2016 IEEE 40th Annual Computer Software and Applications Conference

Improving Defect Detection Ability of Derived Test Cases Based on Mutated UML Activity Diagrams

Haiying Sun*, Mingsong Chen*, Min Zhang*, Jing Liu* and Ying Zhang†
*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
Email: {hysun,mschen,mzhang,jLiu}@sei.ecnu.edu.cn
†Software Engineering Institute, East China Normal University, Shanghai, China
Email: 10122510209@ecnu.cn

***Abstract*—Structure coverage driven test generation is the key approach for automatic testing at source code level. However, the defect detection ability of the generated test cases should be carefully evaluated since the correlation between coverage and test effectiveness is in doubt. In this paper, we propose a test generation approach based on mutation testing with the intent to derive test cases towards finding defects rather than just covering certain syntactic structures. Moreover, instead of generating test cases from the code under test directly, we base our approach on UML activity diagrams to make it possible to decide verdicts of test inputs. Mutation operators for activity diagrams are defined and the test generation algorithms are based on solving mutated path constraints. Experimental results have shown that by applying the proposed mutated testing approach, test cases with higher defect detection ability can be generated.**

I. Introduction

Finding defects is the fundamental purpose of each testing technique. Defect detection ability is the indictor used to measure the possibility of a set of test cases on finding defects. A test case set is said to be of high quality or high defect detection ability if it is more likely to find more defects than the others. Structure coverage criteria are a set of related testing adequacy rules which are widely applied in many code level test generation tools. Statement coverage, condition coverage and etc. are among the frequently used ones. These tools are surely efficient and can reduce testing effort. However, since they generate test cases directly from the program under test by applying expected coverage criteria, some questions should be answered clearly before putting them into practical testing activities.

The first question is how to evaluate a program under test passes or fails test cases when expected results are absent from these derived test cases. It is well known in traditional manual testing low-level design specification is one of the ideal test references when considering about code testing. However, as for coverage based test generation from source code, no but the program under test itself is the main reference. As a result, in our opinion, these derived test cases are towards executing the program under test rather than detecting detects hiding in it. And this question leads to the second one: how is the defect detection ability of the derived test case set. In fact, very recently, more and more researchers have questioned on the defect detection ability of test sets generated from coverage criteria [1], [2], [3], [4]. Moreover, some research results also point out that the correlation between coverage and test effectiveness is low to moderate [5].

The activity diagram is the modeling formalism in UML to emphasize the sequence and conditions for coordinating lower-level behaviors. Compared with the control flow graph (CFG) based formalism used in code based test generation approaches, the expressive power of activity diagrams is much stronger. Except that, a kernel set of activity modeling elements is armed with precisely defined semantics [6] which is crucial for qualified test case generation. Mutation testing is a defect-guided testing technique. In mutation testing, defects are deliberately seeded into the program under test by applying mutation operators. These defects represent certain kinds of mistakes that a software engineer is mostly like to make when programming. According to the fundamental premise of mutation testing, if a test case set can find these inserted defects, it can also find the corresponding real defects. Because mutation testing requires not only executing expected program behaviors but also producing incorrect outputs, it is considered superior to structure coverage criteria [7].

To derive test cases towards detecting defects rather than just covering expected certain kind of code structures, in this paper, we propose a test generation method based on mutated activity diagrams. That is, instead of generating test cases from the code under test directly, we not only take activity diagrams as the test references but also deliberately insert certain kinds of defects into them to guide qualified test cases generation.

II. Mutation Operators for Activity Diagrams

Mutation operators are at the heart of mutation testing. In this section, according to the intended fault models and the foundational subset of the UML definition [6], we design mutation operators for activity diagrams.

A. Modeling Elements in Activity Diagrams

In UML, an activity diagram is the graphical notation of an activity. An activity models behavior by defining the transformation of inputs to outputs through a controlled sequence of actions. An action represents a single step within an activity which can't be further decomposed. In the research, we base our method on the foundational subset of the standard UML activity models for it is armed with precisely defined executable semantics [6]. Figure 1 illustrates the corresponding model structures.

