Technical Report No. NJU-SEG-2012-IC-006

Regression Test Cases Generation Based on Automatic Model Revision

Nan Ye, Xin Chen, Wenxu Ding,
Peng Jiang, Lei Bu and Xuandong Li

Postprint Version. Originally Published in:
International Symposium on
Theoretical Aspects of Software Engineering 2012
Page(s): 127 - 134

2012 IEEE Sixth International Symposium on Theoretical Aspects of Software Engineering

Regression Test Cases Generation Based on Automatic Model Revision

Nan Ye, Xin Chen*† Wenxu Ding, Peng Jiang, Lei Bu and Xuandong Li

State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing, Jiangsu 210093, P.R. China
yenan@seg.nju.edu.cn, chenxin@nju.edu.cn, {dingwx1987, jiangpeng}@seg.nju.edu.cn, {bulei, lxd}@nju.edu.cn

***Abstract*—Regression testing is a widely used way to assure the quality of modified software. It requires executing a suite of test cases to ensure that modifications do not introduce any negative impact to software behavior. To collect test cases in the suite that can reveal modifications, different versions of software must be compared carefully. Existing approaches, relying on manual examination on programs or models to identify differences, are expensive. In the paper, we present a fully automatic approach to generating regression test cases based on activity diagram revision. By collecting execution traces and revising old activity diagrams, the approach firstly constructs new activity diagrams that can reveal software behavior changes. Then, both affected paths and new paths in activity diagrams are identified. Finally, an execution-based approach is applied to generate regression test cases whose execution can cover these paths. Experiments show the effectiveness of our approach.**

***Keywords*-regression testing, generation, automatic, model-based testing**

I. Introduction

In the life cycle of software, regression testing is an important assurance for the quality of modified software. After modifications are made to software, a suite of regression test cases should be executed to ensure that these modifications did not introduce any negative impact to software behavior. As the software systems evolve with their scale and complexity, the amount of test cases becomes larger and larger. It is very costly and time-consuming to execute all the cases during the regression testing process. To improve the efficiency in regression testing, test cases selection techniques are used to select a subset of test cases that reveal modifications in the software [6].

To select modification-revealing test cases, one question should be answered first: how to identify the modifications in software? As model-centric approaches are widely used in software development, UML models, such as activity diagrams, can be used to describe the dynamic behavior of software. Therefore, identifying the modifications in software can be taken by comparing two different versions of UML models. Our previous work[5] presented an approach to identifying modifications in software by comparing different versions of activity diagrams and generating regression test cases to cover these modifications. To adopt this approach, activity diagrams for both versions of software need to be provided in advance. Besides the laborious work, as the scale and complexity of software grow up quickly, manual model maintenance becomes harder and harder.

As the design model, an activity diagram can represent the dynamic behavior of its related software. For any execution in software, its matching path can be found in the activity diagram. Modifying software may bring inconsistency between its current behavior and its out-of-date behavior models. If the activity diagram is out-of-date, execution traces may mismatch activity paths in the diagram. As modifications made to software usually affect limited aspects in software, even though the matching relationship does not exist between a complete execution trace and a complete activity path, it still exists between the subsequences of the trace and sub-paths in the activity diagram. We can revise the diagram on the base of partly-matching relationships to fit the trace. That means, by revising activity diagrams according to execution traces, new up-to-date diagrams can be constructed automatically without manual intervention. This is the approach deployed in the paper to construct activity diagrams according to execution traces automatically.

In this paper, we present an automatic approach for regression test cases generation based on activity diagrams revision. Here, activity diagrams play the role as the behavior models of the software under test. Path coverage in activity diagrams is chosen as the test adequacy criteria in regression testing. In this approach, firstly, a certain amount of test cases are executed on the evolved software. Our approach demands that the executions of these test cases cover all behaviors in current software. The selection strategy of test cases to be executed is discussed in Section 4. Then based on the execution traces and the old activity diagram, a

*Corresponding author.

†This work is supported by Research Fund for the Doctoral Program of Higher Education of China (No.20110091120058), the National Natural Science Foundation of China (No.61100036, No.61021062), National Grand Fundamental Research 973 Program of China (No.2009CB320702), and by the National 863 High-Tech Programme of China (No.2011AA010103).

new activity diagram representing the behavior of modified software is constructed automatically. Affected paths and new paths are identified as well. Finally, an execution-based approach is used to generate regression test cases for these paths. These test cases make up the regression testing suite.

The rest of the paper is organized as follows: Section 2 introduces the background of model-based regression testing techniques. Section 3 describes the underlying idea of this approach. Section 4 gives technical details on how to revise an activity diagram based on execution traces of modified software. Section 5 explains how to build the regression test suite on the base of revision. The case study is taken in section 6 to show the effectiveness of our approach. Related works are discussed in section 7. Section 8 gives the conclusion.

