCEDURE
```

Figure 9. Algorithm for converting a UAD graph to a set of EBTs

Generating test scenarios is a key step in the generation of test cases. Because it is usually impossible or infeasible to test all possible paths (due to limited testing resources), three coverage criteria [33] have been proposed:

1. *Weak concurrency coverage*. Test scenarios are derived to cover *only one* feasible sequence of parallel processes, without considering the interleaving of activities between parallel processes.
2. *Moderate concurrency coverage*. Test scenarios are derived to cover *all* feasible sequences of parallel processes, 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.

Consider the parallel activities of the UAD example in Figure 1. For clarity, we denote the activities "Attend University Overview Presentation", "Enroll in Seminar(s)" and "Make Initial Tuition Payment" as a_1, a_2, and a_3, respectively. When the *weak concurrency coverage* is used, *either* "$a_1 \rightarrow a_2 \rightarrow a_3$" or "$a_2 \rightarrow a_3 \rightarrow a_1$" should be tested; for the *moderate concurrency coverage*, *both* "$a_1 \rightarrow a_2 \rightarrow a_3$" and "$a_2 \rightarrow a_3 \rightarrow a_1$" should be tested; and for the *strong concurrency coverage*, "$a_1 \rightarrow a_2 \rightarrow a_3$", "$a_2 \rightarrow a_3 \rightarrow a_1$", and "$a_2 \rightarrow a_1 \rightarrow a_3$" should *all* be tested. Clearly, these concurrency coverage criteria require that the derived test scenarios cover each parallel process at least once. Both weak and moderate concurrence coverage test the activities and control flows within a parallel process in a sequential way. Strong concurrency coverage considers the crossing of activities and control flows from parallel processes, which may result in a huge number of test scenarios, and thus may be impractical. Our proposed approach has so far implemented only the weak and moderate concurrency coverage.

We propose an algorithm (shown in Figure 10) to generate test scenarios from EBTs, with respect to weak concurrency coverage. The algorithm generates test scenarios from a *start* node to an *end* node recursively. The algorithm first traverses all the nodes of each EBT to generate test scenarios according to their types, and then combines all partial test scenarios for each part to form complete test scenarios. We next briefly explain this algorithm.

- If the *start* node is a NORMAL node, add it to the generated test scenario, and process the next following node.
- If the *start* node is a *FAND* or *BOR* node, first create two scenarios: one from the *start* node to the matching response node that is specified by its *responseID*, and the other from the first node after the matching response node to the *end* node; then merge these two scenarios together.
- If the *start* node is a *JAND* or *MOR* node, traverse all nodes starting from the start node of another EBT.
- If the *start* node is a *CYCLE* node, first create two scenarios: one from the first node to the last node in the loop, and another from the first node after the *CYCLE* to the last node; then merge them together.
- If the *start* node is the END node, add it to the generated test scenario, and return the resulting sequence.

Because the algorithm in Figure 10 traverses all nodes in the EBTs, its complexity is proportional to the number of nodes n, namely $O(n)$.

We also propose an algorithm to generate test scenarios satisfying the moderate coverage criteria, as shown in Figure 11. The algorithm is similar that in Figure 10, except that additional processing is required in relation to the FAND node (permutation between the node and its corresponding *responseID* node).

- If the *start* node is a NORMAL node, add it to the generated test scenario.
- If the *start* node is a *BOR* node, first create two test scenarios: one from the *start* node to the matching response node specified by its *responseID*, and the other from the first node after the matching response node to the *end* node; then merge these two scenarios together.

```
INPUT: $T$ is a set of EBTs;
OUTPUT: $ResultPaths$ is a set of test scenarios;
PROCEDURE WeakCoverage(Node start, Node end)
1.  tmpPaths ← ∅, resultPaths ← ∅, tmpPaths1 ← ∅, tmpPaths2 ← ∅, set all elements in tmp[] to 0;
2.  IF start ≠ end THEN
3.    IF start.type ==“NORMAL” THEN
4.      add start to tmpPaths;
5.      tmpPaths ← WeakCoverage(start.afterNodes.get(0), end);
6.      resultPaths ← merge resultPaths and tmpPaths;
7.      RETURN resultPaths;
8.    ENDIF
9.    IF start.type ==“FAND” OR start.type == “BOR” THEN
10.     FOR i ← 1 TO start.afterNodes.size() DO
11.       tmp[i] ← WeakCoverage(start.afterNodes.get(i), responseNode);
12.     END FOR
13.     tmpPaths ← WeakCoverage(start.afterNodes.get(0), end);
14.     resultPaths ← merge tmp[ ] and tmpPaths;
15.     RETURN resultPaths;
16.   END IF
17.   IF start.type ==“MOR” OR start.type == “JAND” THEN
18.     add startNodes to temNode;
19.     tmpPaths1 ← WeakCoverage(tmpNode.lChild,responseNode);
20.     tmpPaths2 ← WeakCoverage(tmpNode.rChild,responseNode);
21.     resultPaths ← merge tmpPaths1 and tmpPaths2;
22.     RETURN resultPaths;
23.   END IF
24.   IF start.type ==“CYCLE” THEN
25.     tmpPaths1 ← WeakCoverage(first node of the loop, last node of the loop);
26.     tmpPaths2 ← WeakCoverage(first node after the loop, end);
27.     resultPaths ← merge tmpPaths1 and tmpPaths2;
28.     RETURN resultPaths;
29.   END IF
30. ELSE
31.     add start to resultPaths;
32.     RETURN resultPaths;
33. END IF
END PROCEDURE
```

Figure 10. Algorithm for generating test scenarios with weak concurrency coverage

- If the *start* node is a *FAND* node, first create three partial test scenarios and store them separately. Then combine the three partial test scenarios to get all possible scenarios.
- If the *start* node is an *MOR* or *JAND* node, then process all nodes in another EBT starting from the *start* node.
- If the *start* node is a *CYCLE* node, first create two test scenarios: one from the first to the last node in the loop, and the other from the first node after the loop to the *end* node; and then merge these two scenarios together,
- If the *start* node is the END node, the add it to the generated test scenario, and return the resulting sequence.

The algorithm generates partial test scenarios by only processing each node once in the EBTs. To combine the concurrent nodes, the algorithm needs to repeatedly call a traversal function. Assuming that the number of concurrent nodes is *m*, then the number of concurrent partial test scenarios is $m!$, making the overall complexity of the algorithm $O(m! + k(nm))$, where n is the number of EBT nodes, and k is a constant.

3.5. Test Data Generation

The Category Partition Method (CPM) [29] is used to create functional test suites from 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, and a *test frame* is a set of choices. In the context of test scenarios, we revise these definitions: each complete and feasible test

```
INPUT: $T$ is a set of EBTs
OUTPUT: $ResultPaths$ is a set of test scenarios
PROCEDURE ModerateCoverage(Node start, Node end)
1.  tmpPaths ← ∅, resultPaths ← ∅, tmpPaths1 ← ∅, tmpPaths2 ← ∅, set all elements in tmp[] to 0;
2.  IF start ≠ end THEN
3.    IF start.type ==“NORMAL” THEN
4.        tmpPaths ← ModerateCoverage(start.afterNodes.get(0), end);
5.        resultPaths ← merge resultPaths and tmpPaths;
6.        RETURN resultPaths;
7.    END IF
8.    IF start.type ==“BOR” THEN
9.        FOR i ← 1 TO start.afterNodes.size() DO
10.           tmp[i] ← ModerateCoverage(start.afterNodes.get(i), responseNode);
11.       END FOR
12.       tmpPaths ← ModerateCoverage(start.afterNodes.get(0), end);
13.       resultPaths ← merge tmp[ ] and tmpPaths;
14.       RETURN resultPaths;
15.   END IF
16.   IF start.type ==“FAND” THEN
17.       leftPaths ← ModerateCoverage(start.lChild, responseNode);
18.       midPaths ← ModerateCoverage(start.rChild.lChild, responseNode);
19.       rightPaths ← ModerateCoverage(start.rChild.rChild, responseNode);
20.       resultPaths ← Array(leftPaths, midPaths, rightPaths);
21.       RETURN resultPaths;
22.   ENDIF
23.   IF start.type ==“MOR” OR start.type ==“JAND” THEN
24.       add start to temNode;
25.       tmpPaths1 ← ModerateCoverage(tmpNode.lChild,responseNode);
26.       tmpPaths2 ← ModerateCoverage(tmpNode.rChild,responseNode);
27.       resultPaths ← merge tmpPaths1 and tmpPaths2;
28.       RETURN resultPaths;
29.   ENDIF
30.   IF start.type ==“CYCLE” THEN
31.       tmpPaths1 ← ModerateCoverage(first node of the loop, last node of the loop);
32.       tmpPaths2 ← ModerateCoverage(first node after the loop, end);
33.       resultPaths ← merge tmpPaths1 and tmpPaths2;
34.       RETURN resultPaths;
35.   END IF
36. ELSE
37.       add start to resultPaths;
38.       RETURN resultPaths;
39. END IF
END PROCEDURE
```

Figure 11. Algorithm for generating test scenarios with moderate concurrency coverage

scenario is a complete test frame, and a test case with respect to a particular test scenario corresponds to a set of choices whose values can be used to execute the test scenario. 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 in the values for those guards and inputs required in each activity along the scenario path.

Recall the example in Figure 1 (in Section 2.1), the guard condition of the “Obtain Help to Fill Out Forms” activity is “[help available]”, the guard condition of the *branch* activity is “[incorrect]”, and there is a dependency between the choices “[help available]” and “[incorrect]” because the choice “[help available]” holds depending on the choice “[incorrect]”. If a test frame does not satisfy this dependency, it is infeasible; otherwise, it is feasible. For instance, “[incorrect]”→“[help available]” is a feasible test frame, while “[correct]”→“[help available]” is an infeasible test frame.

To partially automate the test data generation, we randomly generate a set of potential test data, and select as test cases those satisfying the choices. Specifically, we first analyze the input format of the given implementation. For instance, the acceptable input of the student enrollment system in the example could be a student record, potentially consisting of identity, name, age, gender, etc.

Next, we randomly generate a large number of test cases without considering possible constraints among the input variables. Finally, we select those test cases that satisfy the guard conditions within a test frame to be part of the test suite. Techniques that can select test case for a program path, such as constraint solver techniques [25], can also be used. In the current study, we give higher priority to boundary valves in order to improve the fault detection capability of our approach, and thus the selection of test cases has been done manually. Automatic test case generation for a specific scenario path is left for our future work.

4. TOOL PROTOTYPE

In this section, we introduce a tool, ConcurTester, developed to automate the proposed approach.

4.1. Features

When UADs are used to model complex business processes or workflow systems, the result may often contain a large number of activities and transitions. Because generating test scenarios from such a UAD is time-consuming and error prone, a tool which could automate the proposed approach is highly desirable.

ConcurTester was developed using Java. It consists of 1966 lines of code, and has the following functionality:

1. *Preprocessing*: It imports the UAD specification file (in XMI format), and parses it to extract the relevant elements, including activity and transition entries, then stores them as a graph structure.
2. *Transformation*: It converts the graph structure into EBTs based on the transformation rules.
3. *Generating test scenarios*: It generates test scenarios from the EBTs with respect to different concurrency coverage criteria, and presents the generated scenarios for further analysis.

4.2. Pilot test

We use the UAD shown in Figure 12 to illustrate usage of ConcurTester. First, we import the UAD specification of the system to be tested by clicking the *Import File* button on the tool. We assume the specification is a standard XMI file generated by ArgoUML[§].

After the specification is imported, we click the *Convert File* button to parse the XMI file and convert it into a graph structure containing the activity and transition entries. When the conversion is finished, a text file is created to store the graph structure.

After generating the graph structure, we click the *Generate Tree* button to transform it into EBTs using the transformation rules. The transformation result is presented in the *Generated AND-OR Tree* tab, as shown in Figure 13.

After generating the EBTs, we click the *Generate Path* button to create a set of test scenarios. The *"Weak Coverage Path"* tab presents the generated test scenarios satisfying the weak coverage criterion (as shown in Figure 14), and the *"Moderate Coverage Path"* tab shows the generated test scenarios satisfying the moderate coverage criterion (as shown in Figure 15).

5. CASE STUDY

In this section, we report on a case study conducted to examine the proposed approach and evaluate its effectiveness. In the study, we employed ConcurTester to automatically generate test scenarios, using a product ordering system as subject program. Mutation analysis was used to evaluate the effectiveness of the proposed approach.

[§]ArgoUML is a widely recognized, open source UML modeling tool, available from the following website: http://argouml.tigris.org/.

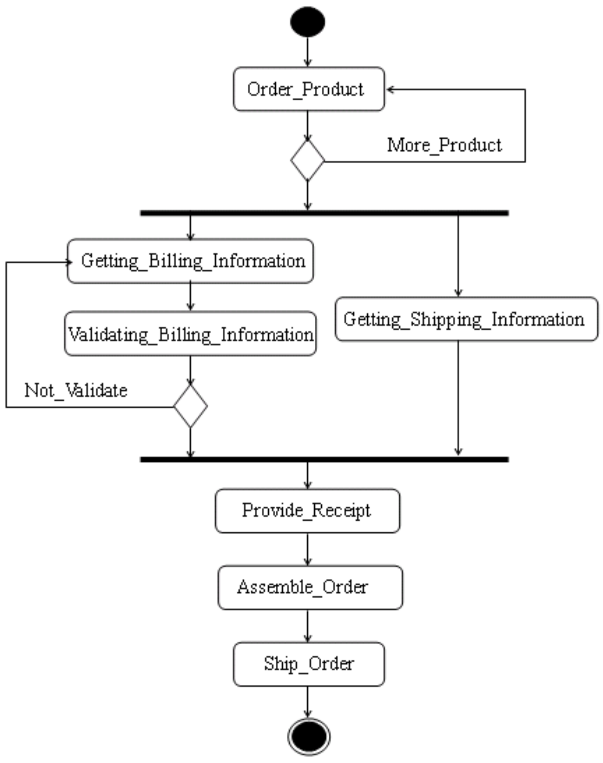
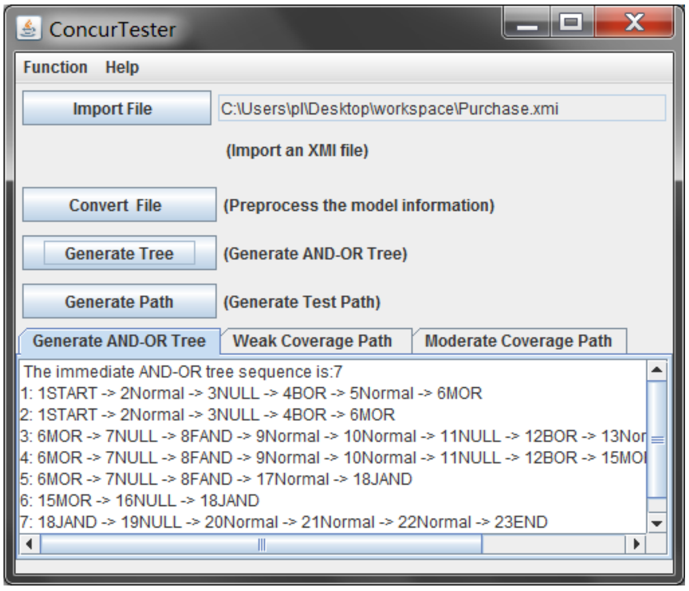

Order_Product
More_Product
Getting_Billing_Information
Validating_Billing_Information
Getting_Shipping_Information
Not_Validate
Provide_Receipt
Assemble_Order
Ship_Order

Figure 12. The *ordering products* process modelled by UAD



C:\Users\pl\Desktop\workspace\Purchase.xmi

(Import an XMI file)

(Preprocess the model information)

(Generate AND-OR Tree)

(Generate Test Path)

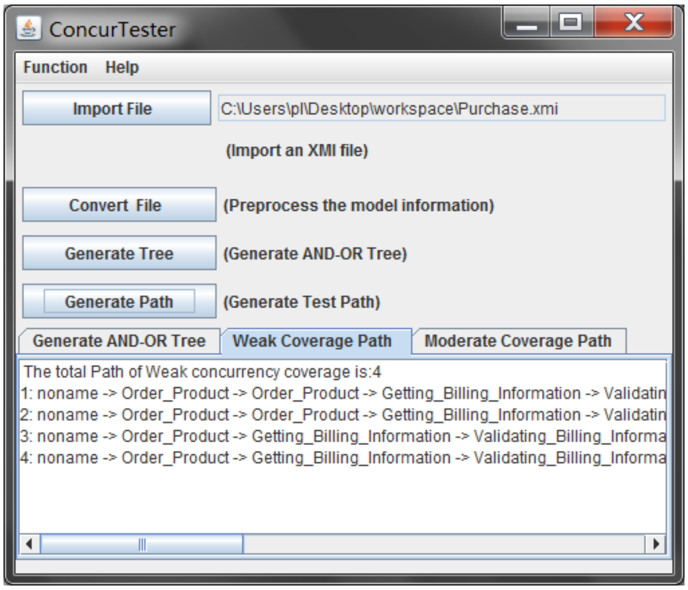
```
The immediate AND-OR tree sequence is:7
1: 1START -> 2Normal -> 3NULL -> 4BOR -> 5Normal -> 6MOR
2: 1START -> 2Normal -> 3NULL -> 4BOR -> 6MOR
3: 6MOR -> 7NULL -> 8FAND -> 9Normal -> 10Normal -> 11NULL -> 12BOR -> 13Nor
4: 6MOR -> 7NULL -> 8FAND -> 9Normal -> 10Normal -> 11NULL -> 12BOR -> 15MOI
5: 6MOR -> 7NULL -> 8FAND -> 17Normal -> 18JAND
6: 15MOR -> 16NULL -> 18JAND
7: 18JAND -> 19NULL -> 20Normal -> 21Normal -> 22Normal -> 23END
```

Figure 13. Generated intermediate AND-OR tree sequence of the UAD

5.1. Research Questions

Through this case study, we attempt to answer the following questions:

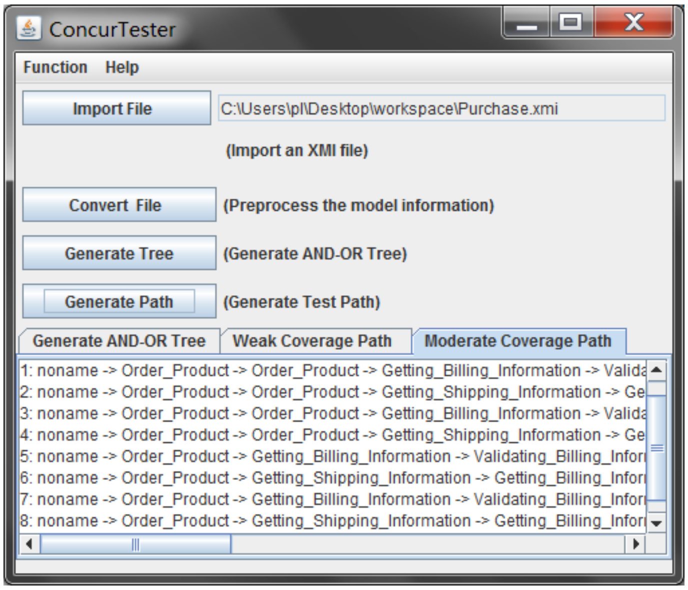
1. Is the proposed approach able to generate test cases for a program whose behavior is modeled by UAD?



```
C:\Users\pl\Desktop\workspace\Purchase.xmi
```

```
The total Path of Weak concurrency coverage is:4
1: noname -> Order_Product -> Order_Product -> Getting_Billing_Information -> Validatin
2: noname -> Order_Product -> Order_Product -> Getting_Billing_Information -> Validatin
3: noname -> Order_Product -> Getting_Billing_Information -> Validating_Billing_Informa
4: noname -> Order_Product -> Getting_Billing_Information -> Validating_Billing_Informa
```

Figure 14. Generated test paths for weak coverage criterion



```
C:\Users\pl\Desktop\workspace\Purchase.xmi
```

```
1: noname -> Order_Product -> Order_Product -> Getting_Billing_Information -> Valida
2: noname -> Order_Product -> Order_Product -> Getting_Shipping_Information -> Ge
3: noname -> Order_Product -> Order_Product -> Getting_Billing_Information -> Valida
4: noname -> Order_Product -> Order_Product -> Getting_Shipping_Information -> Ge
5: noname -> Order_Product -> Getting_Billing_Information -> Validating_Billing_Infor
6: noname -> Order_Product -> Getting_Shipping_Information -> Getting_Billing_Infor
7: noname -> Order_Product -> Getting_Billing_Information -> Validating_Billing_Infor
8: noname -> Order_Product -> Getting_Shipping_Information -> Getting_Billing_Infor
```

Figure 15. Generated test paths for moderate coverage criterion

2. What is the fault detection capability of the test suite generated using the proposed approach?

5.2. *Experimental Design*

We next describe the experimental settings, including the subject program, metrics, mutation, and procedure.

5.2.1. Subject Program An order processing (*product ordering*) system, the workflow for which is shown in Figure 12, was selected as subject program in the case study. It is small but representative, with most characteristics of a workflow system, and has been previously used to illustrate the

UAD [11]. Until now, however, there has not been an implementation available, so we implemented it using Java in a total of 281 lines of code. In our implementation, the program receives five input parameters, namely *morePro*, *proNum*, *proPrice*, *totalPrice*, and *shipInfo*: *morePro* indicates if there are remaining order items to be processed; *proNum* denotes the number of products; *proPrice* is the price of each product; *totalPrice* is the total price of all products; and *shipInfo* is the transport information.

The UAD in Figure 12 involves various types of activities, including *start*, *end*, *action* (*Getting_Shipping_Information*, *Validating_Billint_Information*, *Provide_Receipt*, *Assemble_Order*, and *Ship_Order*), *branch*, *merge*, *fork*, *join*, and *loop* activities (*Order_Prodcuts* and *Getting_Billing_Information*). *Getting_Billing_Information* involves calculation of cost, comparison of the prices of each product, and comparison of the total price with that of the calculated amount for the number of products; and *Validating_Billint_Information* verifies the cost. Among these activities, the billing information processing (*Getting_Billing_Information* and *Validating_Billint_Information*) and shipping information processing (i.e. *Getting_Shipping_Information*) are treated as parallel activities, which are accordingly implemented using concurrent threads.

5.2.2. Metrics The effectiveness of the proposed approach was measured using the *mutation score* (MS), which indicates the adequacy of a test suite for the program under test.

5.3. Experimental Procedure

1. Preprocessing, transformation, and generating test scenarios using ConcurTester: We used ConcurTester to parse the UAD specification (in an XMI file) for *product ordering*, as illustrated in Figure 12. After this, we extracted the collection of activity and transition entries and stored them as a graph structure. Then, ConcurTester was used to transform the graph structure into the intermediate representation (EBTs). During the transformation, all branches and concurrent flows were represented as EBTs. Each loop body was either executed once, or not at all. Finally, ConcurTester generated a set of test scenarios from the EBTs based on a given coverage criterion.
2. Test case generation: We generated a large amount of random test data, from which we selected only those satisfying the generated test scenarios to be in the test suite. As a result, we selected 15 test cases for each test scenario — when the weak coverage criterion was used, four test scenarios were generated in our experiments.
3. Seeding faults: In the study, an open-source mutation system, muJava [22], was used to randomly seed faults into the Java program for product ordering. The muJava system supports 16 types of method-level (traditional) and 29 types of class-level mutation operators[¶] of which 7 method-level and 5 class-level operators were applicable for our study. Using these applicable operators, a total of 170 method-level and 47 class-level mutants were generated, among which 30 method-level and 17 class-level mutants were equivalent, and therefore excluded.
4. Executing tests and collecting the results: We next applied each test in the test suite to both the original program and the mutants, comparing the output. If the output was the same, then the current test passed; otherwise, a fault was detected (and the mutant "killed"). If none of tests could kill the mutant, the test suite failed.

5.4. Results and Analysis

5.4.1. Feasibility For the *product ordering* system, we extracted the collection of activity and transition entries through preprocessing, as shown in Tables I and II, respectively. We then used ConcurTester to generate test scenarios. Table III shows four test scenarios generated according to the weak coverage criterion. Finally, we generated test data for each scenario.

[¶]Their detailed descriptions are available on the website: http://cs.gmu.edu/˜offutt/mujava/

Table I. The collection of activity entries

ID	responseID	noOutTransitions	type	name
0	1	1	start	noname
1	2	1	loop	Order_Product
2	2	2	branch	noname
3	8	2	fork	noname
4	6	1	loop	Getting_Billing_Information
5	6	1	action	Validating_Billing_Information
6	6	2	branch	noname
7	8	1	action	Getting_Shipping_Information
8	9	1	join	noname
9	12	1	action	Provide_Receipt
10	12	1	action	Assemble_Order
11	12	1	action	Ship_Order
12	12	0	end	noname

Table II. The collection of transition entries

iID	oID
0	1
1	2
2	1
2	3
3	4
3	7
4	5
5	6
6	4
7	8
6	8
8	9
9	10
10	11
11	12

Table III. Generated Test scenarios using weak coverage criterion

No.	Test Scenario
1	noname -> Order_Product -> Getting_Billing_Information -> Validating_Billing_Information -> Getting_Shipping_Information -> Provide_Receipt -> Assemble_Order -> Ship_Order
2	noname -> Order_Product -> Order_Product -> Getting_Billing_Information -> Validating_Billing_Information -> Getting_Shipping_Information -> Provide_Receipt -> Assemble_Order -> Ship_Order
3	noname -> Order_Product -> Getting_Billing_Information -> Validating_Billing_Information -> Getting_Billing_Information -> Validating_Billing_Information -> Getting_Shipping_Information -> Provide_Receipt -> Assemble_Order -> Ship_Order
4	noname -> Order_Product -> Order_Product -> Getting_Billing_Information -> Validating_Billing_Information -> Getting_Billing_Information -> Validating_Billing_Information -> Getting_Shipping_Information -> Provide_Receipt -> Assemble_Order -> Ship_Order

5.4.2. Fault detection capability We next analyze the fault detection capability of the test suite generated using the proposed approach. In order to study the impact of the test suite size on the effectiveness of the proposed approach, we varied the size to be 4, 20, 40, and 60, which corresponds to 1, 5, 10, and 15 test cases per scenario, respectively. We further compare the fault detection effectiveness of our proposed approach with that of random testing, comparing their *MS* scores for the same numbers of test cases.

Table IV presents the *MS* results for each test scenario, displayed according to "Method-level" and "Class-level". Note that when the evaluation results for different test suite sizes are the same, we merge the evaluation results into one column in order to reduce the redundancies. For instance, for method-level faults, test suites with a size of 20, 40, and 60 have the same mutation scores, we show the evaluation results in the "size=20/40/60" row. This reduction rule was also applied to the other tables in this section.

From the table, we can observe that: (i) for both method-level and class-level faults, the generated test suite shows a good fault-detection effectiveness, i.e. the test cases generated using the weak coverage criterion were able to detect more than 45.7% of the method-level faults, and were able to detect more than 73.3% of the class-level faults; (ii) the test suites derived for different test scenarios have a different fault detection capability; (iii) the detection rates of class-level faults appear higher than those of method-level faults; and (iv) the generated test suite size appears to have a slight impact on the fault detection rates of method-level faults. For class-level faults, test suites of different sizes have the same mutation scores, while for method-level faults, test suites whose sizes are 20, 40, and 60 have the same mutation scores, and one test suite of size 4 (namely one test case per test scenario) has slightly lower mutation scores. This further indicates that our approach does not need a large number of test cases for each scenario.

Table IV. The *MS* results using the weak concurrency coverage criterion for each test scenario

Level	Number of Test Cases	Test Scenario 1	Test Scenario 2	Test Scenario 3	Test Scenario 4
Method-level	size=4	51.4%	45.7%	56.4%	54.3%
	size=20/40/60	56.4%	54.3%	63.6%	61.4%
Class-level	size=4/20/40/60	73.3%	73.3%	86.7%	86.7%

Table V presents the *MS* results of both our approach and the random approach, according to "Method-level" and "Class-level", and also overall ("Total"). From the table, we can observe that: (i) the test suites generated by our approach using the weak coverage criterion can detect 100% of non-equivalent class-level faults, regardless of suite size. One test suite of size 4 generated by the random approach can detect only 73.3% of non-equivalent class-level faults, while it can detect 100% class-level faults when suite sizes are 20, 40, and 60; (ii) the test suite whose size is 4 generated by our approach can detect 75.7% of non-equivalent method-level faults, giving an overall detection rate of 80% for all seeded faults, whereas the test suite of size 4 generated by the random approach can detect only 34.3% of non-equivalent method-level faults, giving an overall detection rate of 41.2% for all seeded faults; (iii) test suites whose sizes are 20, 40, and 60 generated by our approach can detect 80% of non-equivalent method-level faults, giving an overall fault detection rate of 83.5% for all seeded faults, indicating that increasing the size of test suits composed of randomly generated test cases for each test scenario does not always improve the fault detection rates. Test suites whose sizes are 20, 40, and 60 generated by the random approach can detect 69.3% of non-equivalent method-level faults, giving an overall fault detection rate of 74.7% for all seeded faults; and (iv) for the same sizes, test suits generated by our approach achieve higher mutation scores than those achieved by the random approach, with the differences being more prominent when the size is small (with a size of 4, their mutation scores are 80% and 41.2%, respectively).

Tables VI and VII report the mutation scores for the method-level and class-level mutation operators, respectively. Accordingly, we have the following observations:

- For AORB (replace basic binary arithmetic operators with other binary arithmetic operators), AOIU (replace basic unary arithmetic operators with other unary arithmetic operators), and

Table V. The mutation score *MS* results of our approach and the random approach

Number of Test Cases	Level	Number of Total Mutants	Our Approach		Random Approach	
			Number of Killed Mutants	Mutation Score (*MS*)	Number of Killed Mutants	Mutation Score (*MS*)
Size=4	Method-level	140	106	75.7%	48	34.3%
	Class-level	30	30	100%	22	73.3%
	Total	170	136	80%	70	41.2%
Size=20/40/60	Method-level	140	112	80%	97	69.3%
	Class-level	30	30	100%	30	100%
	Total	170	142	83.5%	127	74.7%

COI (insert unary conditional operators) method-level mutants, the mutation scores (*MS*) were 100% when our approach was used, which means that the test suite generated using the proposed approach can detect all of these types of faults.

- For ROR (replace relational operators with other relational operators, and replace the entire predicate with true and false) and LOI (insert unary logical operator) method-level mutants, although the *MS* was less than 100%, it was relatively high, indicating that these faults were relatively easily detected using the proposed approach.
- For AOIS (insert short-cut arithmetic operators) and COR (replace binary conditional operators with other binary conditional operators) method-level mutants, the *MS* was low, which suggests that these types of faults were harder to detect, and the test suite generated by the proposed approach cannot detect all such faults. In addition, there were more equivalent mutants in the AOIS type, which further indicates that this type of fault should be treated carefully when designing test cases and doing test specifications.
- For PRV (replace reference assignment with other comparable variable), JTI (this keyword insertion), JTD (this keyword deletion), JSI (static modifier insertion), and JID (member variable initialization deletion) class-level mutants, the mutation scores were 100%, which indicates that these types of faults were relatively easy to detect. In particular, the detection rates for all class-level mutants were 100%, which suggests that the proposed approach is very effective for such mutants.
- In the same settings, our approach outperforms random testing. For method-level mutants, the mutation scores of our approach are much greater than those for random testing when the size is 4; and slightly greater than or equal when the size is 20, 40, and 60. For class-level mutants, test suites whose size is 4 generated by random testing cannot guarantee 100% detection of JTI and JTD mutants, while test suites generated by our approach can detect 100% of all these kinds of mutants, regardless of suite size.

We have reported the fault detection capability of the test suite generated using the weak concurrency coverage criterion. When the moderate concurrency coverage criterion is used, more test scenarios are generated for *product ordering*, but the current version of muJava does not have concurrency-specific mutation operators. Furthermore, test suites generated with the moderate concurrency coverage criterion and the weak concurrency coverage criterion should have similar fault detection capability because: (i) test suites generated using these two coverage criteria cover the same basic paths; (ii) although the moderate concurrency coverage criterion requires coverage of more combinations of parallel processes, which is beneficial to detect concurrency-specific faults such as data races, such faults are not included in those generated using muJava. For these reasons, in this paper we only evaluate the fault detection capability of the test suite generated using the weak concurrency coverage criterion. Although deadlock is a risk, and an important issue for concurrent systems, because the faults in our experiments were simulated by MuJava-generated mutation operators, and MuJava is not designed for deadlock faults, no deadlocks were included for evaluation.

Table VI. The mutation scores *MS* for the method-level mutants using our approach and the random approach

Number of Test Cases	Mutation Operators	Number of Total Mutants	Number of Equivalent Mutants	Our Approach: Number of Killed Mutants	Our Approach: Mutation Score (*MS*)	Random Approach: Number of Killed Mutants	Random Approach: Mutation Score (*MS*)
Size=4	AOIS	80	27	30	56.6%	2	0.04%
	COR	4	0	2	50%	2	50%
	ROR	33	3	23	76.7%	19	63.3%
	LOI	24	0	22	91.7%	10	41.6%
	AORB	4	0	4	100%	0	0%
	AOIU	16	0	16	100%	8	50%
	COI	9	0	9	100%	9	100%
Size=20/40/60	AOIS	80	27	35	66%	21	39.6%
	COR	4	0	2	50%	2	50%
	ROR	33	3	24	80%	23	76.7%
	LOI	24	0	22	91.7%	22	91.7%
	AORB	4	0	4	100%	4	100%
	AOIU	16	0	16	100%	16	100%
	COI	9	0	9	100%	9	100%

Table VII. The mutation scores *MS* for the class-level mutants using our approach and the random approach

Number of Test Cases	Mutation Operators	Number of Total Mutants	Number of Equivalent Mutants	Our Approach: Number of Killed Mutants	Our Approach: Mutation Score (*MS*)	Random Approach: Number of Killed Mutants	Random Approach: Mutation Score (*MS*)
Size=4	PRV	8	0	8	100%	8	100%
	JTI	12	1	11	100%	7	63.6%
	JTD	12	1	11	100%	7	63.6%
	JSI	14	14	0	100%	0	100%
	JID	1	1	0	100%	0	100%
Size=20/40/60	PRV	8	0	8	100%	8	100%
	JTI	12	1	11	100%	11	100%
	JTD	12	1	11	100%	11	100%
	JSI	14	14	0	100%	0	100%
	JID	1	1	0	100%	0	100%

5.5. *Threats to Validity*

Through this case study, we have validated the feasibility and effectiveness of the proposed approach. The experimental results show that, even using the weak concurrency coverage criterion, the test suite generated using our approach can detect more than 80% of seeded faults with a very small size of test suite (one test case per scenario). Furthermore, more than 75.7% of method-level faults and 100% of class-level faults can be detected by the generated test cases. For the same situations, our approach achieved a higher mutation score than random testing. These results indicate that the proposed approach is both effective and efficient.

Our study may face the following threats to validity. One threat is related to the subject program: although it would be good to have a large number of benchmarks for the experiments, to the best of our knowledge, there are currently none available. Developing such benchmarks would be very expensive in both time and labor, and is thus infeasible. In our study, we developed a subject program which is relatively small in size, but which covered most major UAD features.

Another possible threat to validity is related to how the experiments were designed: In our study, mutation operators were used to simulate possible faults. Although mutation analysis has been widely used to evaluate the effectiveness of various testing techniques [2], the mimicked faults (mutants) are possibly different from the real-life faults. Finally, we have so far only evaluated the fault detection effectiveness of the weak concurrency coverage criterion, which may affect the conclusive effectiveness of the analysis. In future work, we will look at other concurrency coverage criteria.

6. RELATED WORK

The UML has becomes a standard visual modelling language, providing three categories of diagrams for modeling different aspects of a system. Research has been conducted into how to test systems under development based on different UML diagram specifications, and into how to develop various test techniques [8], including generating test cases from Class Diagrams [5][32], State Diagrams [3][7][13][17][19][26][36][40], Sequence/Collaboration Diagrams [27][38], Use Case Diagrams [14], Activity Diagrams [33][34], and combinations of Use Cases and State Charts [30]. There has also been interest in generating test cases from UML state machine diagrams [23], and from activity diagrams [12].

An important issue in testing based on UADs relates to generating test scenarios, which is usually a manual and time-consuming task. Much effort has been put into developing various methods for automatically generating test scenarios from UADs. In our previous work [42], we developed a three-layer framework for automated test case generation from UAD specifications, according to which a UAD is first transformed into a test outline model, from which a set of test outlines are generated. Then, based on input data and the generated test outlines, a test case model is developed, leading to a set of test cases being generated. An important contribution of this work was to propose a set of transformation rules for each type of activity, providing a sound and convenient basis for the development of test cases generation algorithms [33]. We developed a tool, TCaseUML, which extracts the UAD specifications from Rational Rose, and generates test cases in terms of each activity. However, it was not clear how test cases could be effectively generated for test scenarios from the transformed test outline model.

Liu et al. [21] proposed a set of structural coverage criteria for scenario-oriented testing of UAD specifications. The proposed coverage criteria require that the test scenarios generated from UADs should cover activities, transitions, paths, and typical values of branch activities. Although these criteria are useful when generating tests from UADs, their actual implementation has not been discussed. In our previous work [34], we developed a recursive algorithm to generate test scenarios which are able to satisfy basic path coverage criteria, and a tool, TSGen, to automate the algorithm. The presented algorithm and tool were illustrated with two case studies, but no experiments reported on the fault-detection effectiveness of the coverage criterion.

Li and Lam [20] proposed using so-called anti-ant-like agents to automatically generate test threads from UADs, an approach suggesting the potential to automate test scenario generation. However, this approach has some shortcomings, such as redundant exploration of UADs (hence reducing the efficiency of the generation process), and limited treatment of complex UAD structures, such as fork and join activities. To overcome such limitations, Xu et al. [39] proposed an automated approach to directly generate test scenarios from UADs using adaptive agents. Their approach is capable of dealing with UADs containing more complicated structures, and an algorithm and supporting tool were described. Unfortunately, however, no experiments into the effectiveness of the approach have been reported.

Wang et al. [37][41] proposed a gray box-based approach to generating test scenarios from UADs, and generating test cases by extracting the information from such test scenarios. The test scenarios are generated in terms of basic paths, and the test scenario generation algorithm is based on the Petri net model [31]. Since there are many parallel and conditional behavior items in UADs, it can be difficult to directly define the basic path from the original UAD. The presented algorithm does not address the concurrency issue, an essential component in UADs.

Chen et al. [4] proposed an automatic method to generate test cases when UADs are used as design specifications. Their approach first randomly generates a large number of test cases, and then obtains program execution traces by running the program with those test cases. Finally, some redundant test cases are pruned by comparing these traces with the UAD according to a specific coverage criterion, resulting in a reduced, but adequate, test case set. The test adequacy criteria include activity coverage, transition coverage and simple path coverage. In order to generate tests to meet a specific coverage criterion, their approach needs to execute the program and retrieve the execution traces by means of instrumentation. Their approach is very expensive (i.e. multiple executions), and has difficulty when there are inconsistencies between the implementation and the design specification. Furthermore, it is not clear how test cases are selected for concurrent threads in a program under test.

Kim et al. [16] proposed a transformation-based method to generate test cases from UADs. Their method first builds an I/O explicit activity diagram from an ordinary UAD, and then transforms it into 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 transformation, but they differ in that our transformation rules were developed based on activity types instead of I/O flows. Furthermore, our approach generates controllable test scenarios satisfying the specific concurrence coverage criterion, while their approach is based on the single stimulus principle [15], which is used to deduce the number of test cases.

Kundu and Samanta [18] proposed a conversion-based approach to generating test cases using UADs. In their approach, a set of conversion rules were proposed to map UAD elements to nodes in a graph model. These rules are quite similar to the transformation rules that we previously proposed [42]. They proposed an algorithm to generate test scenarios satisfying the activity coverage criterion, but how fork and join activities were handled was not discussed. It was claimed [18] that the generated test suite based on the activity path coverage criterion was able to uncover more synchronization and loop faults than existing work, however no evaluation experiments were reported.

Khandai et al. [24] proposed generating test cases from the combination of UADs and Sequence Diagrams. Their approach assumes that UADs and Sequence Diagrams (SDs) are used to model a system. It converts UADs into Activity Graphs (AGs) and SDs into Sequence Graphs (SGs), and then combine two to form Activity Sequence Graphs (ASGs). Finally, an algorithm can be developed to traverse the ASG to generate test cases. Their approach requires that both SDs and UADs be used for modeling a system, while our approach eases this constraint. Furthermore, the issue of how to combine the AG and SG into an ASG, especially when there are inconsistencies or mismatches between SDs and UADs, is not clearly discussed, nor is an algorithm presented for how to generate test cases from the resulting ASG. Lastly, no supporting tools or evaluation experiments were reported.

An important issue relates to handling concurrency when generating test scenarios from UADs, something ignored in some existing approaches, which usually focus on presenting some coverage criteria or algorithms for generating test scenarios to satisfy a given coverage criterion. In this paper, we propose a transformation-based scenario-oriented approach to generating test cases from UADs. Our approach provides a comprehensive treatment of concurrent and loop activities. Our approach is built on transformation rules developed in our previous work [42], which we have extended to enable loop processing. We further developed algorithms and a tool to support the automation of the proposed approach, and validated the effectiveness of the proposed approach in a case study, which, to the best of our knowledge, is the first attempt to test a real-life application with test cases generated from UADs. Finally, we briefly compare our work with other related work, as shown in Table VIII. From the table, we can observe that our approach provides a comprehensive treatment for dealing with concurrency elements, has tooling support, and its fault-detection effectiveness is validated in a case study.

Table VIII. A brief comparison of our approach with related work

Approach	Deals with concurrency	Tooling support	Fault-detection effectiveness evaluation
[42]	Yes	Yes	No
[21]	No	No	No
[20]	Yes	No	No
[39]	Yes	Yes	No
[37][41]	No	Yes	No
[4]	No	Yes	No
[16]	Yes	No	No
[18]	No	No	No
[24]	No	No	No
Our approach	Yes	Yes	Yes

7. CONCLUSIONS AND FUTURE WORK

We have presented a transformation-based approach to generate scenario-oriented test cases from UAD specifications, focusing mainly on the testing of concurrent activities. The approach employs and extends a set of transformation rules to convert a UAD specification into a well-formed intermediate representation, and thereby helps address challenges caused by fork and join concurrency in the UAD, and helps reduce invalid test scenarios. Algorithms have been developed to generate test scenarios from the intermediate representation. The approach supports different coverage criteria, and can therefore test concurrent processes quite effectively. Finally, we have implemented a tool to automate the proposed approach, and conducted a case study to validate its feasibility and effectiveness.

The presented approach is a kind of model-driven testing technique which allows testers to not only schedule the software test process earlier, but also better allocate the testing resources. With the proposed approach, testers can start test design in the design stage instead of having to wait until the coding stage. Furthermore, testing resources are often limited, requiring that the complexity and number of tests should be controllable, something that the proposed approach supports by automatically generating different sets of test scenarios to satisfy different concurrency coverage criteria. Therefore, the proposed approach is particularly useful for enhancing the testing efficiency of concurrent applications.

In our future work, we plan to extend the developed tool and integrate it as a plug-in for UML supporting tools, such as ArgoUML. We are interested in evaluating the proposed approach using more real-life, concurrent applications, such as multi-threaded Java programs. We would also like to further investigate and evaluate the fault detection effectiveness, and costs, of the proposed concurrency coverage criteria, including for deadlock faults.

ACKNOWLEDGEMENT

This research is supported by the National Natural Science Foundation of China (Grant Nos.: 61370061, 60903003), the Beijing Natural Science Foundation of China (Grant No. 4112037), the Fundamental Research Funds for the Central Universities (Grant No. FRF-SD-12-015A), and the Beijing Municipal Training Program for Excellent Talents (Grant No. 2012D009006000002).

REFERENCES

1. Ambysoft Inc. UML activity diagramming guidelines. http://www.agilemodeling.com/style/activityDiagram.htm, 2007.
2. J. H. Andrews, L. C. Briand, Y. Labiche. Is mutation an appropriate tool for testing experiments? Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), IEEE Computer Society, 2005, pp.402-411.

3. L. C. Briand, J. Cui, Y. Laboche. Towards automated support for deriving test data from UML statecharts. Proceedings of 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML 2003), Lecture Notes in Computer Science 2863, pp.249-264.
4. M. S. Chen, X. K. Qiu, X. D. Li. Automatic test case generation from UML activity diagram. Proceedings of the 2006 International Workshop on Automation of Software Test. pp.2-8.
5. H. Y. Chen. An approach for OO cluster-level tests based on UML. Proceedings of IEEE International Conference on the Systems, Man and Cybernetics (SMC 2003), IEEE Computer Society, 2003, pp.1064-1068.
6. T. Y. Chen, P. L. Poon, T. H. Tse. A choice relation framework for supporting category-partition test case generation. IEEE Transactions on Software Engineering, 2003, 29(7):577-593.
7. P. Chevalley, P. T. Fosse. Automated generation of statistical test cases from UML state diagrams. Proceedings of 25th Annual International Computer Software and Applications Conference(COMPSAC 2001), IEEE Computer Society, 2001, pp.205-214.
8. Z. R. Dai. Model-Driven Testing with UML 2.0. Technical Report, Computer Science at Kent, 2004, pp.179-187.
9. S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, B. M. Horowitz. Model-based Testing in Practice. Proceedings of International Conference on Software Engineering (ICSE 1999), IEEE Computer Society, 1999, pp.285-294.
10. R. A. DeMillo, R. J. Lipton, F. G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 1978, 1(4):31-41.
11. M. Fowler, K. Scott. Activity Diagrams. http://www.sts.tu-harburg.de/projects/UML/Activity Diagrams.pdf, pp.151-164.
12. H. M. Gao, D. Xu, Z. T. Liu. Test study of UML activity diagram. Journal of Computer Science, 2008, 35(2):263-281.
13. J. Hartmann, C. Imoberdof, M. Meisenger. UML-Based Integration Testing. Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA 2000), 2000, pp.60-70.
14. IBM Center for Software Engineering. Use Case Based Testing. http://www.research.ibm.com/softeng/testing/ucbt.htm
15. S. Kang, J. Shin, M. Kim. Interoperability test suite derivation for communication protocols. Computer Networks, 2000, 32(3):347-364.
16. H. Kim, S. Kang, J. Baik, I. Ko. Test cases generation from UML activity diagrams. Proceedings of Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing(SNPD 2007), Volumn 3, 2007, pp.556-561.
17. Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, S. D. Cha. Test cases generation from UML state diagrams. IEEE Software, 1999, 46(4):187-192.
18. D. Kundu, D. Samanta. A novel approach to generate test cases from UML activity diagrams. Journal of Object Technology, 2009, 8(3):65-83.
19. L. Y. Li, Z. C. Qi. Test selection from UML statecharts. Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems (TOOLS'99), IEEE Computer Society, 1999, pp.273-281.
20. H. Li, C. P. Lam. Using anti-ant-like agents to generate test threads from the UML diagrams. Proceedings of the 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems (TESTCOM 2005), LNCS 3502, 2005, pp.69-80.
21. M. Liu, M. Z. Jin, C. Liu. Automated test scenarios generation based on UML activity diagram model. Journal of Computer Engineering and Applications, 2002, 28(12):122-124.
22. Y. S. Ma, J. Offutt, Y. R. Kwon. MuJava: an automated class mutation system. Software Testing, Verification and Reliability, 2005, 15(2):97-133.
23. M. Aggarwal, S. Sabharwal. Test case generation from UML state machine diagram: A survey. Proceedings of Third International Conference on Computer and Communication Technology (ICCCT 2012), IEEE Computer Society, 2012, pp.133-140.
24. M. Khandai, A. A. Acharya, D. P. Mohapatra. Test Case Generation for Concurrent System using UML Combinational Diagram. International Journal of Computer Science and Information Technologies, 2011, 2(3):1172-1181.
25. L. de Moura, N. Bjørner. Z3: An Efficient SMT Solver. Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008), Lecture Notes in Computer Science Volume 4963, 2008, pp.337-340.
26. J. Offutt, A. Abdurazik. Generating tests from UML specifications. Proceedings of 2nd International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML'99), 1999, pp.416-429.
27. J. Offutt, A. Abdurazik. Using UML collaboration diagrams for static checking and test generation. Proceedings of 3rd International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML'00), 2000, pp.383-395.
28. Object Management Group. UML Specification (v1.5). http://www.omg.org/uml, March 2003.
29. 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.
30. M. Riebisch, I. Philippow, M. Gätze. UML-Based Statistical Test Case Generation. Proceedings of International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World (NODe 2002), Lecture Notes in Computer Science 2591, pp.394-411.
31. G. Rozenburg, J. Engelfriet. Elementary Net Systems Lectures on Petri Nets I: Basic Models - Advances in Petri Nets. Lecture Notes in Computer Science 1491, Springer, 1998, pp.12-121.
32. M. Scheetz, A. Mayrhauser, R. France, E. Dahlman, A. E. Howe. Generating test cases from an OO model with an AI planning system. Proceedings of 10th International Symposium on Software Reliability Engineering (ISSRE99), IEEE Computer Society, 1999, pp.250-259.
33. C.-A. Sun. A transformation-based approach to generating scenario-oriented test cases from UML activity diagrams for concurrent applications. Proceedings of 32nd Annual IEEE International Computer Software and Application Conference (COMPSAC 2008), IEEE Computer Society, 2008, pp.160-167.

34. C.-A. Sun, B. Zhang, J. Li. TSGen:A UML activity diagram-based test scenario generation tool. Proceedings of 2009 IEEE/IFIP International Symposium on Trusted Computing and Communications (TrustCom 2009), IEEE Computer Society, 2009, pp.853-858.
35. C.-A. Sun, G. Wang, K.-Y. Cai, T. Y. Chen. Distribution-aware mutation analysis. Proceedings of 9th IEEE International Workshop on Software Cybernetics (IWSC 2012), IEEE Computer Society, 2012, pp.170-175.
36. M. Vieira, D. J. Richardson. Object-Oriented Specification-Based Testing Using UML Statechart Diagrams. Proceedings of First Workshop on Automated Program Analysis, Testing, and Verification held in conjunction with the 22nd International Conference on Software Engineering (ICSE 2000), IEEE Computer Society, 2000, pp.101-105.
37. L. Wang, J. Yuan, X. Yu, J. Hu, X. D. Li, G. L. Zheng. Generating test cases from UML activity diagram based on gray-box method. Proceedings of 11th Asia-Pacific Software Engineering Conference (APSEC 2004), IEEE Computer Society, 2004, pp.284-291.
38. Y. Wu, M. H. Chen, J. Offutt. UML-based integration testing for component-based software. Proceedings of Second International Conference on COTS-Based Software Systems (ICCBSS 2003), Lecture Notes in Computer Science 2580, 2003, pp.251-260.
39. D. Xu, H. Li, C. P. Lam. Using adaptive agents to automatically generate test scenarios from the UML activity diagrams. Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005), IEEE Computer Society, 2005, pp.385-392.
40. J. Yan, J. Wang, H. W. Chen. Deriving software statistical testing model from UML model. Proceedings of Third International Conference on Quality Software (QSIC 2003), IEEE Computer Society, 2003, pp.343-351.
41. J. S. Yuan, L. Z. Wang, X. D. Li, G. L. Zheng. UMLTGF: A tool for generating test cases from UML activity diagrams based on grey-box method. Journal of Computer Research and Development, 2006, 43(1):46-53.
42. 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.