UDC 004.415.53+004.832.23+004.052.3 https://doi.org/10.15407/pp2020.02-03.331

EFFICIENT INCREASING OF THE MUTATION SCORE DURING MODEL-BASED TEST SUITE GENERATION

A. Kolchin, S. Potiyenko, T. Weigert

The purpose of the method is to increase the sensitivity of an automatically generated test suite to mutations of a model. Unlike existing methods for generating test scenarios that use the mutational approach to assess the resulting test set, the proposed method analyzes the possibility of detecting mutations on the fly, in the process of analyzing the model's behavior space, by adding of special coverage goals. Two types of mutants manifestation are considered: deviations in the behavior of paths for (weak case) and in the observed output (strong case). A new algorithm is proposed for efficient search of a path with observable effect of a mutation.

Key words: testing, model checking, mutation testing.

Мета методу – підвищити чутливість автоматично генерованого тестового набору до мутацій моделі. На відміну від існуючих методів генерації тестових сценаріїв, які використовують мутаційний підхід для оцінки отриманого тестового набору, запропонований метод аналізує можливість виявлення мутацій «на льоту», в процесі аналізу простору поведінки моделі, додаючи спеціальні цілі покриття. Розглядаються два види прояву мутацій: відхилення в поведінці шляхів (випадок слабкого виявлення) і в спостережуваних вихідних сигналах (випадок сильного виявлення). Запропоновано новий алгоритм для ефективного пошуку шляху до спостереження ефекту мутації.

Ключові слова: тестування, перевірка моделі, мутаційне тестування.

Цель метода – повысить чувствительность автоматически генерируемого тестового набора к мутациям модели. В отличие от существующих методов генерации тестовых сценариев, которые используют мутационный подход для оценки полученного тестового набора, предложенный метод анализирует возможность обнаружения мутаций «на лету», в процессе анализа пространства поведения модели, добавляя специальные цели покрытия. Рассматриваются два вида проявления мутаций: отклонение в поведении путей (случай слабого обнаружения) и в наблюдаемых выходных сигналах (случай сильного обнаружения). Предложен новый алгоритм для эффективного поиска пути ведущего к наблюдаемому эффекту мутации.

Ключевые слова: тестирование, проверка модели, мутационное тестирование.

1. Introduction

Test cases efficiency problem. Testing is the most used way of assessing quality in the software industry. Manual testing is labour intensive and often inefficient. Model-based development often considers automated test generation from the model as a form of requirements-based testing and also resolves the oracle problem. Commonly, test suite effectiveness is associated with its fault detection ability, while efficiency is measured by taking the ratio of the number of tests over the number of faults found. The majority of test generation approaches use some structural coverage criterion based on a behavioral model of the SUT to guide the selection of test cases [1]. However, whether or which coverage criterion best guides software testing towards fault detection remains a controversial and open question [2, 3]. Moreover, many empirical studies [2, 4, 5] suggest that the level of structural coverage itself is not a good indicator of the effectiveness of a test suite. For example, in [5], authors made MC/DC coverage for a model of a flight guidance system, and then executed the tests on implementations that had been seeded with errors. They found that the auto generated tests detected relatively few bugs due to absence of observability effect in test cases, and generally performed even worse than random testing.

Mutation Testing is a fault-based testing technique which provides a coverage criterion called the "mutation adequacy score". The mutation score is a widely used assessment of estimating the ability to reveal defects by a test suite [8]. The score is defined as the percentage of killed mutants with the total number of (non-equivalent) mutants. Mutation Testing is considered as an effective approach to identifying of adequate test data which can be used to find real faults [7, 8]. The main drawback of the mutation assessment is the high cost of generating the mutants and executing each test case against each mutant program.

Test cases generation problem. Reachability checking of required coverage criteria is a research topic of increasing importance, see e. g., [8–13]. Problems with decidability and performance encountered in the development of automating software testing techniques have stimulated different research approaches [9]: stochastic and combinatorial methods are easy to implement and are fast, but result in poor coverage and high redundancy. Genetic algorithms [10, 11] enhance coverage by selecting more promising test populations. Many systems implement hybrid and heuristic strategies. For example, systematic methods [12–16], including the proposed approach, extract constraints for executing model paths and obtain test inputs that direct model behavior along these paths. For the observability, such path shall contain some data flow chain from the fault to its visible output.

For model-based test generation, a test goal with regards to a chosen coverage criterion could be specified through a temporal logic formula which is typically represented in the form "always not p". Using different methods of model checking [1, 9, 18], it is possible to produce a counterexample trace, which violates the required property p. However, explicit specifying of a temporal logic formula for a mutant which shall be manifested in an output signal

may turn to be impractically unwieldy: model may potentially include huge amount of different ways leading to observation of the mutant, and, while it is sufficient to find only one of them, such approach will require their specification, additionally complicating the search task.