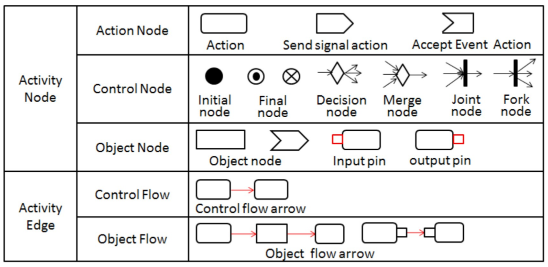


Fig. 1. Graphics Notation of Modeling Elements in Activity Diagram

B. The Fault Models

A mutation operator is a predefined rule to introduce a certain kind of fault into the object under test. Therefore, the first step on designing a mutation operator set is to identify valuable fault models. The purpose of mutating an activity diagram in our method is to generate defect-detecting test cases for the program under test rather than validate the activity diagram itself. As a result, the selected fault models are programming language based. However, different programming languages have different fault models, it is impossible to include all. In the paper, we only synthesize three categories of common fault models for programming languages. It is surely feasible to add new mutation operators with certain testing objectives to this set for specific programming language.

- Expression Error: errors made in formulating an expression, such as misused operators, variables and constants;
- Control Structure Error: errors made in using iterative, conditional and concurrent statements;
- Integration Error: errors made in the connection of individual code segments.

Because we don't include structure modeling information, such as class diagram, in our method, the encapsulation, inheritance, polymorphism, dynamic binding related fault models are excluded by now.

C. The Mutation Operators

Over the years, researchers have designed mutation operators for various programming languages and several specification languages. However, there is none for activity diagrams. This is perhaps because mutation testing is traditionally used more on program testing than specification. Although it is reasonable to include as many mutation operators as possible for a language when first designing its mutation operators, it has been already realized that a large set of mutation operators will lead to too many redundant mutants which may not only sharply increase the testing cost but also inflate the result [8].

Selective mutation is a reduced mutation approach focusing on establishing an effective subset of mutation operators. Given a mutation operator set O and one of its subset $S \subseteq O$, if test cases that are created specifically to kill mutants created by S also kill mutants created by O, then S defines an effective subset of O. To provide an effective mutation operator set, we base our design on summarizing typical selective mutation studies [9], [10], [11], [12]. The result is shown in Table I. According to the syntactic definition of activity diagrams, we construct five classes of mutation operators:

1) action node mutation operators: ANDO and ANRO.
2) control node mutation operators: DNDO, FJDO, FJAO, FJUO and FDJU.
3) object node mutation operators: ONDO and ONRO.
4) flow mutation operators: SNEO and DNEO.
5) expression mutation operators: EORO, EENO, ELRO, ERRO, ESRO and EBRO.

Since parameters are special kinds of object nodes, the mutants produced by changing parameters for detecting integration errors can be generated by applying object node mutation operators. By now, we don't support initial node and final node mutation but the preconditions and postconditions of an activity diagram can be mutated by using expression mutation operators because they are essentially expressions.

In our approach, the activity diagram mutants are calculated by applying first-order mutation. That is, only a single change is introduced into the original activity diagram. Though we have no evidence to prove the first-order mutation for activity diagrams has similar effects with that of high-order mutation. However, this approach is the simplest and is applied widely in program mutation testing.

III. Test Case Generation

Given an activity diagram $\mathcal{A}$ and the mutation operator set $\mathcal{O}$, our proposed test case generation process based on mutated activity diagrams is composed of the following steps

1) For $\forall o \in \mathcal{O}$, generate set of mutants $\sum \mathcal{A}'$ for $\mathcal{A}$ by applying first order mutation
2) For $\forall \mathcal{A}' \in \sum \mathcal{A}'$, if $\mathcal{A}'$ is equivalent to $\mathcal{A}$, then $\sum \mathcal{A}' = \sum \mathcal{A}' - \mathcal{A}'$
3) For $\forall \mathcal{A}' \in \sum \mathcal{A}'$, calculate set of prime activity diagrams $\sum \mathcal{A}'_p$ for $\mathcal{A}'$
4) For $\forall \mathcal{A}'_p \in \sum \mathcal{A}'_p$, calculate path constraint set which may weakly kill the mutant. If there are forks and joints nodes in $\mathcal{A}'_p$, the constraints are calculated by calculating representative paths otherwise by basic paths