II. Background

In regression testing, regression test cases are usually selected from a prepared pool which satisfies some test adequacy criteria. It's assumed that the original software has been tested adequately for the criteria. Generally, the scale of pool is very large, and executing all the test cases will costly and time-consuming. In practice, modification-revealing techniques are widely used, which suggest only those test cases that can reveal modifications made to software are selected to execute in regression testing.

The adoption of model-based development provides opportunities for model-based testing. As the abstraction of software, models can be used to represent the modifications in software. By comparing different versions of models, those test cases related to changed parts in models are selected out as regression test cases, since they can reveal the effects of software modifications.

Activity diagrams are widely used as design models, since they are able to describe the internal behavior of software, i.e. a workflow of actions. Each activity in workflows is interpreted as the execution of a specific method call in software. As execution traces are composed of method calls, their corresponding covered paths can be found in activity diagrams by analyzing the execution traces[2]. Revealed from different versions of activity diagrams, their differences represent the behavior variation deriving from modifications. Our work [5] presents such an approach that identifies the behavior variation in software by comparing different versions of activity diagrams.

To adopt these model-comparison techniques, we need to construct models for each version of software under test. Unfortunately, in practical software development processes, model updating usually lags behind software changes since it relies on manual manipulations. Manual revision for models is laborious and lowers the availability of these techniques.

III. Essentials of our approach

Modifications made to software result in software behavior variation. As a result, the behavior variation may not

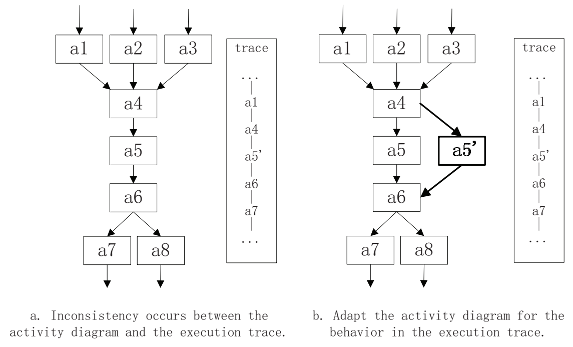
a1
a2
a3
trace
...
a1
a4
a5'
a6
a7
...
a4
a5
a6
a7
a8
a1
a2
a3
a4
a5
a5'
a6
a7
a8
trace
...
a1
a4
a5'
a6
a7
...
a. Inconsistency occurs between the activity diagram and the execution trace.
b. Adapt the activity diagram for the behavior in the execution trace.

Figure 1. The revision of activity diagram.

conform to the original activity diagram any more. The out-of-date diagram cannot be used as the correct abstraction for software any more. If the execution of a test case involves behavior variation in software, inconsistency may occur. That is, its execution trace cannot be accepted by the original activity diagram or cover its former related path in the diagram. As shown in Figure 1a, an unexpected method call *a5'* occurs in the execution trace, which cannot be accepted by current activity diagram.

Each execution trace reflects the associated dynamic behavior of a test case executed on modified software. So the activity diagram can be revised according to the information extracted from execution traces. As shown in Figure 1b, if the sequence $\langle ...a_1, a_4, a_5', a_6...\rangle$ occurs in the trace, it can be inferred that a new activity a_5' should be added between the activity a_4 and a_6. According to the inference, the original activity diagram can be revised to fit behavior in execution traces.

The small example in Figure 1b gives some clues on how to automatically revise activity diagrams according to execution traces. Given an activity diagram, a trace and a mapping between activities in the diagram and method calls in the sequence, we can mark both matched parts and unmatched parts between traces and the diagram. For each unmatched part, the revision operation is taken, including the adding and removing of activities and transitions. After the revision is completed, the trace can be accepted by the diagram.

Note that the revision may lead to the emergence of new paths in the activity diagram. Considering the example in Figure 1b, after the revision is taken, besides the path $\langle ...a_1, a_4, a_5', a_6, a_7...\rangle$, there are still other 5 additional paths required to be covered, e.g., $\langle ...a_2, a_4, a_5', a_6, a_7...\rangle$ and $\langle ...a_3, a_4, a_5', a_6, a_8...\rangle$. Those paths that contain new activities or new transitions are of high possibility not to be covered by the original test cases. To meet the path coverage criteria, new test cases need to be created to cover these paths.

The overview of our approach is shown in Figure 2. It con-

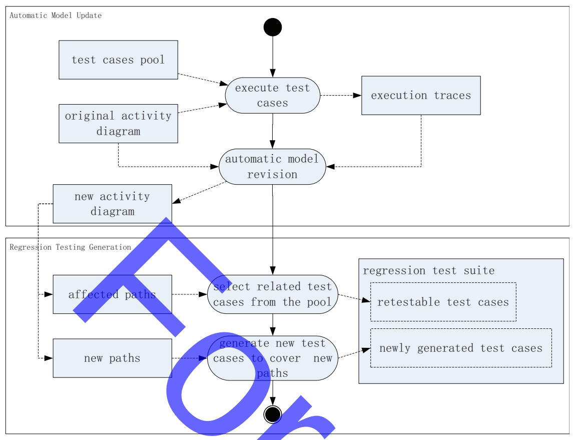

Figure 2. The workflow of regression test cases generation based on automatic model revision.