Reachability analysis can be augmented by automata based models, where the information about data flow chain to be covered is maintained in auxiliary automata [13, 14]. For example, in [15] a specialized decision procedure for early termination of path unfolding is proposed, which allows exploration of a state only if it might increase the requested data flow coverage, resulting in more efficient state-space exploration.

Proposed solutions. Our algorithm tends to select inputs that cause each mutant to fail making a difference in the mutated model behavior and also in its visible output. Briefly, our approach to the mutation score increasing is based on the following steps: (1) generation of auxiliary goals to reach mutations directly, (2) paths prolongation to make the mutation effect observable via coverage of some data flow chain, (3) considering info about mutants killed by each test case during test suite minimization.

The rest of the paper is organized as follows. Section 2 is devoted to mutation-based approach and common fault types, section 3 describes auxiliary goals to reach the mutations and the main algorithm, section 4 presents an approach for ensuring observability of the mutation effect and a new algorithm for data flow chain coverage, section 5 discusses related work, section 6 concludes.

2. Mutation-based approach

Commonly, mutation analysis is used as a method for estimation of ability to reveal faults. Its theory is based on two hypotheses: the Competent Programmer Hypothesis and Coupling Effect [19]. The first states that programmers are competent, which implies that they tend to develop programs close to the correct version. Coupling Effect concerns the type of faults used in mutation analysis. It states that "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors" [7]. Mutation analysis deliberately introduces artificially-generated faults into the original model, which are called 'mutants'. Commonly it is assumed that each mutant should contain a single fault which is a small syntactical change so that it does not affect the overall objective of the program.

Types of Mutation Testing. Many observations of real faults conclude that typical programmer errors include missing or extra conditions and use of incorrect operator or operands. For example, the following fault types can be hypothesised:

Arithmetic: Changes an arithmetic operator (+, -, /,*, mod, exp).

Relational: Changes a relational operator (=, !=, <, >, <=, >=).

Value mutation: These faults involves modification of values or constants:

a. Stuck at 0: A primary parameter is always set to 0.
b. Off-by-one error: A variable or parameter or constant is greater or less by 1.

Statement faults: Usually just a missing of some assignment.

Decision faults: Incorrect logical operators, can be classified as follows:

a. Operator reference fault: ' $\vee$ ' is replaced by ' $\wedge$ ', or vice versa, e.g., $x_1 \wedge x_2$ by $x_1 \vee x_2$.
b. Expression negation fault: A subformula replaced by its negation, for example, $x_1 \wedge (x_2 \vee x_3)$ implemented as $x_1 \wedge \neg (x_2 \vee x_3)$.
c. Variable negation fault: One of the conditions is replaced by its negation or negation is missed in the formula.
d. Associative Shift Fault: Incorrect implementation due to misunderstanding about operator evaluation priorities, and is caused by missing the brackets. For example, $x_1 \wedge (x_2 \vee x_3)$ if implemented as $x_1 \wedge x_2 \vee x_3$ will mean $(x_1 \wedge x_2) \vee x_3$, since ' $\wedge$ ' has higher priority as compared to ' $\vee$ '.
e. Missing variable fault: absence of a condition in the formula, e.g., $x_1 \vee x_2$ implemented as x_1.
f. Variable reference fault: A condition is replaced by another input that appears in the formula, for example, $x_1 \wedge x_2 \vee \neg x_1 \wedge x_3$ implemented as $x_1 \wedge x_2 \vee \neg x_2 \wedge x_3$.

A test case is said to kill a mutant, if it fails during execution simulation on model with that mutant. The fact of failure is ascertained using expected value oracles, which define concrete expected observable values for each test input. For formal models, it is possible to exploit so-called *maximum oracle* that defines expected values for all outputs and all internal state variables at each step of simulation (also called *weak mutation* criterion). To kill a mutant using *strong mutation* [19], the following conditions must be met [16, 20]:

- the test must reach the mutation;
- the test must infect program state by causing it to differ between the original and mutated program;
- incorrect state must propagate to program output;
- the test oracle must reveal the difference.

Strong Mutation is often referred to as traditional Mutation Testing [7] though it is computationally much more expensive than weak mutation.

The success of a test set is measured by the number of the mutants it can kill. Many researchers have conducted experiments to evaluate the effectiveness of Mutation Testing. There are evidences reported [3, 7, 8] that mutation-wise adequate test sets more easily satisfy the "all-uses" data flow coverage criteria than "all-uses" test sets satisfy mutation criteria [21]. The authors of [21] came to a conclusion that while the number of coverage items for mutation testing is much larger than that needed for statement or branch coverage, it did not require significantly more tests.