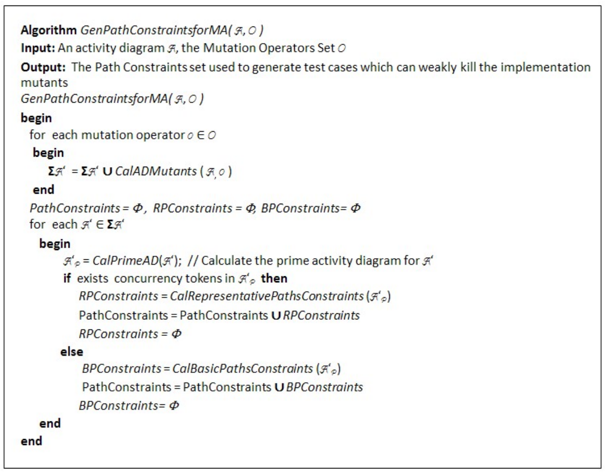
Figure 2 is the main algorithm. The path constrains are solved with the Yices SMT solver.

A. Prime Activity Diagram

Prime activity diagrams are used to deal with infinite paths that may occur in an activity diagram. Given an activity diagram $\mathcal{A}$, a *path* of $\mathcal{A}$ is a sequence of action sets where each action set is a sequence of action nodes which may occur concurrently and each pair of adjacent actions in a action set and each pair of adjacent action sets in the path should satisfy their corresponding transition relations and guard conditions defined in $\mathcal{A}$. For example, the action set sequence $< a >, < b, c >, < e >, < g >, < h >, < b >, < f >, < i, j >$ is a path of the activity diagram defined in Figure 5. In the path, the action set $< b, c >$ is composed of two action nodes b and c which means action b and c can happen concurrently. A path is called a *basic path* if no action set appears more than once in the path. A *prime path* is a basic path and it should not

TABLE I
MUTATION OPERATORS FOR ACTIVITY DIAGRAM

Mutation Operator	Description
ANDO (**A**ction **N**ode **D**eletion **O**perator)	Each action (including called action) is replaced by a special action named Failed which should report a failure
ANRO (**A**ction **N**ode **R**eplacement **O**perator)	Each action (including called action) is replaced by another action defined within its scope
DNDO (**D**ecision **N**ode **G**uard **O**perator)	Each guard condition of decision node is replaced with boolean constant *true* and *false*
FJDO (**F**ork and **J**oint **N**ode **R**emovement **O**perator)	Remove each pair of fork and joint nodes in the activity diagram
FJAO (**F**ork and **J**oint **N**ode **A**ddition **O**perator)	Add a new pair of fork and joint within an existing pair of fork and joint in the activity diagram
FJUO (**F**ork and **J**oint **N**ode **D**own **O**perator)	Shift down each pair of fork and joint nodes
FDJU (**F**ork **D**own and **J**oint **U**p)	Shift down the fork node while lift up the joint node. That is, the concurrency region is shrink
ONDO (**O**bject **N**ode **D**eletion **O**perator)	Delete input pin and output pin (including parameters of the activity) of an action one by one
ONRO (**O**bject **N**ode **R**eplacement **O**perator)	Each input pin and output pin of an action (including parameters of the activity) is replaced by another
SNEO (**S**ource **N**ode **E**xchange **O**perator)	Modifying the source node of an object flow by replacing to another output pin of the owner action node of the source node
DNEO (**D**estination **N**ode **E**xchange **O**perator)	Modifying the destination node of an object flow by replacing to another input pin of the owner action node of the destination node
EORO (**O**perand **R**eplacement **O**perator)	Replace an operand of the expression by another syntactically legal operand
EENO (**E**xpression **N**egation **O**perator)	Replace an expression (including its subexpression) by its negation
ELRO (**L**ogical Operator **R**eplacement **O**perator)	Replace a logic operator by each of another logical operator
ERRO (**R**elational Operator **R**eplacement **O**perator)	Replace a relational operator by each of another relational operator
ESRO (**S**hift Operator **R**eplacement **O**perator)	Replace a shift operator by each of another shift operator
EBRO (**B**itwise Operator **R**eplacement **O**perator)	Replace a bitwise operator by each of another bitwise operator



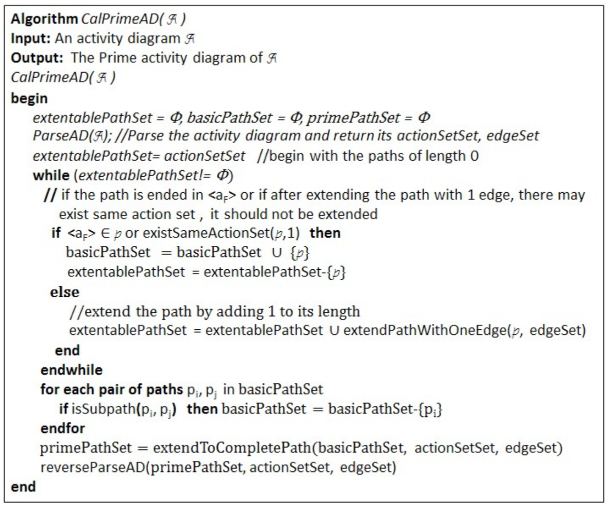
```
Algorithm GenPathConstraintsforMA( 𝒜, 𝒪 )
Input: An activity diagram 𝒜, the Mutation Operators Set 𝒪
Output: The Path Constraints set used to generate test cases which can weakly kill the implementation
mutants
GenPathConstraintsforMA( 𝒜, 𝒪 )
begin
  for each mutation operator o ∈ 𝒪
   begin
     Σ𝒜' = Σ𝒜' ∪ CalADMutants ( 𝒜, o )
   end
  PathConstraints = Φ, RPConstraints = Φ, BPConstraints= Φ
  for each 𝒜' ∈ Σ𝒜'
    begin
        𝒜'ₚ = CalPrimeAD(𝒜'); // Calculate the prime activity diagram for 𝒜'
        if exists concurrency tokens in 𝒜'ₚ then
          RPConstraints = CalRepresentativePathsConstraints (𝒜'ₚ)
          PathConstraints = PathConstraints ∪ RPConstraints
          RPConstraints = Φ
       else
          BPConstraints = CalBasicPathsConstraints (𝒜'ₚ)
          PathConstraints = PathConstraints ∪ BPConstraints
          BPConstraints= Φ
    end
end
```

Fig. 2. Generating Path Constraints From An Activity Diagram and the Mutation Operators Set

appear as a subpath of any other basic path. If the first action set of a path is composed of the initial action node and the last action set is composed of the final action node, the path is called as a *complete path*. Given an activity diagram $\mathcal{A}$, its *prime activity diagram* is composed of the complete path set which covers all the prime paths of $\mathcal{A}$. Figure 3 presents the algorithm to generate a prime activity diagram. The algorithm begins with the paths of length 0 and then extends the path by increasing its length 1 a time according to the transition relation until all paths can't be extended any more.

The prime activity diagram can solve the infinite path problem imposed by iterations but it can't solve the many path combinations caused by concurrency. Therefore, we introduce the concept of representative path. In a prime activity diagram, if there exists many complete prime paths satisfying:

1) These paths have the same action set sequence
2) For the corresponding action sets in these paths, they have same action nodes but different sequence



```
Algorithm CalPrimeAD( 𝒜 )
Input: An activity diagram 𝒜
Output: The Prime activity diagram of 𝒜
CalPrimeAD( 𝒜 )
begin
   extentablePathSet = Φ, basicPathSet = Φ, primePathSet = Φ
   ParseAD(𝒜); //Parse the activity diagram and return its actionSetSet, edgeSet
   extentablePathSet= actionSetSet  //begin with the paths of length 0
   while (extentablePathSet!= Φ)
     // if the path is ended in <a_F> or if after extending the path with 1 edge, there may
        exist same action set , it should not be extended
      if <a_F> ∈ p or existSameActionSet(p,1)  then
        basicPathSet = basicPathSet ∪ {p}
        extentablePathSet = extentablePathSet-{p}
      else
         //extend the path by adding 1 to its length
         extentablePathSet = extentablePathSet ∪ extendPathWithOneEdge(p, edgeSet)
       end
    endwhile
    for each pair of paths p_i, p_j in basicPathSet
       if isSubpath(p_i, p_j)  then basicPathSet = basicPathSet-{p_i}
    endfor
    primePathSet = extendToCompletePath(basicPathSet, actionSetSet, edgeSet)
    reverseParseAD(primePathSet, actionSetSet, edgeSet)
end
```

Fig. 3. Prime Activity Diagram Generation

only one of them will be selected as representative path. For example, there are overall 33 complete prime paths in its prime activity diagram of Figure 5. For the following four paths, only one (any one is permitted) will be selected:

1) $<a>, <b, c>, <e>, <g>, <h>, <b>, <f>, <i, j>$
2) $<a>, <c, b>, <e>, <g>, <h>, <b>, <f>, <i, j>$
3) $<a>, <b, c>, <e>, <g>, <h>, <b>, <f>, <j, i>$
4) $<a>, <c, b>, <e>, <g>, <h>, <b>, <f>, <j, i>$

B. Path Constraint Generation

Path constraint is a predicate used to define the conditions under which the path may be executed. A path constraint is the conjunction of each condition appears along with a complete path. Calculating path constraints which may kill the mutants is crucial in the test case generation process.

According to the definition of mutation testing, a mutant is killed if and only if the output of a mutant is differently from that of the original one. However, it has been pointed out that this standard is too expensive and hard to be automated [13].

TABLE II
CONSTRAINTS FOR WEAKLY KILLING MUTANTS

Mutation Operator	Constraints Rules
ONRO	$\forall O_1, O_2 \in A_O$, $pin(O_1)! = pin(O_2)$
SNEO	$\forall opin_1, opin_2 \in O$, $opin_1 \rightarrow opin_1! = opin_2$
DNEO	$\forall ipin_1, ipin_2 \in O$, $ipin_1 \rightarrow ipin_1! = ipin_2$
EORO	$\forall v_1, v_2 \in V$, $v_1 \rightarrow v_1! = v_2$
EENO	$e \rightarrow \neg e$
ELRO	$\forall$ **op** $\in \{\&\&, \vert\vert\}$, v_1**op**$v_2 \rightarrow \forall$ **op**$_m \in \{\&\&, \vert\vert\}\backslash$**op**, v_1**op**$_m v_2$
ERRO	$\forall$ **op**$\in \{<, >, =, >=, <=, ! =\}$, v_1**op**$v_2 \rightarrow \forall$ **op**$_m \in \{<, >, ==, >=, <=, ! =\}\backslash$**op**, $(!(v_1\mathbf{op}v_2) \wedge (v_1\mathbf{op}_m v_2)) \vee ((v_1\mathbf{op}v_2) \wedge !(v_1\mathbf{op}_m v_2))$
ESRO	$\forall$ **op** $\in \{\ll, \gg\}$, v_1**op**$v_2 \rightarrow \forall$ **op**$_m \in \{\ll, \gg\}\backslash$**op**, v_1**op**$_m v_2$
EBRO	$\forall$ **op** $\in \{\&, \vert, \hat{}\,\}$, v_1**op**$v_2 \rightarrow \forall$ **op**$_m \in \{\&, \vert, \hat{}\,\}\backslash$**op**, v_1**op**$_m v_2$

TABLE III
TCAS MUTANTS

Applied Mutation Operator	Mutants Number
OAAN (arithmetic operator mutation)	4
OLRN (logical operator by relational operator)	102
ORLN (relational operator by logical operator)	30
ORAN (relational operator by arithmetic operator)	75
OLLN (logical operator mutation)	17
ORRN (relational operator mutation)	75
OLNG (logical negation)	47
SSDL (statement deletion)	10
Total	360

TABLE IV
THE JAVA PROGRAM MUTANTS

Applied Mutation Operator	Mutants Number
AOR$_B$ (basic binary arithmetic operators replacement)	12
AOI$_U$ (insert basic unary arithmetic operators)	9
AOI$_S$ (insert short-cut arithmetic operators)	96
AOD$_U$ (delete basic unary arithmetic operators)	2
ROR (relational operator replacement)	126
COR (conditional operator replacement)	4
COD (conditional operator deletion)	3
COI (conditional operator insertion)	22
LOI (logical operator insertion)	27
ASR$_S$ (short-cut assignment operator replacement)	8
MSP (modify synchronized block parameter)	2
RTXC (remove thread method-X call)	4
RSK (remove synchronized keyword from method)	33
Total	348

Therefore, the most majority of mutation testing approaches adopt weak mutation criterion which only requires to change the internal state of the mutants immediately after executing the mutated point. In order to calculate test cases which may weakly kill the mutants, special conditions should be added. Table II lists the mutation operators and their corresponding weakly killed constrains. A_O is the object node set of an activity diagram $\mathcal{A}$, O is an object node, V is the variables set, $ipin_i$, $opin_i$, v_i denote input pin, output pin and variable respectively, **op** is an operator. These special constraints are defined as the result of extending Zhang's constraint generation rules [7]. Because the action node mutation operators: ANDO, ANRO and control node mutation operators: DNDO, FJDO, FJAO, FJUO, FDJU may change the executing sequence of the code, mutants generated from these operators can be surely weakly killed due to different program counters. The ONDO operator changes the number of variables, as a result, the internal states of mutants generated from ONDO are different from that of the original one.

IV. EXPERIMENTAL STUDY

To validate our approach, we have conducted three experiments on the TCAS program and a concurrent Java program described in [14]. These two programs are selected because of their distinct characteristics. The former is a typical process-oriented program with complex control logics while the latter is an object-oriented program with concurrent behaviors. We deliberately select programs written in different languages to study the generality of our approach since activity diagrams are independent of implementation technology.

The downloaded TCAS package has already included a set of 41 mutants. But since we don't know which mutation operators are used to generate these 41 mutants, we also generate another mutant set as shown in Table III. These mutants are generated manually as we only want to applying those mutation operators which are studied to be efficient on safety critical software [2]. Except for concurrent mutants, the Java program mutants are generated by using $\mu Java$ as shown in Table IV. Those mutation operators which generate zero mutants aren't listed in the table. Concurrent mutants are generated manually since we can't find any available concurrency mutation tool.

The process of our experiments can be divided into two phases. In the first phase, we first use random testing to derive the initial test cases and select the ones which are satisfied with the expected coverage. Then, we evaluate their defect detection ability by calculating mutation scores against the program mutants. In the second phase, we apply our proposed mutated approach to generate test cases and calculate their mutation scores as that of the first phase. The experimental results have shown that by applying the mutated testing approach, the mutation scores of the derived test sets in the case studies are improved about 70.4%, 36.6% and 21.6% respectively.

A. Experiments on TCAS

TCAS is a safety critical system equipped on commercial aircrafts to avoid a potential collision. Figure 4 is its functional flow defined in an activity diagram. We have conducted two experiments on TCAS. The first experiment is based on the downloaded 41 mutants and the second one is based on our generated 360 mutants.

The first task for the TCAS experiments is to write a program which randomly generates test cases satisfying the expected coverage criteria. We have tried to generate 100, 200, 400, 1000 and 5000 random test cases and found the 400 set is

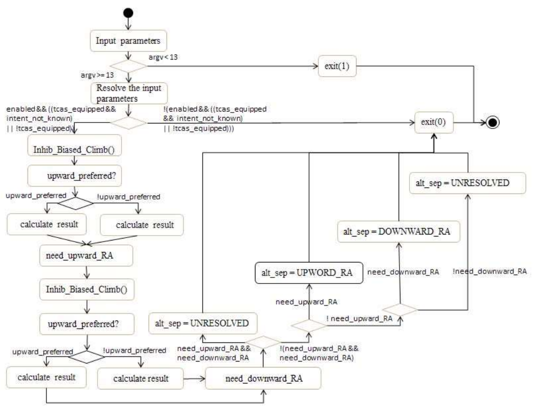


Fig. 4. The Activity Diagram of TCAS

TABLE V
THE 6 COVERAGE SATISFIED TESTS AGAINST THE 41 TCAS MUTANTS

Total Tested Mutants	Mutants killed	Equivalent Mutants	Mutants Survived	Mutation Score
41	17	0	24	41.50

TABLE VI
THE 6 COVERAGE SATISFIED TESTS AGAINST THE 360 TCAS MUTANTS

Mutation Operator	Total Mutants	Mutants killed	Equivalent Mutants	Mutants Survived	Mutation Score
OAAN	4	3	0	1	75.00
OLRN	102	43	25	34	55.84
ORLN	30	12	8	10	54.55
ORAN	75	29	20	26	52.73
OLLN	17	7	0	10	41.17
ORRN	75	34	20	21	61.82
OLNG	47	32	4	11	74.41
SSDL	10	4	2	4	50.00
Total	360	164	79	117	58.36

the most effective as it is satisfied with 91% condition/decision coverage and 93% decision coverage with the least test case number(Because there is an infeasible path in the origin TCAS program, the coverage ratio can never achieve 100%). Then, we select a minimal subset from this 400 set which can keep the coverage ratio unchanged since not all of the 400 test cases contribute to the expected coverage and this set includes 6 test cases. In the third step, we execute these 6 test cases against the 41 and 360 mutants to evaluate its defect detection ability by calculating mutation scores. The results are shown in Table V and Table VI. It can be seen from the tables that the average mutant scores of the 6 test cases set are 41.5% and 58.36%.

In the remaining experiment steps, we apply our proposed approach to derive test cases. According to its structure, 4 out of the 17 mutation operators can be applied to the TCAS activity diagram and 165 feasible mutated path constraints are generated which is shown in Table VII. These path constraints are then translated into the Yice-readable format to calculate the corresponding test cases. After deleting duplicated outputs, we finally get 46 test cases and the mutation score of this test set against the 41 and 360 mutants are 70.73% and 79.71% as shown in Table VIII and Table IX. Therefore, the mutation scores have been improved about 70.4% and 36.6%.

TABLE VII
THE TCAS ACTIVITY DIAGRAM MUTANTS

Applied Mutation Operator	Number of Mutants	Number of feasible path	Number of Infeasible path
ANDO	32	16	16
EENO	9	7	2
ELRO	48	37	11
ERRO	108	105	3

TABLE VIII
THE 46 MUTATION BASED TESTS AGAINST THE 41 TCAS MUTANTS

Total Tested Mutants	Mutants killed	Equivalent Mutants	Mutants Survived	Mutation Score
41	29	0	12	70.73

TABLE IX
THE 46 MUTATION BASED TESTS AGAINST THE 360 TCAS MUTANTS

Mutation Operator	Total Mutants	Mutants killed	Equivalent Mutants	Mutants Survived	Mutation Score
OAAN	4	4	0	0	100.00
OLRN	102	49	25	28	63.64
ORLN	30	20	8	2	90.91
ORAN	75	50	20	5	90.91
OLLN	17	11	0	6	64.71
ORRN	75	48	20	7	87.27
OLNG	47	35	4	8	81.40
SSDL	10	7	2	1	87.5
Total	360	224	79	57	79.71

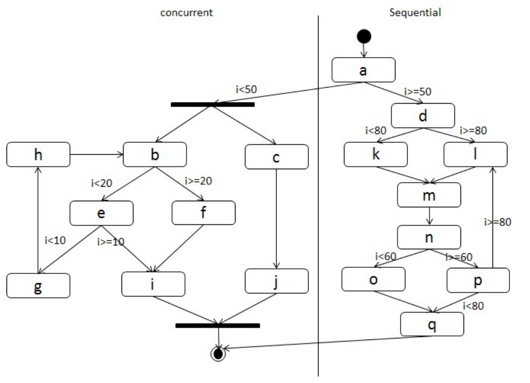


Fig. 5. Activity Diagram of the Java program

B. Experiment on a Java Program

The main function of the java program is to record its concurrent behavior traces into a log file. Fig. 5 is its activity diagram [14]. In the activity diagram, an action node denotes a method of the corresponding class which appends its name to the current concurrent trace in the log file. As that of the TCAS experiment, we write a program to randomly generate test cases satisfying with the condition coverage. We have tried 50,100 and 200 test cases and found the 100 set is the most effective. In the next step, 5 out of the 100 test cases are selected as the final set which can achieve condition coverage with the least size. We then evaluate this test set against the 348 mutants and Table X presents the corresponding results. It can be seen that the average mutation score is 81.73% which is much better than that of the TCAS. After careful analysis, we conclude that this is because the logical complexity of TCAS is higher than that of the java program.

TABLE X
THE 5 COVERAGE SATISFIED TESTS AGAINST THE 348 JAVA MUTANTS

Mutation Operator	Total Mutants	Mutants killed	Equivalent Mutants	Mutants Survived	Mutation Score
AOR$_B$	12	10	0	2	83.33
AOI$_U$	9	7	0	2	77.78
AOI$_S$	96	67	1	28	70.53
AOD$_U$	2	2	0	0	100.00
ROR	126	106	2	18	85.48
COR	4	2	0	2	50.00
COD	3	3	0	0	100.00
COI	22	18	0	4	81.82
LOI	27	24	0	3	88.89
ASR$_S$	8	6	0	2	75.00
MSP	2	2	0	0	100.00
RTXC	4	4	0	0	100.00
RSK	33	31	0	2	93.94
Total	348	282	3	63	81.73

TABLE XI
MUTANTS OF THE JAVA PROGRAM ACTIVITY DIAGRAM

Mutation Operator	Number of Mutants	Feasible Paths	Infeasible Paths
EENO	35	15	20
ERRO	175	108	67
DNDO	70	35	35
FJDO	1	8	0
FDJU	2	10	0
Total	283	176	122

TABLE XII
THE 14 MUTATION BASED TESTS AGAINST THE 348 JAVA MUTANTS

Mutation Operator	Total Mutants	Mutants killed	Equivalent Mutants	Mutants Survived	Mutation Score
AOR$_B$	12	12	0	0	100.00
AOI$_U$	9	9	0	0	100.00
AOI$_S$	96	93	1	2	97.89
AOD$_U$	2	2	0	0	100.00
ROR	126	124	2	0	100.00
COR	4	4	0	0	100.00
COD	3	3	0	0	100.00
COI	22	22	0	0	100.00
LOI	27	27	0	0	100.00
ASR$_S$	8	8	0	0	100.00
MSP	2	2	0	0	100.00
RTXC	4	4	0	0	100.00
RSK	33	33	0	0	100.00
Total	348	343	3	2	99.42

Because of its structure characteristics, 5 out of 15 activity diagram mutation operators can be applied to the java program activity diagram. These operators generate 283 mutants and Yices solves 176 which is shown in Table XI. After deleting the duplicate test cases, we get 14 test cases and its mutation score against the 348 mutants is 99.42% which is improved about 21.6%. Table XII provides more details.

V. CONCLUSION

Activity diagrams support both low level behavior modeling and system level process modeling with formal token flow semantics. Therefore, we regard activity diagrams as a promising modeling formalism for model-driven testing. Mutation based test generation is considered as a powerful method on finding defects. Recently, more and more researchers have applied mutation ideas to their testing methods.

In the paper, we propose a test generation method based on mutated activity diagrams. Our intention is to optimize the derived test set towards finding defects rather than just covering certain syntactic structures. A set of mutation operators are defined for activity diagrams. Test cases are generated by solving mutated activity path constraints which include conditions for weakly killing the relative mutants. Three experiments have been conducted and the corresponding results show that test cases with higher defect detection ability can be derived. Although the test generation method is demonstrated on code testing, it also can be applied to system level testing. Corresponding experiments are under preparing.

ACKNOWLEDGMENT

The authors would like to thank anonymous reviewers for their comments and suggestions. This paper is partially supported by the projects funded by the NSFC Trustworthy Software Track 91318301, NSFC 91418203, NSFC 61502171 and NSFC Creative Team 61321064. The Shanghai Trustworthy Computing Key Lab is supported by Shanghai Knowledge Service Platform ZF 1213, NSFC Creative Team 61321064 and Shanghai Project (No.13511503100, No.012FU125X15).

REFERENCES

[1] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, “The risks of coverage-directed test case generation,” *Software Engineering, IEEE Transactions on*, vol. 41, no. 8, pp. 803–819, 2015.

[2] R. Baker and I. Habli, “An empirical evaluation of mutation testing for improving the test quality of safety-critical software,” *Software Engineering, IEEE Transactions on*, vol. 39, no. 6, pp. 787–805, 2013.

[3] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, “On the danger of coverage directed test case generation,” in *Fundamental Approaches to Software Engineering*, 2012, pp. 409–424.

[4] G. Fraser and A. Zeller, “Mutation-driven generation of unit tests and oracles,” in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010, pp. 147–158.

[5] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 435–445.

[6] Omg, “Semantics of a foundational subset for executable uml models v1.1,” 2013.

[7] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei, “Test generation via dynamic symbolic execution for mutation testing,” in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[8] P. Ammann, M. E. Delamaro, and J. Offutt, “Establishing theoretical minimal sets of mutants,” in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, 2014, pp. 21–30.

[9] W. Wong, J. Maldonado, and et al., “A comparison of selective mutation in c and fortran,” in *Proceedings of Workshop Validation and Testing of Operational Systems Project*, 1997, pp. 71–84.

[10] E. F. Barbosa, J. Maldonado, and A. M. R. Vincenzi, “Toward the determination of sufficient mutant operators for c,” *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 113–136, 2001.

[11] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, “Sufficient mutation operators for measuring test effectiveness,” in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 351–360.

[12] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, “Selective mutation testing for concurrent code,” in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 224–234.

[13] W. E. Howden, “Weak mutation testing and completeness of test sets,” *IEEE Tran. on Soft. Eng.*, vol. 8, no. 4, pp. 371–379, 1982.

[14] C. Mingsong, Q. Xiaokang, and L. Xuandong, “Automatic test case generation for uml activity diagrams,” in *Proceedings of the 2006 International Workshop on Automation of Software Test*, 2006, pp. 2–8.