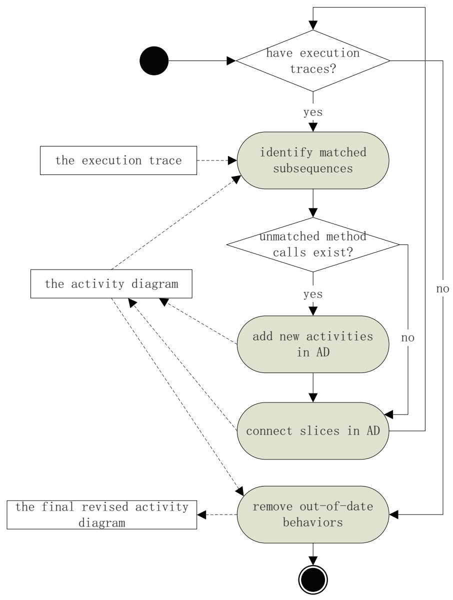
have execution traces?
yes
no
the execution trace
identify matched subsequences
unmatched method calls exist?
the activity diagram
yes
no
add new activities in AD
connect slices in AD
the final revised activity diagram
remove out-of-date behaviors

Figure 3. The process of automatic activity diagram revision.

tains 2 phases. First, original activity diagrams are revised in accordance with current software behavior automatically. A certain amount of test cases are selected from the prepared pool of test cases and executed on the current evolved software. Then the activity diagram is revised according to unmatched parts between each execution trace and the diagram. The revised diagram can accept the execution traces which it could not accept previously. Second, the regression test suite is built on the base of the revised diagram. The regression test suite is composed of two parts: test cases selected from the pool which cover those affected paths in the activity diagram, and newly generated test cases which cover new paths in the diagram.

IV. Automatic Activity Diagram Revision

In this section, we present the automatic approach to revise the original activity diagram in accordance with execution traces from the modified software. To illustrate the approach, we employ the formal definitions of activity diagrams in [2].

Definition 1. *activity diagram*

*An **activity diagram** is a tuple, (A, T, F, a_I, a_F), where*

1) *$A = \{a_1, a_2, ..., a_m\}$ is a finite set of activity states;*
2) *$T = \{t_1, t_2, ..., t_n\}$ is a finite set of completion transitions;*
3) *$F \subset (A \times T) \bigcup (T \times A)$ is the flow relation;*
4) *$a_I \in A$ 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 $\forall t_i \in T \bullet (t_i, a_I) \notin F \bigwedge (a_F, t_i) \notin F$.*

As Figure 3 shows, the automatic revision process consists of four steps: identifying matched subsequences, adding new activities for unmatched method calls, connecting the discrete slices in the diagram, and removing out-of-date behaviors.

As the revision of activity diagrams is based on execution traces, a suite of test cases need to be executed to explore current software behavior. Software behavior need to be covered as sufficiently as possible. These test cases are selected from a pool of test cases. As the traceability between test cases in the pool and paths in the original activity diagram has been built, for each path, a certain count (in our case, the count is 5.) of test cases are selected for execution on purpose. Although the traceability may have been out-of-date in evolved software, it still can provide more confidence for behavior coverage than random selection.

A. match subsequences with slices in AD

The partly matching between an execution trace and an activity diagram is carried out by detecting *matched subsequences*. Here, an execution trace is a sequence of method calls s, represented as $s = \langle c_1, c_2, ..., c_n \rangle$, where c_i represents the i-th method call in the sequence. Treating an activity diagram as an design model, the correlation between activities in the diagram and methods in the program can be built. The mapping ac from method calls in s to activities in its related activity sequence represents the correlation. Thus, $ac(c_i)$ represents the related activity of the method call c_i. The *matched subsequence* is defined as follows.

Definition 2. *matched subsequence*

*Let $AD = (A, T, F, a_I, a_F)$ be an activity diagram, $s = \langle c_1, c_2, ..., c_n \rangle$ be a method call sequence and ac maps each element in s to its related activity in AD to form the related activity sequence. A subsequence ms of s, of the form $ms = \langle c_j, c_{j+1}, ..., c_m \rangle (1 \leq j < m \leq n)$, is a **matched subsequence** with respect to AD, if the following condition*

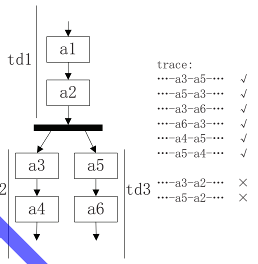

Figure 4. The thread parallel relation in activity diagrams.

holds:

$$\forall i \in j..m-1 \bullet \exists t \in T \bullet (ac(c_i), t) \in F \wedge (t, ac(c_{i+1})) \in F$$

By definition, a matched subsequence can find its matched slice in AD by applying the mapping ac to all its elements. For convenience, given a matched subsequence ms, we use $AS(ms)$ to denote its matched slice, that is $AS(ms) = \langle ac(c_j), ac(c_{j+1}), ..., ac(c_m)\rangle$. The *maximum matched subsequence* of s is its longest matched subsequence. It's easy to see that if an execution trace can be accepted by the activity diagram without any mismatches, its maximum matched subsequence will be itself.