One of the major sources of computational cost in Mutation Testing is the large number of mutants to be executed against the test suite. As a result, reducing the number of generated mutants without significant loss of test effectiveness has become a popular research problem. For example, Mutant Sampling [22] approach randomly chooses a small subset of mutants from the entire set for analysis and the remaining mutants are discarded. There were many empirical studies of this approach; the results suggested that random selection of 10% of mutants is only 16% less effective than a full set of mutants in terms of mutation score [7].

3. Method description

Throughout this paper, algorithms will be presented using the model of extended finite state machine (EFSM).

Definition. An EFSM A is a tuple <C, c_0, s_0, E, V, T>, where C is a set of control flow locations, $c_0 \in C$ the initial location, E is a set of events, V is a finite set of variables with finite value domains and T is a finite set of transitions. A transition is of the form <c, g, t, u, c′>∈T, where $c \in C$ is the source location and $c' \in C$ – destination, g is a guard (a first-order predicate) over V, $t \in E$ is an event, and u is an update in the form of an assignment of variables in V to expressions over V. A state of an EFSM is a mapping from V to values, s_0 an initial state. A model state transition is of the form $s \xrightarrow{t} s'$ and is possible if there is a transition <c, g, t, u, c′>∈T where the guard g is satisfied for the valuation of s, and the result of updating s according to u is s′. A path is a proper sequence of states leading from initial state: $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots$. A state s_i is reachable, if there is a path leading from initial state s_0 to s_i, S denotes the whole set of reachable states.

Producing of auxiliary goals to reach.

Performing of the generation of mutations is not considered in this paper; it could be done using some off-the-shelf mutations generator. Our approach assumes that the set of mutations is represented by subsets MUT(t), where t is a transition name. Faults can be introduced either in pre- or post-condition in accordance with the types of mutation. We will consider mutations in pre-conditions; for this case we assume that the mutated behavior shall take a different execution branch.

Definition (necessity). A path p weakly kills a mutant m of an original precondition q if it includes a state s such that the following is satisfiable:

$$(s \wedge q \wedge \neg m) \vee (s \wedge \neg q \wedge m)$$

In the algorithm description we will only consider case $s \wedge q \wedge \neg m$ since we assume that each model's conditional statement will have both – 'if' and 'else' branches, and thus, case $s \wedge \neg q \wedge m$ will be handled for 'else'-case.

In Strong Mutation, for a given program p, a mutant m of program p is said to be killed only if mutant m gives a different output from the original program p [7, 19]. So we adopt the definition of the observability via requirement of coverage of appropriate data flow interactions sequence which are represented as alternating definitions and uses. A sequence $[\, d_{v_1}^{x_1} u_{v_2}^{x_1} d_{v_2}^{x_2} u_{v_3}^{x_1} \ldots d_{v_n}^{x_n} u_{v_{n+1}}^{x_n} \,]$ is a *data flow chain* (df-chain) if, for every $1 \le i \le n$, $(d_{v_i}^{x_i}, u_{v_{i+1}}^{x_i})$ is a du-pair [23]. Note that the use $u_{v_{i+1}}^{x_i}$ and definition $d_{v_{i+1}}^{x_{i+1}}$ occur at the same vertex for every $1 \le i \le n$. A path $v_1\pi_1v_2\pi_2\ldots v_{n+1}$ is an interaction subpath of a df-chain if, for every $1 \le i \le n$, $v_i\pi_iv_{i+1}$ is a definition-clear path from v_i to v_{i+1} with respect to x_i. A data flow chain consisting of $k-1$ du-pairs, $k \ge 2$, is a k-definition/reference interaction (k-dr interaction) in the terminology used by Ntafos [24].

Definition (observability). A path strongly kills a mutant if the mutant is weakly killed and there is: (1) some affected output or (2) some non-empty set of affected definitions and there is some path covering some data flow sequence (Ntafos *k*-dr interaction) starting from the definition point of affected variable and passing to some parameter of some observable output.

This definition of observability is optimistically inaccurate – it may report that certain mutations are observable when they are actually not. This is easily demonstrated [20] by a code fragment: if (cond) then out := 0 else out := 0. Note that in general, the problem of equivalent mutations is undecidable [7].

A control-flow location is affected by a mutant if the mutant causes difference in a conditional operator decision (and thus, difference in the immediate flow location) and the location is inside of a dominance frontier of the conditional operator. This implies that any definition and observable output which location is in the affected locations set also becomes 'affected by the mutation'. For the branch taken by mutated condition, affected definitions are previous ones w.r.t. definitions inside of the branch frontier. Let's consider examples given at fig.1a, 1b, 1c.

Let the mutated assignment in code snippet at Fig. 1a is condition '`A == 0`' at line 2 so that it becomes non-satisfiable. Observable difference with original model then will be absence of '`ok`' message.

Let the mutated condition in code snippet at Fig. 1b is cond1 at line 2 so that it becomes satisfiable, and thus, original model behavior path will go through the else-branch. The affected locations are 2–8, so the set of affected definitions is {1:A, 3:A, 6:B, 8:C}. There is a k-dr interaction [8:C, 11:Z, 13:Z] which, if reachable, allows the mutation be observed (in the original model correct output will be 1, in mutated – 0).

Let the mutated condition in code snipped at Fig. 1c is cond1 at line 2 so that it becomes non-satisfiable. There is no k-dr interaction which can drive to the observation, however, the mutation effect is observable: in original model correct output will be 0, and in mutated – 1. This example shows lack of sensitivity to the mutation effect in the proposed approach – the divergence in condition 6 is uncaught. In order to overcome this issue, in [20] special tagging procedure is maintained in order to avoid possible masking of affected variables in conditions encountered during prolongation paths.

```
1 A:=0;
2 if(A == 0)
3    …
4    print('ok');
5 else
6    …
7 print('end');
8 return;
```

a

```
1 A:=0; B:=0; C:=0;
2 if(cond1)
3    A:=1;
4 else
5    if(cond2)
6       B:=1;
7    else
8       C:=1;
9 if(cond3)
10   C:=2;
11 Z:=C;
12 if(cond4)
13   OUTPUT(Z);
14 return;
```

b

```
1 A:=0; B:=0; C:=0;
2 if(cond1)
3    A:=1;
4 else
5    B:=1;
6 if(B==1)
7    C:=1;
8 Z:=C;
9 if(cond2)
10   OUTPUT(Z);
11 return;
```

c

Fig. 1. Observability example: a – divergence in output signal; b – divergence in output parameter; c – uncaught divergence

Main algorithm. Below the main algorithm is described (see fig. 2). It takes an input sets MUT of mutated transitions. Its output is two sets – strongly killed mutants (i. e., for which observable k-dr interaction is found) and weakly killed mutants (for which only difference in some conditional operator is identified).

```
1  procedure search(MUT)
2    weakly_killed:=∅;
3    strongly_killed:=∅;
4    POSTPONED:=initial state;
5    while POSTPONED ≠ ∅
6       select s from POSTPONED;
7       if(s can contribute or can reach undiscovered mutant) then
8          for all t from transition alternatives at s
9             if sat(s ∧ t.precond) then
10               s' := pt(s,t);
11               add s' to POSTPONED;
12            else propagate_back(unsat_core(s ∧ t.precond), s);
13            for all tm ∈ MUT(t)
14               if (s ∧ t.precond ∧ ¬tm.precond) then
15                 affted_nodes := affected_set(s, t, tm);
16                 (res_observe; observe_point) = find_observation(s, affected_nodes);
17                 if res_observe = true then
18                    add (tm; observe_point) to strongly_killed;
19                    remove tm from MUT(t);
20                 else
21                    add (tm; s) to weakly_killed;
22                    propagate_back(res_observe, s);
23    return (strongly_killed, {weakly_killed except strongly_killed});
```

Fig. 2. A reachability analysis algorithm adopted for on-the-fly mutants killing

Exact state-space traversal procedure description is out of scope of this paper, we only mention that decision about state unfolding may rely on unsatisfiability cores [18], which are propagated by `propagate_back` procedure. Operator `pt` at line 10 realizes forward predicate transformation [25] from one state to another using existentional abstraction. Procedure `find_observation` (described in Section 4) finds a prolongation of the path with observable effect of the mutation or provides a feedback to the test generation process via `res_observe` which accumulates info

about failures of the search. Procedure `affected_set` find all sets of variables definitions, parameterized by event name, which values are affected by the mutant `tm` at state `s` w.r.t. original transition `t`. If the mutation is in precondition, and thus, the difference with original model is in the different decision branch, the procedure will collect all definitions (assignments) down to the dominance frontier of the difference. If the mutation is in post-condition, then the procedure will result only in one definition – the mutated assignment.

4. Ensuring observability of the mutation effect

This section describes the path prolongation procedure which finds a suffix covering some (non-predefined) data flow chain which essentially represents Ntafos *k*-dr interaction [24].

We will consider a mutation effect as observable if a prolongation of the mutated path from the divergence point will manifest a difference with the normal path in one of two ways: it will differ in some output signal or in some parameter of some output signal. For this reason we will identify a set of assignments which are affected by the detected behavior difference and try to find a sequence of actions such that at least one of such assignment via some transitive assignments will finally appear in any parameter of any output signal.

The idea of the search performance improving is to extend the path termination condition so that it can avoid unfolding of non-perspective states with respect to the sought observability. The improving is based on two modifications: for each state, the novel algorithm will store information about (1) reachable set of observable variables, and (2) affected by mutation difference variables definitions. For this purpose the algorithm needs auxiliary attributes of a state – special sets – respectively '`observable`' and '`affected`'. Note that the sets will be computed on-the-fly, and at the moment of states comparison it is unknown whether the '`observable`' set can be enlarged, but, nevertheless, the decision about path termination shall be made. In order to resolve the contradiction, the proposed algorithm has a *state refinement* option, which may resume previously terminated state and continue it's unfolding.

Thus, the state structure is extended by a tuple <`transition`, `affected`, `observed`, `idems`, `prev`> where `transition` is a name of event which adduces to the state, `affected` and `observed` stores information about prospects of the search, they will be used to prune analyzed behavior branches that will not be able to reach desired observation; set `idems` keeps track of equivalent states, it is used for terminated paths resuming and `prev` holds the previous state on the current path. The new algorithm consists of two procedures – `find_observation` (fig. 3) and `propagate_observable` (fig. 4).

```
 1 procedure find_observation(s0, affected_defs)
 2    s0.transition:=∅; s0.prev:=∅; s0.observable:=∅; s0.idems:=∅;
 3    s0.affected:= affected_defs;
 4    WAIT:=s0; VISITED:=∅;
 5    while WAIT≠∅
 6       select s from WAIT;
 7       if (∃s':s'∈VISITED ∧ restrict(s)⊆restrict(s')) then
 8          for all v∈s'.observable
 9             propagate_observable(v, p);
10          if {s'.observable ∩ vars(s.affected)} = ∅ then
11             add (s, p) to s'.idems;
12             continue;
13       add s to VISITED;
14       for all (t, s'): s --t--> s' do
15          s'.affected := s.affected;
             //Remove all re-defined variables from affected_defs
16          for all v,c:(v.c)∈defs(s') ∧ (v.c)∉affected_defs ∧ v∈vars(affected_defs)
17             remove (v.c) from s'.affected;
            //Update affected set with new defs which are affected by affected_defs
18          for all v: (v.t)∈defs(s') ∧ ∃d: d∈s.affected ∧ (v.t)∈c-uses(d)
19             add (v.t) to s'.affected;
20          if ∃v,c: (v.c)∈s'.affected ∧ v∈observable_output(t)
21             return (true; p);
22          for all v:v∈observable_output(t)
23             propagate_observable(v, p);
24          s'.observable:=∅; s'.idems:=∅;
25          s'.transition:=t; s'.prev:=s;
26          add s' to WAIT;
27 return (unsat reasons propagated upto s0, ∅);
```

Fig. 3. The algorithm for finding of observability: procedure `find_observation`

The procedure `propagate_observable` propagates detected observation of variables bottom-up to their redefinition along the current path, also adding by transitivity left-side of assignments which has observable variables at the right-side, and also adds every encountered state in `idems` for which observability has not been propagated yet into the `WAIT` set again in order to maintain completeness of the search. We will write `v.c` to denote definition of a variable `v` at location `c` (location is identified with transition name). Function `vars(ds)` will be used to denote set of variables belonging to the set of definitions `ds`.

```
28 procedure propagate_observable(v, curr_pos)
29    if v ∈ curr_pos.observable then return;
30    add v to curr_pos.observable;
31    while curr_pos.prev ≠ ∅
32       x := curr_pos.prev;
33       if v∉x.observable then
34          for all i:i∈x.idems
35             add i to WAIT;
36             x.idems := ∅;
37       for all r:r∈defs(x,curr_pos.transition)∧v∈c-uses(x,curr_pos.transition,r)
38          propagate_observable(r, x);
39       if v∈defs(x,curr_pos.transition) then return;
40       curr_pos := curr_pos.prev;
41 return;
```

Fig. 4. The algorithm for finding of observability: procedure `propagate_observable`

At line 18, `c-uses(s,t,v)` denotes a set of variables which are used for computation of variable `v` at state `s` during performing of transition `t`, i.e., all variables from the right hand side of the assignment to `v`. Predicate '`observable_output(t)`' (used at lines 20 and 22) denotes set of variables which are observable at point `t`.

Note that the current searching state is considered as non-perspective (and thus, it will be terminated) if it can not reach observation of the mutation effect: none of its '`affected`' set of variables can become observable. The condition is formulated at line 10. However, as it was mentioned before, the decision is not irrevocable: the idem-state can be refined later by the `propagate_observable` procedure at lines 9 or 23, and the terminated state will be placed to the set `WAIT` again at line 35.

In example 4 (fig. 5) the mutated condition is cond1, which holds in original model and, by condition at line 14 of the main algorithm, fails in mutated version. Therefore set of affected variables definitions is {(1,F), (3,A), (5,F)}. The observability algorithm will terminate unfolding of path p1={1,2,3,6,7,8,12,6} due to condition at lines 7 and 10, because the '`observable`' set of a state at point '6' is empty for the second time. But later, during processing of path p2={1,2,3,6,7,9,10,11} it will be refined because the `propagate_observable` procedure will add the variable 'A' (via 'B' and 'X' by transitivity) to '`observable`' set of state at point '6', forcing resuming of path p1 (because condition at line 10 is no longer hold), so that it will be re-constructed to p1={1,2,3,6,7,8,12,6,7,9,10,11}, making effect of the mutation observable via du-sequence (3,A)->(8,B)->(10,X)->(11,X).

Example 5 (Fig. 6) demonstrates avoiding of exponential growth of state-space to be traversed for the model without observable effect of mutated condition cond. While number of reachable paths is $O(2^{n+1})$, after the first path {1,2,3,4,6,…, n+4,n+6} all posterior paths {1,2,3,4,6,…,n+4,n+5}, {1,2,3,4,6,…,n+2,n+3}, …, {1,2,3,4,5,6} will be terminated at line 12 of the algorithm, despite enlarging of '`observable`' set, because the intersection of sets of affected variables {A} and observable variables $\{X_1,\ldots,X_n\}$ is empty. This allows to reduce number of visited states to O(n).

```
 1 A:=0; F:=0; i:=0;
 2 if(cond1)
 3    A:=1;
 4 else
 5    F:=1;
 6 while(i<N)
 7    if(cond2)
 8       B:=A;
 9    else
10       X:=B;
11       OUTPUT(X);
12    i++;
13 return;
```

Fig. 5. Coverage example 4

```
 1  A:=0; forall(i=0..n) Xi:=0;
 2  if(cond)
 3     A:=1;
 4  if(…)
 5     OUTPUT(X1);
 6  if(…)
 7     OUTPUT(X2)
 …
n+4 if(…)
n+5    OUTPUT(Xn);
n+6 return;
```

Fig. 6. Coverage example 5

Theorem. Let the size of the initial set of affected variables definitions is 1 and let the model has only one feasible finite k-dr interaction from s_0 to the observation point. Then the `find_observation` algorithm guaranteed to find a path which will cover the sequence in time $O(\max(|S|, |E|)\cdot|V|)$ for finite-state models.

Proof sketch.

Termination. In the path termination decision at line 7 the algorithm may consider at most |S| number of states, condition at line 10 may become false at most |V| times per state, number of transitions outgoing from a state is at most |T|. The `WAIT` set is extended at line 26, where a new state is reached, and at line 35, where idem-state is refined. Number of model transitions and states is finite, and the number of refinements, according to condition at line 33 and updates of the `observable` set at line 30 at worst is $|S|\cdot|V|$. Total time of other computations, including the `propagate_observable` procedure, is obviously no more than $O(\max(|T|, |S|)\cdot|V|)$.

Correctness. By induction on the number of iterations of the loop at lines 5–26 that have completed their work (at lines 12, 21, 26), it is sufficient to show that if the supposed k-dr interaction shall be covered, then the following alternatives only are possible:

1. the sequence is already covered (i.e., the procedure terminated at line 21),
2. the state from which the path for the k-dr interaction coverage is reachable, is in `WAIT` set,
3. the state `s` from which the path for the k-dr interaction coverage is reachable, is in `idem`-set of some state `q`$\in$ `VISITED`, but there is another state in `WAIT` set such that its analysis will eventually lead to the extension of '`observable`' set of `q`, and thus, `s` will be placed in `WAIT` again at line 35.

Note that only two termination cases are possible: at line 21, when the desired path is found, and at line 27, when `WAIT` becomes empty, meaning that no satisfying path exists. Since at the first iteration of the loop the set `WAIT` is non-empty (it will contain initial state) and, under assumption that the supposed covering path exists, it will, according to the induction proposition, remain non-empty until termination at line 21. This means that the algorithm will find the path which covers the k-dr interaction since the algorithm will eventually terminate.

Implementation notes.

1. For models with well-structured control flow graph, over-approximation of the '`observable`' set could be computed statically, e.g., before actual state space traversal. This will allow example 2 to terminate much earlier.
2. Due to memory consumption for larger models it is efficient to check mutations in iterative manner.
3. In case of faulty post-condition, the mutated definition will just be added to the initial affected set. Also, to avoid masking of the mutation effect, all atomic predicates in p-usages shall be substituted with conditional clause $\neg$(`v=ori_v`), where `ori_v` is the value of the appropriate variable `v` at current state of original model. For example, for faulty `b`, original condition (`a`$\vee$`b`) will be mutated to (`a`$\vee\neg$`b`), and thus, in order to recognize the difference with the mutated behavior, a state satisfying $\neg$`a`$\wedge$`b` shall then be reached.
4. In order to avoid duplicate search, before looking for a new observability path, check if it was already created for another mutant killing.
5. The traversal can be easily tuned to BFS or DFS strategy: for this reason, the path depth shall be taken into account in the operator `select`.

Test suite minimization.

The aim of a good test suite generation technique is killing as many mutants with fewer test cases. A test suite may exhibit substantial levels of redundancy, as each test likely kills several mutants. In general, the minimization task is NP-complete, so greedy approximation is used in practice [26]. Having information regarding the mutants each test will kill, the greedy algorithm selects the test scenario that covers the largest number of mutants not killed by the previously selected tests.

5. Related work

Model-based test generation is a topic of interest of much research [9]. Largely existing approaches use some coverage criterion, pure or with auxiliary heuristics, and try to achieve better results (performance, quality, maintainability, ability to reveal bugs etc.). Ability of detection of certain types of faults by different coverage criteria is discussed, for example, in [21].

Mutation score. The main idea of mutation based test data generation is to produce test data that can effectively kill mutants. Constraint-based test data generation is one of the automatic approaches where each condition for a mutant is turned into constraint. However, without observability, the method [16] only guarantee weak killing; the technique also suffers from some of the drawbacks associated with symbolic evaluation [7, 12]. Works [10, 11] apply genetic approach to increase mutation score: starting from some initial test suite the algorithms proposed produce new populations with higher mutation ratio. Authors of [27] emphasize influence of test suite minimization on its fault-detection ability. They came to a conclusion that test suite size is much more responsible for that rather than different coverage criteria achieved. Similarly, in [2] the authors observed a significant correlation between the effectiveness of a test suite and its size.

Observability. In [23], data flow-based testing is extended to consider control flow dependencies; this approach allows to make the effect of coverage items observable. Following this approach, in our previous work [14] all generated test paths are prolonged with respect to a pre-defined def-use-chain which completes at a statement with an observable effect of the coverage item (e. g., a statement where this item is output to the environment or written to persistent storage). Note however, that the observability method itself does not imply sufficiency condition for test output divergence: the method only focused on def-use chain, not taking into account actual computed values and possible 'masking' effect, which occurs when the value of a condition (an atomic variable or subexpression) in a Boolean decision hides the effects of other conditions. In order to overcome the masking issue, in [20] authors have proposed special tagging procedure, which introduces an auxiliary attribute – tag – for each variable corresponding to a Boolean expression or atomic value in the program, the observability of which is tracked through the execution of the a program. In order to generate tests that meet the conditions of observability, the tags tracking is annotated with trap properties so that the path condition is considered to be fulfilled if the tag is propagated to the output.

Performance. In [13], test generation is performed using a coverage criterion in the form of a set of items to be covered and introduces the notion of coverage subsumption, which allows to truncate the exploration of a path if it does not cover more items than were previously generated. The method proposed in [17] truncates exploration of a path as soon as the analysis can determine that continued execution will produce effects that have already been seen. This approach collects the set of all read-accessed (live) variables during the search and compares states only up to those variables. However, in order to identify the read-set for some state, the approach requires complete depth-first traversal of all paths after that state. In previous work we presented an algorithm that efficiently computes all-uses coverage without requiring observability [15].

6. Conclusions

High computational complexity often causes a perception that Mutation Testing is costly and impractical and remains a barrier to deployment in industry [7, 8]. Our method can be considered as a compromise approach to make it cheaper: the proposed method does not *guarantee* mutation adequacy, nevertheless significantly *increases* the mutation score, and thus, ability to reveal defects.

Unlike existing mutation-based methods, which assume construction of as many copies of initial model as many desired mutants produced, we generate a global 'super-mutated' model which includes all mutants and developed an algorithm of state-space traversal which tries to kill mutants on-the-fly using weak (just behavior divergence) and strong (requiring observable output effect) mutant recognition. During paths generation, our algorithm checks for possibility of detection of a mutant not killed yet, and, upon detection, stores the path with appropriate mark or keeps track of its unsatisfiable core otherwise.

Also the proposed procedure of finding of observability path itself can serve as an auxiliary strengthener for more thorough testing – that is, while making provision for a certain coverage criterion, the observability effect of the required item being covered rises the chances to make defects visible.

The prototype was implemented and applied as a complementary tool for automatic test suites generation aimed for branch and all-uses coverage criteria for testing of migration of legacy systems [14, 28, 29]. Iterative process turns to be very practical since each test case killing some mutant will very likely accidentally kill many others. The authors of [18] came to a similar conclusion stating that while the number of coverage items for mutation testing is much larger than that needed for statement or branch coverage, it did not require significantly more tests.

It is well known that the ability to handle the exponential growth of the search space is the most critical feature of space exploration based methods [9, 12, 13, 14, 15, 17, 28]. In this respect, our search algorithm aims to recognize redundancy of state exploration in advance. This method relies on the fact that if a state provably has no reachable observations of affected variables, then its exploration can be pruned from the search. For this reason it stores and dynamically refines knowledge about observable items reached from each state to prune the remaining exploration; an early-terminated path can later be resumed upon the refinement in order to hold completeness of the search. In many practical cases the analyzed state space potentially can be reduced exponentially and the proposed algorithm terminates much earlier, and thus, it significantly increases the maximum size of models for which the proposed analysis can be algorithmically computed.

References

1. Dssouli R., et al. Testing the control-flow, data-flow, and time aspects of communication systems: a survey. *Adv. Comput*. 2017. 107. P. 95–155.
2. Inozemtseva L., Holmes R. Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of ACM ICSE. P. 435–445 (2015). http://doi.org/10.1145/2568225.2568271
3. Chekam T., et. al. An empirical study on mutation, statement and branch coverage fault revelation that avoids unreliable clean program assumption. In: IEEE-ACM 39th International Conference on Software Engineering, 12 p. (2017). http://doi.org/10.1109/ICSE.2017.61
4. Gay G., Staats M., Whalen M., Heimdahl M. The risks of coverage-directed test case generation. IEEE Trans. Softw. Eng. 41. P. 803–819. (2015).

5. Heimdahl M., Devaraj G. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In: IEEE Computer Society, HASE. P. 178–186. (2004).
6. Morell L.J. A theory of fault-based testing. IEEE Trans. Softw. Eng. 16(8). P. 844–857. (1990). https://doi.org/10.1109/32.57623
7. Jia Y., Harman M. An Analysis and Survey of the Development of Mutation Testing. IEEE Trans. Softw. Eng. 37(5). P. 1–31. (2010).
8. Papadakis M. at al. Mutation Testing Advances: An Analysis and Survey. (2019) Advances in Computers, Vol. 112. Elsevier. P. 275 – 378. https://doi.org/10.1016/bs.adcom.2018.03.015
9. Volkov V., et al. A survey of systematic methods for code-based test data generation. Artif. Intell. 2. P. 71–85 (2017) .
10. Fraser G., Arcuri A. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with Evosuite. Empir. Softw. Eng. 20(3). P. 611–639. (2015).
11. Choi Y.M., Lim D.J. Model-Based Test Suite Generation Using Mutation Analysis for Fault Localization (2019). Appl. Sci. 9(17). P. 34–92. https://doi.org/10.3390/app9173492
12. Chekam T. at al. Killing Stubborn Mutants with Symbolic Execution. arXiv:2001.02941. 20 p. (2020).
13. Hessel A., Petterson P. A global algorithm for model-based test suite generation. Electr. Notes Theor. Comp. Sci. (2007). Vol. 190. P. 47–59.
14. Weigert T. at al. Generating Test Suites to Validate Legacy Systems. Lecture Notes in Computer Science (2019). Vol. 11753. P. 3–23. https:/doi.org/10.1007/978-3-030-30690-8_1
15. Kolchin A. A novel algorithm for attacking path explosion in model-based test generation for data flow coverage. Proc. of IEEE 1st Int. Conf. on System Analysis and Intelligent Computing, SAIC. (2018). P. 226–231. https://doi.org/10.1109/SAIC.2018.8516824
16. DeMillo Richard A. and A. Jefferson Offutt. 1991. Constraint-Based Automatic Test Data Generation. IEEE Trans. Software Eng. 17, 9 (1991). P. 900–910. https://doi.org/10.1109/32.92910
17. Boonstoppel P., Cadar C., Engler D. RWset: attacking path explosion in constraint-based test generation. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS. Vol. 4963. P. 351–366. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3 27
18. Beyer D., