The Computer Journal Advance Access published August 25, 2007

doi:10.1093/comjnl/bxm057

UML Activity Diagram-Based Automatic Test Case Generation For Java Programs

MINGSONG CHEN, XIAOKANG QIU, WEI XU, LINZHANG WANG, JIANHUA ZHAO AND XUANDONG LI*

State Key Laboratory of Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210093, P.R.China
**Corresponding author: lxd@nju.edu.cn*

Test case generation based on design specifications is an important part of testing processes. In this paper, Unified Modeling Language activity diagrams are used as design specifications. By setting up several test adequacy criteria with respect to activity diagrams, an automatic approach is presented to generate test cases for Java programs. Instead of directly deriving test cases from activity diagrams, this approach selects test cases from a set of randomly generated ones according to a given test adequacy criterion. In the approach, we first instrument a Java program under testing according to its activity diagram model, and randomly generate abundant test cases for the program. Then, by running the instrumented program we obtain the corresponding program execution traces. Finally, by matching these traces with the behavior of the activity diagram, a reduced set of test cases are selected according to the given test adequacy criterion. This approach can also be used to check the consistency between the program execution traces and the behavior of activity diagrams.

Keywords: Software testing; test cases generation; UML activity diagrams; Java

Received 24 September 2006; revised 9 June 2007

1. INTRODUCTION

Testing is an important part of quality assurance in the software life-cycle. As the complexity and the size of software systems grow, more and more time and manpower are required for testing. Manual testing is so labor-intensive and error-prone that it is necessary to employ automatic testing techniques in some circumstances.

The unified modeling language (UML) is a standard visual modelling language that is designed to specify, visualize, construct and document the artifacts of software systems [1, 2]. Since UML became a standard of OMG in 1997, UML models have become a main class of artifacts in software development processes. UML provides a number of diagrams to describe different aspects of software artifacts. UML activity diagrams describe the sequential or concurrent control flows of activities. They can be used to model the dynamic aspects of a group of objects, or the control flow of an operation, which form a kind of design specifications for programs.

In this paper, we use UML activity diagrams as design specifications, and consider test case generation for Java programs. The state of arts of UML model-based test case generation mainly focuses on generating test cases directly from various UML models by applying specification-based testing and/or code-based testing approaches [3–10]. However, those direct approaches are hardly implemented in a fully automatic fashion because of the following reasons. First, only abstract test cases can directly be generated from the models, which specify the system under constructing. They cannot be used directly in program testing without manual concretization. Second, for dynamic models, the test cases are generated by traversing the paths of models to derive the test scenarios. The loops and branches in the models make the path conditions very complicated. It lead to algorithms with high complexity, even undecidable problems.

In this paper, instead of directly deriving test cases from UML activity diagrams, we present an indirect approach which selects test cases from a set of randomly generated ones according to a given coverage criterion concerning the activity diagram specification. In the approach, we first instrument the Java program under testing according to its activity diagram specification, and randomly generate abundant test

cases for the program. Then, by running the program with the generated test cases, we can obtain the corresponding program execution traces. At last, by matching those program execution traces with the behavior of the activity diagram, we can select a reduced set of test cases according to a specific test adequacy criterion. The approach can also be used to check the consistency between the program execution traces and the behavior of activity diagrams.

The paper is organized as follows. In Section 2, we introduce the UML activity diagrams and the related notations. In Section 3, the approach of automatic test case generation for Java programs is described in detail. The related works are discussed in Section 4, and some conclusions are given in Section 5.

2. UML ACTIVITY DIAGRAMS

2.1. Notations

As opposed to other diagrams in UML, an activity diagram extracts the core idea from flowcharts, state transition graphs and Petri nets [1, 2]. An activity diagram contains activity states, which represent the execution of a statement in a procedure or the performance of an activity in a workflow. Instead of waiting for an event, as in a normal wait state, an activity state waits for the completion of its computation. When the activity completes, the execution proceeds to the next activity state within the diagram. A completion transition in an activity diagram fires when the preceding activities are complete. An activity diagram may contain branches, as well as forking of control into concurrent threads. Concurrent threads represent activities that can be performed concurrently by different objects or persons in an organization.

In an activity diagram, an activity state is shown as a box with rounded ends containing a description of the activity; simple completion transitions are shown as arrows; branches are shown as guard conditions on transitions or as diamonds with multiple labeled exit arrows; fork or join of control is shown by multiple arrows leaving or entering a heavy synchronization bar. For example, Fig. 1 shows a simple activity diagram, which consists of most elements to describe a workflow or an operation.

The recent major revision of UML2.0 has introduced significant changes and additions [2]. Compared with UML1.x, the concrete syntax of activity diagrams remains mostly the same, but the abstract syntax and semantics have changed drastically. In UML1.x, the activity diagrams were defined as a kind of state machine diagrams. Now, there is no connection between these two notations. The meaning of activity diagrams is explained in terms of Petri net notion [11] such as the token, flow, edge weight, etc. In this paper, in accordance with UML2.0, we adopt the Petri net-like semantics of the activity diagrams, and formalize an activity diagram as follows.

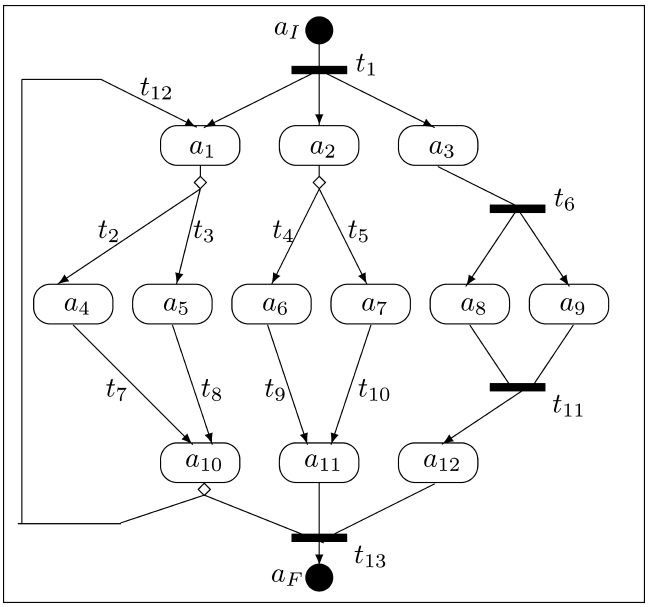

FIGURE 1. A simple activity diagram.

DEFINITION 2.1. *An activity diagram $\mathcal{D}$ is a tuple, $\mathcal{D} = (A, T, F, a_I, a_F)$, where*

(i) $A = \{a_1, a_2, \ldots, a_m\}$ *is a finite set of activity states*;
(ii) $T = \{t_1, t_2, \ldots, t_n\}$ *is a finite set of completion transitions*;
(iii) $F \subset (A \times T) \cup (T \times A)$ *is the flow relation*;
(iv) $a_I \in A$ *is the initial activity state, and* $a_F \in A$ *is the final activity state; there is only one transition t such that* $(a_I, t) \in F$, *and for any* $t_1 \in T$, $(t_1, a_I) \notin F \wedge (a_F, t_1) \notin F$.

A state μ of $\mathcal{D}$ is a subset of A. For any transition, $t \in T$, let ${}^\bullet t = \{a \in A | (a, t) \in F\}$ and $t^\bullet = \{a \in A | (t, a) \in F\}$, *which denote the preset and postset of t, respectively. A transition t is enabled in a state μ if* ${}^\bullet t \subseteq \mu$; *otherwise, it is disabled. Let enabled(μ) be the set of transitions enabled in μ.*

In this paper, we consider an activity diagram as a design specification, which describes the workflow of a Java program. Each activity state in the activity diagram is interpreted as the execution of a method in the Java program. For any activity state a, we let $\xi(a)$ denote the corresponding method, and $\zeta(a)$ denote the class to which $\xi(a)$ belongs.

DEFINITION 2.2. *Let $\mathcal{D} = (A, T, F, a_I, a_F)$ be an activity diagram. A transition $t \in T$ may fire from state μ if and only if $t \in enabled(\mu)$ and $(\mu - {}^\bullet t) \cap t^\bullet = \emptyset$, and the new state μ' is given as $\mu' = (\mu - {}^\bullet t) \cup t^\bullet$, which is denoted by $\mu' = fire(\mu, t)$.*

The behavior of an activity diagram is described in terms of *runs*.

DEFINITION 2.3. *Let $\mathcal{D} = (A, T, F, a_I, a_F)$ be an activity diagram. A run segment ρ of $\mathcal{D}$ is a sequence of states and transitions*

$$\rho = \mu_0 \xrightarrow{t_0} \mu_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} \mu_n,$$

where $\mu_0 = \{a_I\}$ and $\mu_i = fire(\mu_{i-1}, t_{i-1})$ for any i $(1 \le i \le n)$. If $\mu_n = \{a_F\}$, then the run segment ρ is a run.

For example, for the activity diagram depicted in Fig. 1, $\{a_I\} \xrightarrow{t_1} \{a_1, a_2, a_3\} \xrightarrow{t_2} \{a_2, a_3, a_4\} \xrightarrow{t_4} \{a_3, a_4, a_6\} \xrightarrow{t_6} \{a_4, a_6, a_8, a_9\} \xrightarrow{t_7} \{a_6, a_8, a_9, a_{10}\} \xrightarrow{t_9} \{a_8, a_9, a_{10}, a_{11}\} \xrightarrow{t_{11}} \{a_{10}, a_{11}, a_{12}\} \xrightarrow{t_{13}} \{a_F\}$ is a run.

2.2. Paths and trails

In this paper, we are focused on generating test cases for Java programs specified by UML activity diagrams. We thus need to consider the test adequacy criteria with respect to activity diagrams. These criteria mainly deal with the test coverage of elements and behavior of a given activity diagram. In a Java program with an activity diagram as its specification, the execution orders of the concurrent methods in different threads, which correspond to the firing orders of the transitions during the run of the activity diagram, are independent of the program inputs. It means that for a given input, the different program executions may result in the different program execution traces, which indicates that the run coverage in an activity diagram is hardly incarnated in the test adequacy criteria. Thus, for defining the test adequacy criteria, instead of runs we introduce *paths* in activity diagrams as follows.

Intuitively, the *paths* describe the behavior of an activity diagram as if all the transitions separately enabled in a state fire at the same time during the activity diagram execution. For an activity diagram, all the transitions separately enabled in a state form a *concurrent transition*.

DEFINITION 2.4. *Let $\mathcal{D} = (A, T, F, a_I, a_F)$ be an activity diagram. For a state μ in $\mathcal{D}$, a concurrent transition τ is the set of the transitions $t_1, t_2, \ldots, t_m$ such that*

(i) *for any i $(1 \le i \le m)$, $t_i \in enabled(\mu)$;*
(ii) *for any i, j $(1 \le i < j \le m)$, ${}^\bullet t_i \cap {}^\bullet t_j = \emptyset$; and*
(iii) *for any $t \in (enabled(\mu) - \{t_1, t_2, \ldots, t_m\})$, there is t_i $(1 \le i \le m)$ such that ${}^\bullet t \cap {}^\bullet t_i \ne \emptyset$.*

The firing of the concurrent transition τ consists of the firing of t_i $(1 \le i \le m)$, and the new state μ' is given as $\mu' = \bigcup_{i=1}^{m} ((\mu - {}^\bullet t_i) \cup t_i^\bullet)$, which is denoted by $\mu' = fire(\mu, \tau)$.

Notice that there may be more than one concurrent transition for a state in an activity diagram.

DEFINITION 2.5. *Let $\mathcal{D} = (A, T, F, a_I, a_F)$ be an activity diagram. A path σ of $\mathcal{D}$ is a sequence of states and concurrent transitions*

$$\sigma = \mu_0 \xrightarrow{\tau_0} \mu_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{n-1}} \mu_n,$$

where $\mu_0 = \{a_I\}$, $\mu_n = \{a_F\}$ and $\mu_i = fire(\mu_{i-1}, \tau_{i-1})$ for any i $(1 \le i \le n)$. σ is a simple path if there is no execution repetition in σ, i.e. for any τ_i and τ_j $(0 \le i < j < n)$, for any $t \in \tau_i$ and $t' \in \tau_j$, ${}^\bullet t \cap {}^\bullet t' = \emptyset$.

For an activity diagram, a run is regarded intuitively as a linear execution of a path. Notice that a path could have many linear executions, which means that a path could correspond to many runs. For example, Fig. 2 shows four simple paths σ_1, σ_2, σ_3, σ_4 in the activity diagram depicted in Fig. 1. The run $\{a_I\} \xrightarrow{t_1} \{a_1, a_2, a_3\} \xrightarrow{t_2} \{a_2, a_3, a_4\} \xrightarrow{t_4} \{a_3, a_4, a_6\} \xrightarrow{t_6} \{a_4, a_6, a_8, a_9\} \xrightarrow{t_7} \{a_6, a_8, a_9, a_{10}\} \xrightarrow{t_9} \{a_8, a_9, a_{10}, a_{11}\} \xrightarrow{t_{11}} \{a_{10}, a_{11}, a_{12}\} \xrightarrow{t_{13}} \{a_F\}$ is one of the liner executions of σ_1.

For an activity diagram which models a program, because of the existence of loops we hardly generate test cases to cover its paths fully. It compels us to consider the coverage of simple paths, which contains no repetition. To consider the criteria covering the repetitions of executions and/or specifically selected executions, we need to introduce *trails* in activity diagrams, which are a sequence of the transitions enabled and firing one by one.

DEFINITION 2.6. *Let $\mathcal{D} = (A, T, F, a_I, a_F)$ be an activity diagram. A trail γ of $\mathcal{D}$ is a sequence of transitions of the form $\gamma = t_0 \to t_1 \to \cdots \to t_n$ where any $t_i \in T$ $(0 < i \le n)$ is such that ${}^\bullet t_i \cap t_{i-1}{}^\bullet \ne \emptyset$.*

$$\sigma_1 = \{a_I\} \xrightarrow{\{t_1\}} \{a_1, a_2, a_3\} \xrightarrow{\{t_2, t_4, t_6\}} \{a_4, a_6, a_8, a_9\} \xrightarrow{\{t_7, t_9, t_{11}\}} \{a_{10}, a_{11}, a_{12}\} \xrightarrow{\{t_{13}\}} \{a_F\}$$

$$\sigma_2 = \{a_I\} \xrightarrow{\{t_1\}} \{a_1, a_2, a_3\} \xrightarrow{\{t_2, t_5, t_6\}} \{a_4, a_7, a_8, a_9\} \xrightarrow{\{t_8, t_9, t_{11}\}} \{a_{10}, a_{11}, a_{12}\} \xrightarrow{\{t_{13}\}} \{a_F\}$$

$$\sigma_3 = \{a_I\} \xrightarrow{\{t_1\}} \{a_1, a_2, a_3\} \xrightarrow{\{t_3, t_4, t_6\}} \{a_5, a_6, a_8, a_9\} \xrightarrow{\{t_8, t_9, t_{11}\}} \{a_{10}, a_{11}, a_{12}\} \xrightarrow{\{t_{13}\}} \{a_F\}$$

$$\sigma_4 = \{a_I\} \xrightarrow{\{t_1\}} \{a_1, a_2, a_3\} \xrightarrow{\{t_3, t_5, t_6\}} \{a_5, a_7, a_8, a_9\} \xrightarrow{\{t_8, t_{10}, t_{11}\}} \{a_{10}, a_{11}, a_{12}\} \xrightarrow{\{t_{13}\}} \{a_F\}$$

FIGURE 2. Simple paths in the activity diagram in Fig. 1.

It is clear that for a trail γ of the form

$$\gamma = t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_n$$

if $t_i = t_j$ $(0 \leq i < j \leq n)$ then γ may corresponds to a repetition of execution. For example, for the activity diagram depicted in Fig. 1,

$$t_1 \rightarrow t_3 \rightarrow t_8 \rightarrow t_{12} \rightarrow t_2 \rightarrow t_7 \rightarrow t_{12}$$

is a trail.

3. APPROACH TO TEST CASE GENERATION FOR JAVA PROGRAMS

In this section, we give the details of the automatic approach to test case generation for Java programs with activity diagrams as design specifications. The approach first instruments a Java program under testing according to its activity diagram model, and randomly generates abundant test cases for the program. Then, by running the instrumented program, we obtain the corresponding program execution traces. At last, by matching these traces with the behavior of the activity diagram, we obtained a reduced set of test cases according to a specific test adequacy criterion.

3.1. Test adequacy criteria with respect to activity diagrams

Objective measurement of the test quality is one of the key issues in software testing. As an essential part of any testing method, a test adequacy criterion specifies the requirement of a particular testing [12]. In this paper, as we use activity diagrams as the design specifications for Java programs under testing, we need to set up the test adequacy criteria with respect to activity diagrams.

There have been several works [7, 10, 13] on the test adequacy criteria for UML static models, interaction models and state machine diagrams. Those criteria can be used for model testing or program testing. They mainly deal with the coverage of various elements and paths in UML models, and their basic ideas come from the traditional code coverage criteria. Like those works, the test adequacy criteria we consider here mainly deal with the coverage of elements and paths in a given activity diagram during the execution of a program under testing. The coverage is computed through matching the behavior of the activity diagram with the program execution traces. For an activity diagram, we set up the following four test adequacy criteria.

(i) *Activity coverage* requires that all activity states in the activity diagram be covered.
(ii) *Transition coverage* requires that all the transitions in the activity diagram be covered.
(iii) *Simple path coverage* requires that all the simple paths in the activity diagram be covered.
(iv) *Trail coverage* requires that all the given trails in the activity diagram be covered.

The activity coverage and transition coverage are basic coverage criteria and easily satisfied in testing, just like the statement coverage in the code coverage criteria. The simple path coverage is essentially based on a partial order of behavior of an activity diagram. It avoids the indeterminacy caused by concurrency, but does not cover any repetition of execution. The trail coverage deals with some special executions (including the repetitions of executions) in an activity diagrams which should be covered in testing, but the trails are usually picked out manually.

3.2. Program instrumenting

For a Java program under testing, we need to insert some statements into its source code for gathering the program execution traces. As each activity state in an activity diagram is interpreted as the execution of one method in a Java program, the program execution traces we gather are a sequence of events corresponding to method completions.

During the execution of a Java program, a class may have multiple instances. The same method of different instances may be invoked. This causes a trouble because we cannot identify which object's method in a program execution trace is corresponding to a given activity diagram. Therefore, we assume that any Java program under testing, specified by an activity diagram $\mathcal{D}$, is such that for any activity state a in $\mathcal{D}$, $\zeta(a)$ (i.e. the class with the method corresponds to a) has just one instance during the program execution.

Because we use the activity diagrams as the global behavior model of software systems, usually only high-level components of the system are concerned in such activity diagrams. Although most classes in an object-oriented software system may have many instances, which are created and destroyed dynamically, the high-level components are generally stable, and most of them are the only instance of their corresponding classes. Therefore the above assumption does not narrow the applicability of our approach much. In case that this assumption is not satisfied, our approach may still be applicable after rewriting the activity diagram with a coarser granularity or renaming the related classes in the program.

In a Java program, a method finishes its computation after the execution of its last statement. Thus, we insert the statements for gathering the information in the end of each related method definition. Given an activity diagram $\mathcal{D} = (A, T, F, a_I, a_F)$, for any $a \in A$, when its corresponding method $\xi(a)$ finishes its computation, the information we need to log includes the method $\xi(a)$ itself and the class $\zeta(a)$ which $\xi(a)$ is in. The instrumentation algorithm depicted in Fig. 3 runs as follows. First it scans the program and parses

```
scan the program, and parse the source code into a file
of tokens;
open the file of tokens;
read a token from the token file and assign it to
current_token;
while current_token ≠ eof do
  begin
    if current_token indicates a definition of
      method ξ(a) (a ∈ A)
    then insert the code segment
          Log_Finishing_Event after the last
          statement in the method definition;
    read an element from the token file and assign it
    to current_token;
  end;
return true.
```

Algorithm for instrumenting programs

```
try{ synchronized(this)
        java.io.RandomAccessFile receiveLog
          = new java.io.RandomAccessFile(log,"rw");
        receiveLog.seek(receiveLog.length());
        receiveLog.writeBytes
          (ξ(a) + (Object)this.getClassName());
        receiveLog.close(); }}
catch(Exception e){}
```

Code segment *Log_Finishing_Event*

FIGURE 3. Instrumentation algorithm and inserted code segment.

the source code into tokens. Then we check each token to recognize the related method definitions. For each method $\xi(a)$ ($a \in A$), we insert the code segment *Log_Finishing_Event* depicted in Fig. 3 after the last statement of the method. This code segment is used to log the execution information about the method and its class.

3.3. Matching program execution traces with activity diagram behavior

By running the program under testing with randomly generated test cases, we obtain a set of program execution traces. For selecting the test cases according to a given test adequacy criterion, we need to match these program execution traces with the dynamic behaviors of an activity diagram.

For a Java program, its execution traces we gather are a set of sequences of the method completion log items. These log items correspond to the activity state completions in a given activity diagram. Let $\mathcal{D} = (A, T, F, a_I, a_F)$ be an activity diagram, and

$$\rho = \mu_0 \xrightarrow{t_0} \mu_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} \mu_n$$

be a run of $\mathcal{D}$. The firing of t_i $(0 \le i \le n-1)$ means that all the activity states in the preset of t_i are completed. However, some activity states of $\mathcal{D}$ may be completed without firing any transition (e.g. a_8 or a_9 in the activity diagram depicted in Fig. 1). Therefore, the completions of such activity states may not be shown explicitly in a run of $\mathcal{D}$. For matching a program execution trace with a run in an activity diagram, we need to introduce the concept of *extended runs*, which show all activity state completions explicitly. This concept is formally defined as follows.

Let $\mathcal{D} = (A, T, F, a_I, a_F)$ be an activity diagram. An *extended state* of $\mathcal{D}$ is of the form (μ, υ) where μ is a state of $\mathcal{D}$, and $\upsilon \subseteq A$ is the set of the activity states in μ, which have been completed without firing any transition in *enabled*(μ). Since during the execution of $\mathcal{D}$ an activity state may be completed without firing any transition, we introduce a special transition $\lambda \notin T$ which represents that no transition in T fires. For any two extended states (μ, υ) and (μ', υ') of $\mathcal{D}$, we define

$$(\mu, \upsilon) \xrightarrow{t,a} (\mu', \upsilon')$$

if either there is an activity state $a \in \mu$ is completed, but no transition can be fired, or there is an activity state $a \in \mu$ is completed, and the corresponding transition t fires, i.e. one of the following two conditions is satisfied.

(i) $t = \lambda$, $a \in \mu$, $a \notin \upsilon$, $\mu' = \mu$, $\upsilon' = \upsilon \cup \{a\}$, and there is not any $t \in$ *enabled*(μ) such that ${}^\bullet t \subseteq \upsilon'$.

(ii) $t \in$ *enabled*(μ), $a \in {}^\bullet t$, $a \notin \upsilon$, ${}^\bullet t \subseteq \upsilon \cup \{a\}$, $\mu' =$ *fire*(μ, t) and $\upsilon' = \upsilon - {}^\bullet t$.

We define that an *extended run segment* ϱ of $\mathcal{D}$ is a sequence of the form

$$\varrho = (\mu_0, \upsilon_0) \xrightarrow{t_0,a_0} (\mu_1, \upsilon_1) \xrightarrow{t_1,a_1} \cdots \xrightarrow{t_{n-1},a_{n-1}} (\mu_n, \upsilon_n),$$

where (μ_i, υ_i) $(0 \le i \le n)$ is an extended state of $\mathcal{D}$ and $(\mu_0, \upsilon_0) = (\{a_I\}, \emptyset)$. The extended run segment ϱ becomes an *extended run* of $\mathcal{D}$ if $(\mu_n, \upsilon_n) = (\{a_F\}, \emptyset)$. For a run ρ of $\mathcal{D}$ of the form

$$\rho = \mu_0' \xrightarrow{t_0'} \mu_1 \xrightarrow{t_1'} \cdots \xrightarrow{t_{m-1}'} \mu_m',$$

where ϱ is a *linear execution* of ρ if the transition sequence t_0', t_1', …, t_{m-1}' can be constructed from the sequence t_0, t_1, …, t_{n-1} by removing any t_i $(t_i = \lambda, 0 \le i < n)$. Notice that ρ could have many extended runs as its linear executions.

For a Java program, let ω be its execution trace which is a sequence $m_0 \,\hat{}\, m_1 \,\hat{}\, \ldots \hat{}\, m_n$ of method completions where m_i $(0 \le i \le n)$ represents a method. Let $\mathcal{D} = (A, T, F, a_I, a_F)$ be an activity diagram, and

$$\varrho = (\mu_0, \upsilon_0) \xrightarrow{t_0,a_0} (\mu_1, \upsilon_1) \xrightarrow{t_1,a_1} \cdots \xrightarrow{t_n,a_n} (\mu_{n+1}, \upsilon_{n+1})$$

```
current_runsegment := ⟨({a_I}, ∅)⟩;
repeat
   node := (μ, υ) which is the last node of
           current_runsegment;
   if node has no new successive node (μ', υ')
      such that (μ, υ) --t,a--> (μ', υ')
   then delete the last node of current_runsegment
   else begin
          node := a new successive node of node;
          if node is ({a_F}, ∅) and such that
             current_runsegment is consistent with ω
          then
             begin
               append node to current_runsegment;
               return current_runsegment
             end;
          if node is such that current_runsegment
             is a prefix for ω
          then append node to current_runsegment;
        end
until current_runsegment = ⟨⟩;
return "no run consistent with ω".
```

FIGURE 4. Algorithm for matching program execution traces with activity diagram behavior.

be an extended run segment of $\mathcal{D}$. We say that ϱ is *consistent with* ω if $\xi(a_i) = m_i$ for any i $(0 \le i \le n)$. For a run ρ of $\mathcal{D}$, we say that ρ is *consistent with* ω if there is an extended run which is a linear execution of ρ and consistent with ω.

Let $\mathcal{D} = (A, T, F, a_I, a_F)$ be an activity diagram, and $\omega = m_0 \,\hat{}\, m_1 \,\hat{}\, \ldots \,\hat{}\, m_n$ be a program execution trace. For developing an algorithm to find a run of $\mathcal{D}$ which is consistent with ω, we need to introduce *prefixes* for ω. An extended run segment ϱ of $\mathcal{D}$ is a *prefix* for ω if there is i $(0 \le i < n)$ such that ϱ is consistent with $m_0^{\wedge} m_1^{\wedge} \ldots \,\hat{}\, m_i$.

Given an activity diagram $\mathcal{D} = (A, T, F, a_I, a_F)$ and a program execution trace ω, we developed an algorithm to match ω with the behavior of $\mathcal{D}$, i.e. to find a run of $\mathcal{D}$ which is consistent with ω (cf. Fig. 4). The algorithm traverses the state space of $\mathcal{D}$ in a depth first manner starting from the initial extended state ($\{a_I\}$, $\emptyset$). The extended run segment in the state space that we have so far traversed is stored in the list variable *current_runsegment*. For each new extended state that we discover, we first check whether it is ($\{a_F\}$, $\emptyset$) and such that *current_runsegment* is consistent with ω. If yes, then a run of $\mathcal{D}$ consistent with ω is found out, and we are done. Otherwise we check if the new extended state that we discover is such that *current_runsegment* is a prefix for ω. If yes, then we add the new extended state to *current_runsegment* and start the search from it, otherwise we backtrack. The complexity of the algorithm is proportional to the number of the prefixes for ω and to the size of the longest prefix for ω.

3.4. Selecting test cases according to test adequacy criteria

After matching the program execution traces with the runs of a given activity diagram, we need to compute the contribution of the corresponding runs w.r.t. a given test adequacy criterion, and decide whether the corresponding test case should be selected.

For the activity/transition coverage, the test case selection is simple. For each run which is consistent with a program execution trace, if it contains some activity states/transitions which are not covered previously, then the corresponding test case is picked out. The selection process terminates when the test adequacy criterion is satisfied, i.e. all activity states/transitions in the activity diagram are covered, or no more test cases can be picked out. In the later case we need to compute the coverage value, which is the ratio of the covered activity states/transitions to all activity states/transitions in the activity diagram.

The simple path coverage requires that all the simple paths in a given activity diagram be covered. It follows that we first need to find out all the simple paths in a given activity diagram. For an activity diagram $\mathcal{D} = (A, T, F, a_I, a_F)$, we give an algorithm to generate all of its simple paths (cf. Fig. 5). The algorithm traverses the state space of $\mathcal{D}$ along with the concurrent transitions in a depth first manner starting from the initial state $\{a_I\}$. The path segment in the state space that we have so far traversed is stored in the list variable *current_pathsegment*, and the simple paths which are founded

```
current_pathsegment := ⟨{a_I}⟩;
simplepath_set := ∅;
repeat
   node := the last node of current_pathsegment;
   if node has no new successive node
      by firing a concurrent transition
   then delete the last node of current_pathsegment
   else begin
          node := a new successive node of node
                  by firing a concurrent transition;
          if node is {a_F} then
            begin
              append node to current_pathsegment;
              simplepath_set :=
                simplepath_set ∪ {current_pathsegment}
            end;
          if node is such that current_pathsegment
             can be extended into a simple path
          then append node to current_pathsegment;
        end
until current_pathsegment = ⟨⟩;
return simplepath_set.
```

FIGURE 5. Algorithm for generating simple paths.

out are stored in the set variable *simplepath_set*. For each new state that we discover by firing a concurrent transition, we first check whether it is $\{a_F\}$. If yes, then we find out a simple path and put it into *simplepath_set*. Otherwise we check if the new state that we discover is such that *current_runsegment* can be extended further into a simple path. If yes, then we add the new state to *current_pathsegment* and start the search from it, otherwise we backtrack. Notice the above algorithm traverses the state space of an activity diagram along with the concurrent transitions so that it avoids the complexity caused by concurrency.

In order to select test cases according to the simple path coverage, we need to check if a program execution trace covers a simple path. Let $\mathcal{D}$ be an activity diagram, and

$$\sigma = \mu_0 \xrightarrow{\tau_0} \mu_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{m-1}} \mu_m$$

be a simple path in $\mathcal{D}$. The simple path σ is covered by a program execution trace ω if there is a run ρ of $\mathcal{D}$ of the form

$$\rho = \mu_0' \xrightarrow{t_0} \mu_1' \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} \mu_n'$$

consistent with ω which is just a linear execution of σ, i.e.

$$\{t_0, t_1, \ldots, t_n\} = \tau_0 \cup \tau_1 \cup \cdots \cup \tau_{m-1}$$

and $t_i \neq t_j$ for any i, j $(0 \leq i < j \leq n)$. For each program execution trace which is generated by running the program with the random test cases, we can decide if its corresponding test case should be picked out for the simple path coverage by checking if it covers a simple path which is not previously covered. The selection process terminates when all simple paths in the activity diagram are covered or no more test case can be picked out. In the later case, we need to compute the coverage value, which is the ratio of the covered simple paths to all simple paths in the activity diagram.

The trail coverage requires that all the given trails in an activity diagram be covered. As the trails are introduced for specifying some special executions in a given activity diagram which should be covered in testing, they are usually singled out manually. In order to select test cases according to the trail coverage, we need to check if a program execution trace covers a trail. Let $\mathcal{D}$ be an activity diagram, and

$$\gamma = t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_m$$

be a trail in $\mathcal{D}$. The trail γ is covered by a program execution trace ω if there is a run ρ of $\mathcal{D}$ of the form

$$\rho = \mu_0' \xrightarrow{t_0'} \mu_1' \xrightarrow{t_1'} \cdots \xrightarrow{t_{n-1}'} \mu_n'$$

consistent with ω in which $t_0, t_1, \ldots, t_m$ fires in the same order in γ, i.e. the following condition is satisfied.

(i) for any t_i $(0 \leq i \leq m)$, there is t_j' $(0 \leq j < n)$ such that $t_i = t_j'$;
(ii) for any i, j $(0 \leq i < j \leq m)$, if $t_i = t_k'$ and $t_j = t_l'$ then $0 \leq k < l < n$; and
(iii) for any i $(0 \leq i < m)$, if $t_i = t_k'$ and $t_{i+1} = t_l'$ then $t_p' \neq t_q$ for any p $(k < p < l)$ and any q $(0 \leq q \leq m)$.

For example, for the trail $t_1 \rightarrow t_4 \rightarrow t_7$ in the activity diagram depicted in Fig. 1, it is covered by a program execution trace which is consistent with the run $\{a_I\} \xrightarrow{t_1} \{a_1, a_2, a_3\} \xrightarrow{t_2} \{a_2, a_3, a_4\} \xrightarrow{t_4} \{a_3, a_4, a_6\} \xrightarrow{t_6} \{a_4, a_6, a_8, a_9\} \xrightarrow{t_7} \{a_6, a_8, a_9, a_{10}\} \xrightarrow{t_9} \{a_8, a_9, a_{10}, a_{11}\} \xrightarrow{t_{11}} \{a_{10}, a_{11}, a_{12}\} \xrightarrow{t_{13}} \{a_F\}$. Given the trails in an activity diagram as a trail coverage criterion, for each program execution trace which is generated by running the program with the random test cases, we can decide if its corresponding test case should be picked out for the trail coverage by checking if it covers a trail which is not previously covered. The selection process terminates when all the given trails are covered or no more test case can be picked out. In the later case, we need to compute the coverage value, which is the ratio of the covered trails to all the given trails.

3.5. Consistency checking

The algorithm described in Fig. 4 can also be used to check the consistency between the program execution traces and the behavior of activity diagrams. For a program under testing with an activity diagram as its design model, for a program execution trace ω we gather, if there is no run in the activity diagram consistent with ω, then an inconsistent case occurs. There are two causes for this inconsistent case: one is the program bugs resulting from the wrong temporal orders of method calls, the other is that the activity diagram is imperfect itself.

Therefore, the approach presented above can also be used to detect not only the program bugs resulting from the wrong temporal orders of method calls, but also the imperfect activity diagram models constructed in reverse engineering for the legacy systems. Therefore, this approach also leads to a testing tool, which may proceed in a fully automatic fashion.

3.6. Tool prototype and case study

We have implemented a tool prototype to support the approach presented in this paper. The tool has a graphical interface to allow users to construct, edit and analyze activity diagrams interactively. Its snapshot is shown in Fig. 6. The tool can instrument a Java program according to a given activity diagram, use the randomly generated test cases to run the instrumented program and gather the corresponding program execution traces. By comparing these traces with the behavior of the activity diagram, the tool can pick out

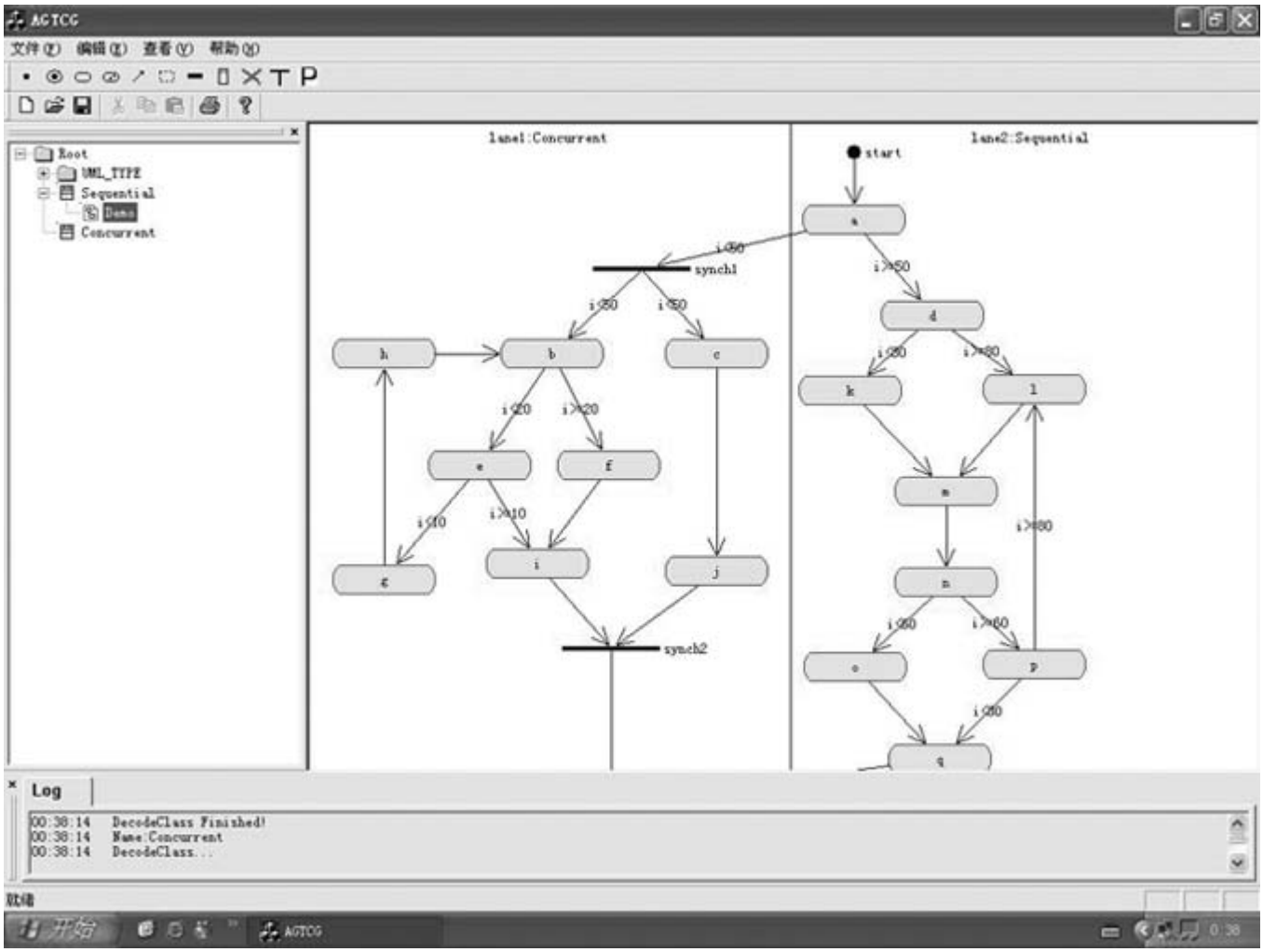

FIGURE 6. Interface of the tool prototype.

test cases into a test suite according to a given test adequacy criterion, as well as evaluate to which extent the test suite satisfies the test adequacy criterion. The tool can also be used to check the consistency between the program execution traces and the behavior of activity diagrams.

By the tool, we have conducted several case studies for showing the potential and usability of the approach presented in this paper. One case study is an on-line stock exchange system (OSES), which is reconstructed from an example in [13]. It is implemented in Java and contains 40 classes, 305 methods. The main purpose of OSES, which is modeled by an activity diagram depicted in Fig. 7, is to accept, check and execute the customer's orders. These features are implemented by Stock Broker and `Securities Exchange`. First, `Stock Broker` accepts a customer's order, and checks it. If the customer's account or the ordered stock does not exist, this order will be stopped. Otherwise, it will be submitted to `Securities Exchange` for further processing. Then, `Securities Exchange` executes the order in different ways according to its type and operation. Executing an order is a matching process, i.e. `Securities Exchange` searches appropriate orders in the database to make trade. For a market-order, `Securities Exchange` just finds the buy-order or sell-order and makes trade by the current stock price. For a limit-order, as it must be traded by the restricted or better price, `Securities Exchange` first checks the price set given by the customer. If the price is valid, Securities Exchange will find the buy-order or sell-order to make trade by the limited or better price. Otherwise, the order will not be traded and its result will be set 'NO MATCH'. For any order, its executing result falls into four classes: 'FAILURE', 'SUCCESS', 'PARTLY EXECUTION' and 'NO MATCH'. If the order is invalid, its result will be 'FAILURE'. If the ordered stock is traded

completely, partly or none, the result will be 'SUCCESS', 'PARTLY EXECUTION' or 'NO MATCH', respectively. After executing the order, `Securities Exchange` starts multiple threads to process the result concurrently, and exits at last.

In the activity diagram depicted in Fig. 7, there are totally 25 activity states, 30 transitions and 18 simple paths. Each activity state in this diagram is labeled with a method in the program. The input of OSES is an order object, which consists of several member variables, such as the number of ordered stock, the ordered amount, the type of order (market-order or limit-order) and the operation of the order (buy-order or sell-order). Ordered price is required for limit-order. We can generate an order by randomly setting these variables. For

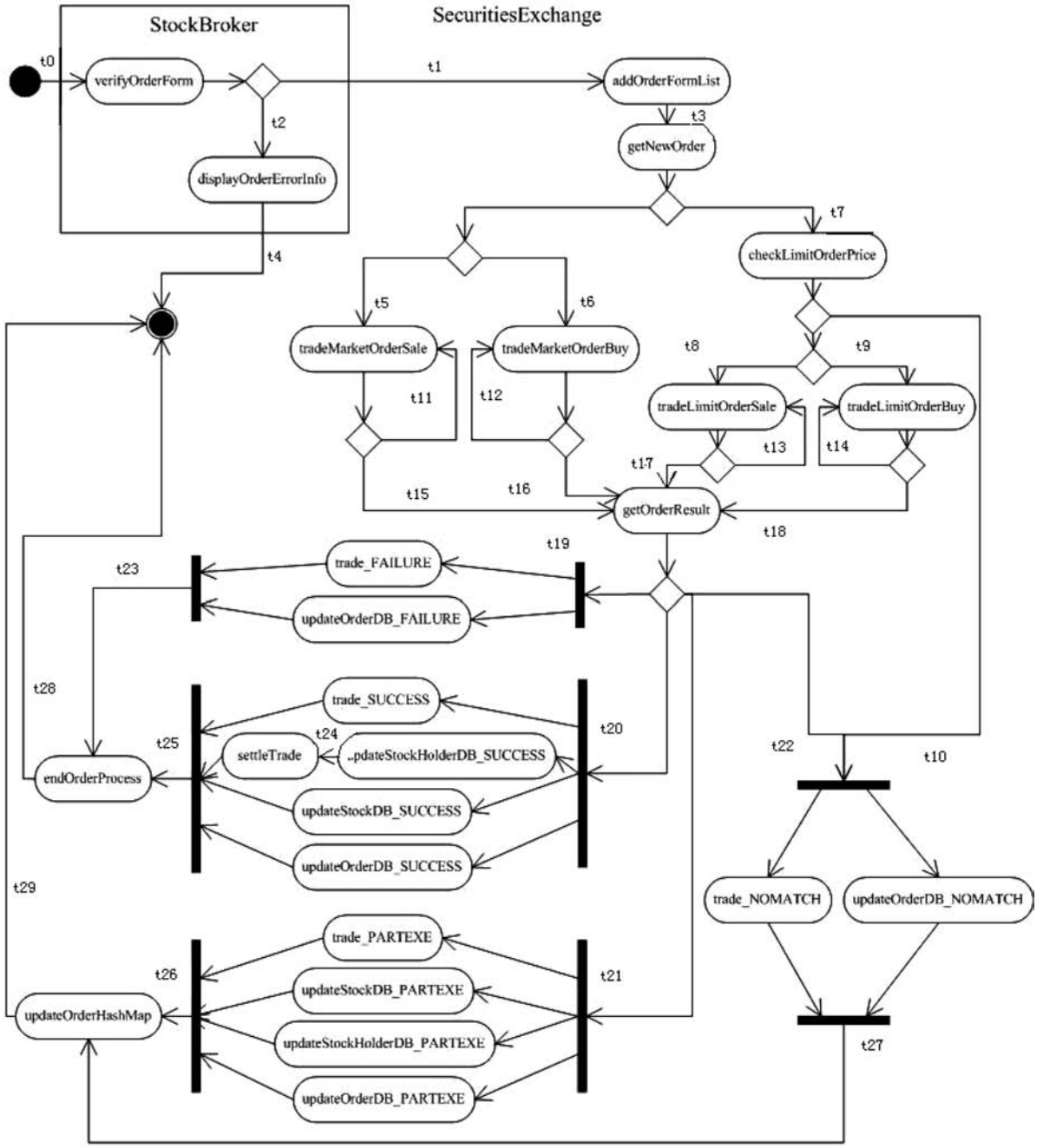

FIGURE 7. Activity diagram for the OSES

the trail coverage criterion, we choose the following four trails as the special executions in the activity diagram which should be covered in testing.

$$\gamma_1 = t_6 \to t_{12} \to t_{12} \to t_{12} \to t_{16} \to t_{19},$$
$$\gamma_2 = t_5 \to t_{11} \to t_{11} \to t_{11} \to t_{15} \to t_{19},$$
$$\gamma_3 = t_7 \to t_8 \to t_{13} \to t_{13} \to t_{13} \to t_{17} \to t_{19},$$
$$\gamma_4 = t_7 \to t_9 \to t_{14} \to t_{14} \to t_{14} \to t_{18} \to t_{19}.$$

With the tool, we instrument the program, run it by generating 20, 50, 100, 300, 500 and 800 random orders, respectively, for OSES as inputs, gather the corresponding execution traces, and select the test cases into the test suite according to the four test adequacy criteria introduced in Section 3.1. The experimental results are shown in Table 1.

With the tool, we also conduct the experiments on checking the consistency between the program execution traces and the behavior of the activity diagram, and the tool comfortably finds out all the inconsistent cases which results from several related bugs embedded manually in the program or activity diagram.

The other case studies we have conducted have approximately the same size as the OSES. Although we do not conduct any more case studies with larger size, we think there is no particular obstacle to implement the approach presented in this paper in a fully automatic fashion since the algorithms in the approach are simple and efficient.

As in the approach we select the test cases from a set of randomly generated test cases, there is an important question in random testing, which is how many random test cases are sufficient? However this problem is not so concerned with us. That is because the goal of our approach is to automate the test case generation process so as to reduce the testing cost. Owing to the inexpensive charge, the tool can run as long as possible. We think the random test cases sufficient when the tool has been running long enough in our tolerable duration, or when an apparent and believable result can be concluded, i.e. the given test adequacy criteria are satisfied, or an inconsistent case is detected.

4. RELATED WORK

UML model-based testing has being attracted more and more research attention [3–10, 14–17]. A large part of them are focused on UML interaction models and state machine diagrams-based testing techniques [3–10, 14]. Only a few works relate to making the use of UML activity diagrams in software testing [15–17]. Most of those approaches generate abstract test cases directly from the UML models only, and none of them makes the use of the programs during the test

TABLE 1. Experimental results on the OSES.

	Activity coverage criteria			Transition coverage criteria		
r	k	Number of covered activities	Coverage (%)	k	Number of covered transitions	Coverage (%)
20	5	20	80	5	28	93.3
50	7	20	80	7	28	93.3
100	7	20	80	7	28	93.3
200	7	20	80	7	28	93.3
300	7	25	100	7	30	100
500	8	25	100	8	30	100
800	7	25	100	7	30	100
	Simple path coverage criteria			Trail coverage criteria		
	k	Number of covered simple paths	Coverage (%)	k	Covered trails	Coverage (%)
20	9	9	50	1	γ_2	25
50	10	10	55.6	1	γ_2	25
100	11	11	61.1	2	γ_1, γ_2	50
200	14	14	78.8	4	γ_1, γ_2, γ_3, γ_4	100
300	15	15	83.3	4	γ_1, γ_2, γ_3, γ_4	100
500	16	16	88.9	4	γ_1, γ_2, γ_3, γ_4	100
800	18	18	100	4	γ_1, γ_2, γ_3, γ_4	100

r is the number of the random test cases.
k is the number of the selected test cases.

generation. Our approach employs both models and programs for test case generation. The randomly generated test cases are first executed with the program, and then they are selected according to the test adequacy criteria with respect to models. The selected test cases are concrete and executable since they have already been executed with the program before the selection.

Chen *et al.* [15] propose an approach to apply category-partition method to UML activity diagrams to identify categories and choices, which is the first step of test data generation. They aim to develop an identification methodology for informal specifications so that their approach is totally manual, and can hardly applied to large industrial projects. Vieira *et al.* [16] redefine the role of activity diagrams in software development process, in which the activity diagrams are used to describe how the functionality depicted in the use case diagrams can be exercised in terms of workflow, and are used to create test drivers to verify the modeled functionality. They derive the test case flow from the workflow described in an activity diagram, and generate the test scripts with more precise TSL language, but their activity diagrams need to be annotated for test purpose, which requires users' strong background of testing, modeling and domain knowledge. We have given an approach to generate test case from UML activity diagrams based on the Gray-Box method [17]. It demonstrates a systematic method to generate test cases directly from UML activity diagrams, and many parts of this method could be automated, but our coverage criteria are limited and the concrete test data generation is still hardly automated.

Agitator [18], a commercial tool, adopts a similar approach to the work in this paper. It (almost randomly) creates very simple test cases, and then refines them to satisfy the test adequacy criteria. But a key difference between Agitator and our approach is that Agitator tests are designed for Java methods (unit testing) and the criteria are based on an implementation rather than a model. Godefroid [19] presented a dynamic test case generation method SMART, which adopts dynamic program analysis, symbolic execution and constraint solver techniques to solve the test case generation problem. With the same idea as our approach, SMART employs the random method to generate concrete test inputs, and selects test cases based on program executions. But currently, SMART can only process C programs, and cannot support object-oriented programs yet.

The approach presented in this paper can also be used to check the consistency between the program execution traces and the behavior of UML activity diagrams. Therefore, our approach is also a kind of runtime verification techniques. The runtime verification techniques have been used to detect the concurrency errors such as deadlocks and data races for Java programs and the other programs [20–25]. In those literatures, most of the specification languages are based on temporal logic. Compared to those temporal logic-based specification languages, UML activity diagrams are more acceptable in industry, and may come directly from the artifacts generated in software development processes.

In traditional regression testing, the test cases, which are generated during previous testing, are selected and reduced based on the efficiency of the test suite. In other words, the reduced test suite has the same fault detecting power with the original test suite [26, 27]. In our approach, the test cases are first generated randomly, and then selected according to the test adequacy criteria with respect to activity diagram models without considering fault detection. The objective of our approach is to automate the test case generation process for reducing the testing cost other than the efficiency of fault detection.

5. CONCLUSION

This paper propose an approach to automatic test case generation for Java programs with UML activity diagrams as design models. Guided by a given activity diagram, the program under testing is first instrumented so as to collect the related program execution traces. Then abundant test cases are randomly generated for driving the program. By running the instrumented program with these randomly generated test cases, we obtain the corresponding program execution traces. By matching those program execution traces with the behavior of the activity diagram, we select a reduced test suite according to the test adequacy criterions concerning the activity diagram. The approach can also be used to check the consistency between the program execution traces and the behavior of activity diagrams.

The approach presented in this paper focuses on the test case generation for Java programs, but its underlying idea is more general and may also be applied to the test case generation for other object-oriented programs. The next work is to extend the approach to support the compositions of UML dynamic models as design specifications for test case generation.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable comments and suggestions. This work is supported by the National Natural Science Foundation of China (No. 60425204, No. 60233020 and No. 60673125), the National Grand Fundamental Research 973 Program of China (No. 2002CB312001) and by Jiangsu Province Research Foundation (No. BK2007714). A preliminary version [28] of this paper appears in the proceedings of International Workshop on Automated Software Testing (AST06).

REFERENCES

[1] Rumbaugh, J., Jacobson, I. and Booch, G. (2001) *The Unified Modeling Language User Guide*. Addison-Wesley, Boston.

[2] UML 2.0 (2005) *UML 2.0 Superstructure Specification*. OMG, available at http://www.omg.org/uml.

[3] Gnesi, S., Latella, D. and Massink, M. (2004) Formal test case generation for UML statecharts. *Proc. Ninth IEEE Int. Conf. Engineering Complex Computer Systems* (*ICECCS'04*), Florence, Italy, April 14–16, pp. 75–84. IEEE Computer Society Press, NJ.

[4] Latella, D. and Massink, M. (2002) On testing and conformance relations for UML statechart diagram behaviors. *Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA2002)*, Roma, Italy, July 22–24. *ACM Softw. Eng. Notes*, **27**, 144–153.

[5] Latella, D. and Massink, M. (2001) A formal testing framework for UML statechart diagram behaviors: from theory to automatic verification. *Proc. Sixth IEEE Int. High-Assurance Systems Engineering Symp.* Florida, October 22–24, pp. 11–22. IEEE Computer Society Press, NJ.

[6] Seifert, D., Helke, S. and Santen, T. (2003) Test case generation for UML statecharts. *Proc. Perspectives of System Informatics*, Akademgorodok, Novosibirsk, Russia, July 9–12. Lecture Notes in Computer Science, Vol. 2890, pp. 462–468, Springer, Berlin Heidelberg.

[7] Offutt, J. and Abdurazik, A. (1999) Generating tests from UML specifications. *Proc. Second Int. Conf. Unified Modeling Language* (*UML1999*), Fort Collins, CO, October 24–30. Lecture Notes in Computer Science, Vol. 1723, pp. 416–429, Springer, Berlin Heidelberg.

[8] Chevalley, P. and Thevenod-Fosse, P. (2001) Automated generation of statistical test cases from UML state diagrams. *Proc. Int. Computer Software and Applications Conf.*, Chicago, IL, USA, October 8–12, pp. 205–214. IEEE Computer Society Press, NJ.

[9] Kim, Y., Hong, H., Cho, S., Bae, D. and Cha, S. (1999) Test case generation from UML state diagrams. *IEEE Proc. Software* **146**, 187–192.

[10] Offutt, J. and Abdurazik, A. (2000) Using UML collaboration diagrams for static checking and test generation. *Proc. Third Int. Conf. Unified Modeling Language* (*UML2000*), York, UK, October 2–6. Lecture Notes in Computer Science, Vol. 1939 pp. 383–395, Springer, Berlin Heidelberg.

[11] Peterson, J. (1981) *Petri Nets Theory and the Modeling of Systems*. Prentice-Hall, NJ.

[12] Zhu, H., Hall, P. and May, J. (1997) Software unit test coverageacy. *ACM Comput. Surv.*, **29**, 366–427.

[13] Blaha, M. and Rumbaugh, J. (2005) *Object-Oriented Modeling and Design with UML* (2nd edn). Pearson Education, Inc.

[14] Andrews, A., France, R., Ghosh, S. and Craig, G. (2003) Test adequacy criteria for UML design models. *Softw. Test. Verif. Reliab.*, **13**, 95–127.

[15] Chen, T., Poon, P., Tang, S. and Tse, T. (2005) Identification of categories and choices in activity diagrams. *Proc. Fifth Int. Conf. Quality Software*, Melbourne, Australia, September 19–21, pp. 55–63. IEEE Computer Society Press, NJ.

[16] Vieira, M., Leduc, J., Hasling, B., Subramanyan, R. and Kazmeie, J. (2006) Automation of GUI testing using a model-driven approach. *Proc. of Int. Workshop on Automation of Software Test (AST06)*, Shanghai, China, May 20–26, pp. 9–14. IEEE Computer Society Press, NJ.

[17] Wang, L., Yuan, J., Yu, X., Hu, J., Li, X. and Zheng, G. (2004) Generating test cases from UML activity diagram-based on Gray-Box method. *Proc. 11th Asia-Pacific Software Engineering Conf.* (*APSEC2004*), Busan, Korea, 30 November– 4 December, pp. 284–291. IEEE Computer Society, NJ.

[18] Agitar. (2007) *Agitator and Agitar Management Dashboard 3.0. Available at* http://www.agitar.com.

[19] Godefroid, P. (2007) Compositional dynamic test generation. *Proc. 34th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, Nice, France, January 17–19. *ACM SIGPLAN Not.*, **42**, 47–54.

[20] Bartetzko, D., Fischer, C., Moller, M. and Wehrheim, H. (2001) Jass—Java with Assertions. *Electron. Notes Theor. Comput. Sci.*, **55**, 103–117.

[21] Havelund, K. and Rou, G. (2001) Monitoring Java programs with Java PathExplorer. *Electron. Notes Theor. Comput. Sci.*, **55**, 200–217.

[22] Kim, M., Kannan, S., Lee, I., Sokolsky, O. and Viswanathan, M. (2001) Java-MaC: a run-time assurance tool for Java programs. *Electron. Notes Theor. Comput. Sci.*, **55**, 218–235.

[23] Brorkens, M. and Moller, M. (2002) Dynamic event generation for runtime checking using the JDI. *Electron. Notes Theor. Comput. Sci.*, **70**, 1–15.

[24] d'Amorim, M. and Havelund, K. (2005) Event-based runtime verification of Java programs. *Proc. Int. Workshop on Dynamic Analysis (WODA2005)*, St. Louis, MI, USA, 17 May, pp. 1–7. ACM Press, NY.

[25] Artho, C., Drusinsky, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Rosu, G. and Visser, W. (2003) Experiments with test case generation and runtime analysis. Lecture Notes in Computer Science, Vol. 2598, pp. 87–107, Springer, Berlin, Heidelberg.

[26] Jones, J. and Harrold, M. (2003) Test-suite reduction and prioitization for modified condition/decision coverage. *IEEE Trans. Softw. Eng.*, **29**, 195–209.

[27] Offutt, J., Pan, J. and Voas, J. (1995), Procedures for reducing the size of coverage-based test sets. *Proc. 12th Int. Conf. Testing Computer Software*, Washington, DC , June, pp. 111–123

[28] Chen, M., Qiu, X. and Li, X. (2006) Automatic test case generation for UML activity diagrams. *Proc. Int. Workshop on Automated Software Testing (AST06)*, Shanghai, China, 23 May, pp. 2–8. ACM Press, NJ.