The above definition only considers the sequential behaviors in the activity diagram. The activity diagrams may contain parallel threads to describe concurrent behavior in software. The thread parallel relation represents the relation between two parallel control flows in the activity diagram. As Figure 4 shows, the thread *td2* and *td3* are parallel threads, so, activities belonging to *td2* and *td3* interleaved with each other in execution traces. Because the parallel relation doesn't exist between *td1* and *td2* or *td1* and *td3*, such interleaved execution traces will not appear.

The following definition for *matched subsequence* handles the concurrent behaviors:

Definition 3. *matched subsequence*

Let $AD = (A, T, F, a_I, a_F)$ *be an activity diagram,* $s = \langle(c_1, p_1), (c_2, p_2)..., (c_n, p_n)\rangle$ *be a method call sequence, where* c_i *and* p_i *represents the i-th method call and its thread id respectively. The map acp maps each element* (c_i, p_i) *in* s *to its related activity* a_i *in the thread* $d_{r_i} \in D$ *in AD to form the related activity sequence.* $D = \{d_1, d_2...d_l\}$ *is the set of all threads in AD.* $PR \subset (D \times D)$ *is the parallel thread relation. A subsequence ms of s, of the form* $ms = \langle(c_j, p_j), (c_{j+1}, p_{j+1}), ..., (c_m, p_m)\rangle (1 \leq j < m \leq n)$*, is a **matched subsequence** with respect to AD, if the following two conditions hold:*

1) *for any successive elements* (c_{k_0}, p_r) *and* (c_{k_1}, p_r) *of the same thread* p_r *in s, their counterpart* (a_{k_0}, d) *and* (a_{k_1}, d) *in AD satisfy:*

$$\exists t \in T \bullet (a_{k_0}, t) \in F \wedge (t, a_{k_1})) \in F$$

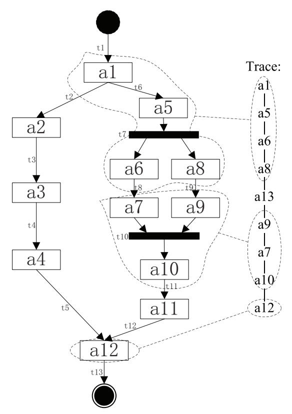

Figure 5. The local matches between the trace and the activity diagram.

2) *for any successive elements* (c_k, p_k) *and* (c_{k+1}, p_{k+1}) *in s, their counterpart* (a_k, d_{r_k}) *and* $(a_{k+1}, d_{r_{k+1}})$ *in AD satisfy:*

$$(\exists t \in T \bullet (a_k, t) \in F \wedge (t, a_{k+1}) \in F) \vee (d_{r_k}, d_{r_{k+1}}) \in PR$$

Algorithm 1 shows the process of identifying the maximum matched subsequence of a method call sequence on an activity diagram. In the algorithm, for a sequence seq, the function $seq[i]$ returns the i-th element in seq(the index starts from 0), $\#seq$ returns the length of seq. The operation $seq\hat{\ }a$ appends the element a to the sequence seq. For a set s, $MAX(s)$ and $MIN(s)$ return the maximum element and the minimum element of s respectively.

For each received execution trace s, its matched subsequences are searched recursively: identify its maximum matched subsequence ms as Algorithm 1 at first, and then repeat the process for the antecedent subsequence and subsequent subsequence of ms in s, until no more matched subsequences detected. We can identify all matched subsequences in execution traces, as shown in Figure 5.

B. add new activities in AD

Modifications made to software may introduce new method calls, which don't have related activities in original activity diagrams, like $a13$ in Figure 5. Obviously these method calls don't belong to any matched subsequences. We call them *unmatched method calls.*

To handle unmatched method calls in execution traces, a new corresponding activity in the diagram is created for each unmatched method call. As shown in Figure 6a, a new activity $a13$ is created to relate the method call $a13$. As the new created activity does not have its thread parallel relations like other existing activities, for each unmatched method call, we search its preceding or following method calls with the same thread id in the execution trace, that is its sibling method calls. We assume that the thread parallel relation of its corresponding activity is consistent with the corresponding activities of its sibling method calls.

Algorithm 1 identify the maximum matched subsequence of a given method call sequence

```
Input:
    An activity diagram $AD = (A, T, F, a_I, a_F)$
    A method call sequence $s = \langle c_0, c_1, ..., c_{n-1} \rangle$
    $TD$ is the set of threads in $AD$
    The thread parallel relation $TR \subset (TD \times TD)$
    $td(a) \in TD$ is the thread of activity $a \in A$
    $act$ is the set of activities correlating with the method of $c_i$ in $s$
Output:
    $ms_{max}$, the maximum matched subsequence in $s$
    $as_{max}$, the related activity sequence of $ms_{max}$
1:  $ms_{max} = \langle\rangle$; {initial empty sequence}
2:  $as_{max} = \langle\rangle$; {initial empty sequence}
3:  for i = 0 to n-1 do
4:    if $act(s[i]) \neq \emptyset$ then
5:      for all $a_{tmp} \in act(s[i])$ do
6:        let $ms_t = \langle s[i] \rangle$; {the temp matched subsequence}
7:        let $as_t = \langle a_{tmp} \rangle$; {its related activity sequence}
8:        let $pos = i + 1$;
9:        while $(pos < n)$ do
10:         if $act(s[pos]) \neq \emptyset$ then
11:           let $a_{t_0} = nil$;
12:           for all $a_t \in act(s[pos]$ do
13:             let $a_t = act(s[pos])$;
14:             if $\exists j \in \mathbb{N} \bullet td(a_t) = td(ac(ms_t[j]))$ then
                {find the immediate preceding activity of $a_t$ on the same thread}
15:               let $j_{max} = MAX(\{j \in \mathbb{N} | td(ac(ms_t[j])) = td(a_t)\})$;
16:               if $\exists t \in T \bullet (ac(ms_t[j_{max}]), t) \in F \wedge (t, a_t) \in F$ then
17:                 $a_{t_0} = a_t$;
18:               end if
19:             else
20:               let $a_f = act(ms_t[\#ms_t - 1])$;
21:               if $(\exists t \in T, j \in \mathbb{N} \bullet (ac(ms_t[j]), t) \in F \wedge (t, a_t) \in F)$ or $(td(a_f), td(a_t)) \in TR$ then
22:                 $a_{t_0} = a_t$;
23:               end if
24:             end if
25:           end for
26:           if $a_{t_0} \neq nil$ then
27:             $ms_t = ms_t \hat{\ } m[pos]$;
28:             $as_t = as_t \hat{\ } a_{t_0}$;
29:           else
30:             break;
31:           end if
32:         else
33:           break;
34:         end if
35:         $pos = pos + 1$;
36:       end while
37:       if $\#ms_t > \#ms_{max}$ then
38:         $ms_{max} = ms_t$;
39:         $as_{max} = as_{tmp}$;
40:       end if
```

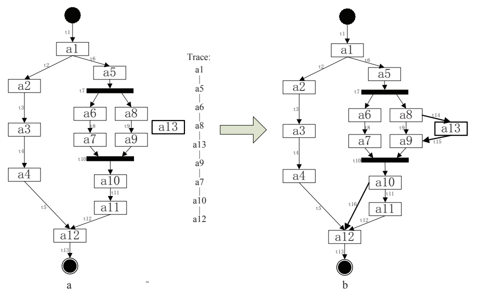

Figure 6. Revise the activity diagram to adapt the behavior in the execution trace.

So the newly created activity will inherit the thread parallel relations of the related activity of this sibling method call.

By this, unmatched method calls own their related activities in the diagram. They can be treated as matched subsequences with the length of one.

C. connect slices in AD

After completing the previous steps, all the matched subsequences in the execution trace are identified and new activities are created for all unmatched method calls. Then the execution traces can be considered as a sequence of matched subsequences.

To adapt the activity diagram for the execution trace, mismatches between the trace and the diagram should be amended. Mismatches derive from modifications made to software. Modifications in software usually lead to two kinds of software behavior variation, the disappearance of original behavior and the appearance of new behavior. In activity diagrams, these variations are reflected as the disappearance of original activities and transitions and the appearance of new ones. Including those unmatched method calls, all matched subsequences have been identified from the execution trace. The activity diagram can be revised to adapt software behavior in the execution trace by connecting these behavior slices in the activity diagram.

For two immediate matched subsequences ps_1 and ps_2 in an execution trace, neglecting concurrent behavior, it is clear that there are no transitions from the last activity a_{m1} in AS(ps_1) to the first activity a_{k2} in AS(ps_2) in the activity diagram, or else the two matched subsequences would join a larger one. Therefore, a new transition from a_{m1} to a_{k2} in the diagram can help connect the two matched subsequences together. Considering concurrent behavior, if an thread exists in both $AS(ps_1)$ and $AS(ps_2)$, there should be a transition to connect the last activity on the thread in $AS(ps_1)$ and the first activity on the thread in $AS(ps_2)$. If the transition doesn't exist, a new one will be created. The corresponding revision algorithm is proposed in Algorithm 2. For each received execution trace, after the revision is completed, the revised diagram can accept this complete execution trace, as Figure 6 shown.

Algorithm 2 revise the activity diagram to fit the behavior of a given execution trace

```
Input:
    An activity diagram $AD = (A, T, F, a_I, a_F)$
    An execution trace $s = \langle ms_0, ms_1, ..., ms_{k-1} \rangle$
    $TD$ is the set of threads in $AD$
    $td(a) \in TD$ is the thread of activity $a \in A$
    A mapping relation $AS$, $AS(s[i])$ is the related activity sequence of $s[i]$
Output:
    The revised activity diagram $AD$ which can accepted the execution trace $s$
1:  for i = 1 to k - 1 do
2:      $AS_1 = AS(s[i])$;
3:      $AS_2 = AS(s[i+1])$;
4:      find $T = [td_1, td_2, ..., td_m]$ as the common threads between $AS_1$ and $AS_2$;
5:      for j = 1 to m do
6:          let $l_{max} = MAX(\{l \in \mathbb{N} | td(AS_1[l]) = td_j\})$
7:          let $l_{min} = MIN(\{l \in \mathbb{N} | td(AS_2[l]) = td_j\})$
8:          if $\forall t \in T \bullet (AS_1[l_{max}], t) \notin F \vee (t, AS_2[l_{min}]) \notin F$ then {add a new transition to AD;}
9:              create a new transition $t_{new}$;
10:             $T = T \bigcup \{t_{new}\}$;
11:             $F = F \bigcup \{(AS_1[l_{max}], t_{new}), (t_{new}, AS_2[l_{min}])\}$;
12:         end if
13:     end for
14: end for
```

The algorithm above can treat most cases involving multi-control-flow activity diagrams and multi-threading programs. However, if the programs are modified to create a new thread, remove an original thread or change the first or last activity of a thread, the activity diagram can not be revised by our approach correctly. Using sequence analysis to identify the trace of an unknown thread is quite difficult, since an unknown thread's execution trace is interleaved with traces of its parent thread and many other threads.

D. remove out-of-date behavior

The last three steps will be repeated for every received execution trace. After all execution traces are received, new behaviors are added to the diagram, that is new activities and new transitions. The revised activity diagram can accept all previous traces finally. We consider that the final revised activity diagram can describe the behavior of evolved software correctly.

It should be pointed out that modifications made to software also may lead to disappearance of original behavior in software. For the accuracy of the revised diagram, those out-of-date parts need to be removed. We have executed a set of test cases and assumed that these test cases have covered all current software behavior, so, for those activities or transitions which have not been traversed by any execution traces, we consider that their related behaviors have disappeared in the current software version. Those activities and transitions are removed from the diagram.

The precondition of the correctness of removing out-of-date behaviors is that executed test cases have covered all current software behavior. The selection strategy of executed test cases must be taken into consideration to assure their coverage.

V. Regression Test Cases Generation

After the revision of the activity diagram is completed, the regression test suite is built on the base of the final revised activity diagram for the following regression testing. A certain count of test cases have been executed in the process of revision. There are two reasons why we still need to select test cases to build regression test suite and execute them. First, the execution of test cases in the process of activity diagrams revision is just used to explore the current behavior of software. These affected parts in software still need to be retested adequately by more test cases. And because some paths in activity diagrams have changed, the traceability between test cases and paths in diagrams should be updated. Second, modification in software may result in new paths in the revised diagram(like Figure 1), which may not have their corresponding covering test cases. New test cases need to be generated to cover these paths.

By applying automatic activity diagrams revision, up-to-date activity diagrams are built. Because all modifications in software are detected in the process of revision, the comparison between activity diagrams is not required. As a path in the revised diagram traverses any activities or transitions which are updated or newly created in the process of revision, it is considered affected by modifications.

If the execution of any test case involves software variation, its related covered path in the original diagram will be different from the one in the revised diagram. For each path in the activity diagram, we select some samples of its corresponding test cases from the pool to execute in the process of revision. If any of these test cases cover different paths in the final revised diagram, we consider behavior in the path is affected. All of its corresponding test cases in the pool are retestable in regression testing.

Besides, modifications made to software may result in new paths in the activity diagram. Some test cases generation approaches should be employed to generate test cases to cover these paths.

The final regression test suite is composed of the retestable test cases in the pool of test cases which cover affected paths and newly generated test cases which cover new paths in the activity diagram.

VI. Case Study

We use an online stocking exchange system(OSES) and its related activity diagram as the case study of this approach.

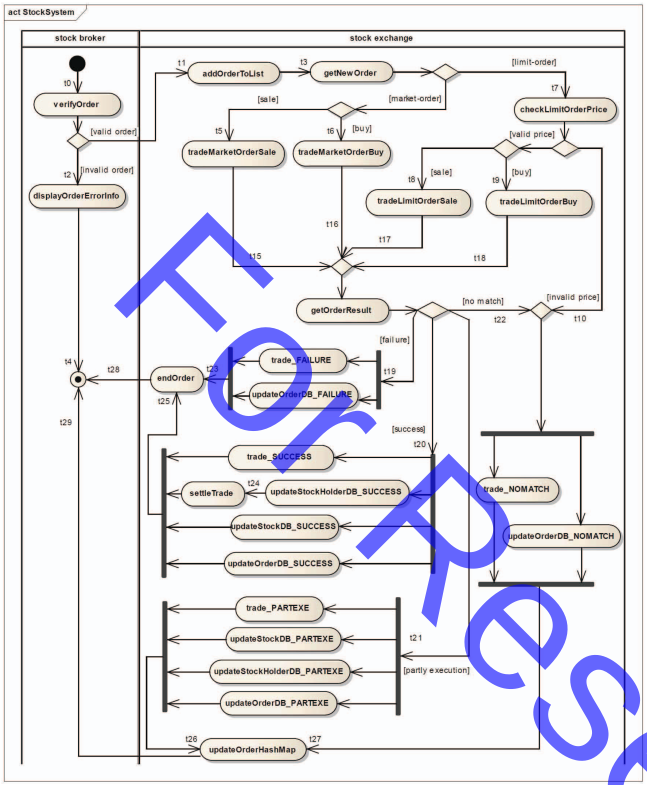

Figure 7. The activity diagram of OSES.

OSES is a JAVA program, reconstructed from an example in [1]. It consists of 40 classes and 305 methods. The activity diagram of OSES contains 25 activity nodes, 6 decision nodes and 18 paths totally(as Figure 7 shown). And OSES contains multi-thread behavior. We have proposed a pool of test cases which contains 1000 test cases to cover the 17 available paths in the diagram.

To show the effectiveness of the approach, 5 cases are designed to make modifications to OSES. These cases cover conditions of the addition and removal of method calls, the occurrence of new paths and the behavior variation in subthreads. The 5 cases are as following:

1) Add a new method call *checkMarketOrder* after the call *tradeMarketOrderBuy* and add a new method call *checkLimitOrder* after the call *tradeLimitOrderBuy* in the program. Execution traces involving modifications will appear like ⟨..., tradeMarketOrderBuy, checkMarketOrder, getOrderResult,...⟩ or ⟨..., tradeLimitOrderBuy, checkLimitOrder, getOrderResult,...⟩. Two new activities *checkMarketOrder* and *checkLimitOrder* should be added to the diagram.
2) Remove the *addOrderToList* call. Execution traces involving modifications will appear like ⟨verifyOrder, addNewOrder,...⟩. The activity *addOrderToList* should be removed from the diagram.
3) Add a new IF clause containing the method call *logout* after the *displayOrderErrorInfo* call. Execution traces involving modifications will appear like ⟨verifyOrder, displayOrderErrorInfo⟩ or ⟨verifyOrder, logout, displayOrderErrorInfo⟩. A new decision node should be added to the diagram, and a new activity *logout* should be in one branch from the decision node.
4) Add a new IF clause containing the method call *setMarketCode* and its corresponding ELSE clause containing the method call *setDefaultMarketCode* in front of the *tradeMarketOrderSale* call and the *tradeMarketOrderBuy*. Execution traces involving modifications will appear like ⟨..., getNewOrder, setMarketCode, tradeMarketOrderSale,...⟩, ⟨..., getNewOrder, setDefaultMarketCode, tradeMarketOrderSale,...⟩, ⟨..., getNewOrder, setMarketCode, tradeMarketOrderBuy,...⟩, or ⟨..., getNewOrder, setDefaultMarketCode, tradeMarketOrderBuy,...⟩. A new decision node should be added to the diagram, and two new activities *setMarketCode* and *setDefaultMarketCode* should be respectively in two branches of the decision node.
5) Add a new method call *checkInconsistency* between the call *settleTrade* and the call *updateStockHolderDB_SUCCESS*. All of three method calls are in one subthread of the program. Execution traces involving modifications will appear like ⟨ ..., updateStockHolderDB_SUCCESS,..., checkInconsistency,..., settleTrade,...⟩. A new activity *checkInconsistency* should be added between the activity *checkInconsistency* and the activity *settleTrade* in the diagram.

In the case study, for each path in the original diagram, 5 test cases are executed to collect execution traces. For each case, the revised activity diagram presents evolved behavior as expected. The retestable test cases and uncovered paths are given, as shown in Table I. In Table I, the column *affected paths(ori.)* presents the affected paths of the 18 paths in the original diagram; *retest cases* are test cases in the pool which cover these affected paths; *uncovered paths* are paths in the revised diagram which are not covered by any execution traces in the process of revision; the column *total paths* presents the count of paths of the revised diagram. Notice that because only 17 of 18 paths are available in the original activity diagram, there are always at least one uncovered paths in the revised diagram.

VII. Related Work

Model-based regression test selection(RTS) techniques receive growing interest in the area of regression testing ([6]). In the techniques, architecture or behavior models are introduced as the representations of software. By comparing different versions of models, modifications to software are

Table I
THE RESULT OF THE ABOVE 5 CASES

ID	affected paths(ori.)	retest cases	uncovered paths	total paths
1	8	338	1	18
2	16	963	1	18
3	1	37	1	19
4	7	703	5	26
5	4	230	1	18

located. Then test cases related with those changed elements are picked out as retestable ones. There have been some works [4],[5],[7] and [8] presenting this ideal. Undoubtedly, before those methods are adopted, models for each version of the software need to be constructed in advance. In practice, model-updating usually lags behind the software-modification. It requires software developers to make heavily manual efforts on updating models. Rather than manually updating models for modified software in those works, our approach can automatically generate the up-to-date behavior models by revising those out-of-date ones based on execution traces. It enables testers independently to perform regression testing without the involvement of developers. It can save a lot of manual efforts.

To achieve model-based regression test selection, modification-revealing test cases need to be selected in accordance with modifications. Thus, the traceability between models and test cases is necessary. In [7], a technique of RTS based on activity diagrams is proposed. The technique maps the activity diagram to the model used for regression test analysis, then selects test cases from existing test suites that traverse the different paths in the class behavior model extracted from the activity diagram. The work in [8] automates regression test selection based on architecture and design information represented with UML and traceability information linking the design to test cases. It performs regression test selection on the base of design change information, and can tackle the regression selection problem at the design level. The two approaches don't involve the traceability between models and test cases definitely. The work in [4] contributes an approach for selective model-based regression testing whereby traceability relationships between model elements and test cases traversing such elements are stored. Explicit fine-grain relationships from entities in models to abstract test cases persisted into a traceability infrastructure throughout the test generation process. The relationships are used to locate abstract test cases covering software modifications for regression testing. The selected abstract test cases need to be further transformed into concrete test cases. As these techniques depend on external traceability between models and test cases or corresponding relation between abstract and concrete test cases, regression test selection cannot be done automatically. By employing the approach in [3], we can get the covered paths for test cases by analyzing their execution traces and build the traceability automatically. Based on this, our approach can propose an automatic solution for regression test selection.

VIII. CONCLUSION

This paper proposed an approach for regression test cases generation based on automatic activity diagrams revision. In the approach, behaviors of evolved software can be built from the execution traces , and the out-of-date activity diagrams can be revised to accept the behavior variation introduced in evolved software. With the evolved software and the out-of-date activity diagram which describes the behavior of original unmodified software proposed, the new up-to-date diagram can be built whereby revision according to evolved software behavior. The regression test cases generation is done on the base of revision.

Compared with other modification-revealing approaches based on the comparison of models, this approach automates the whole process of identifying changed parts, selecting reused test cases and generating new test cases for testing new behaviors. It raises the availability of model-based regression testing.

REFERENCES

[1] M. Blaha, and J. Rumbaugh, *Object-oriented modeling and design with UML*. Pearson Education, 2005.

[2] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao, and X. Li. UML Activity Diagram-Based Automatic Test Case Generation For Java Programs. The Computer Journal, 2007.

[3] X. Chen, N. Ye, P. Jiang, L. Bu, and X. Li. Feedback-Directed Test Case Generation Based on UML Activity Diagrams. In *Proceeding of 5th International Conference on Secure Software Integration & Reliability Improvement Companion (SSIRI-C 2011)*, 2011, pp.9-10.

[4] L. Naslavsky, H. Ziv, and D.J. Richardson. MbSRT2: Model-Based Selective Regression Testing with Traceability. In *Proceeding of Third International Conference on Software Testing, Verification and Validation (ICST 2010)*, 2010, pp.89-98.

[5] N. Ye, X. Chen, P. Jiang, W. Ding, and X. Li. Automatic Regression Test Selection Based on Activity Diagrams. In *Proceeding of 5th International Conference on Secure Software Integration & Reliability Improvement Companion (SSIRI-C 2011)*, 2011, pp.166-171.

[6] S. Yoo, M. Harman. Regression testing minimization, selection and prioritization: a survey. In Software Testing, Verification and Reliability, Wiley InterScience, 2010, Published online: DOI: 10.1002/stvr.430

[7] E. Martins and V. G. Vieira. Regression test selection for testable classes. In *Proceedings of the 5th European Dependable Computing Conference(EDCC 2005)*, Springer, 2005, pp.453-470.

[8] L. C. Briand, Y. Labiche, and G. Soccar, Automating impact analysis and regression test selection based on UML designs, In *Proceedings of the 18th International Conference on Software Maintenance(ICSM 2002)*, IEEE Computer Society Press, 2002, pp. 252-261.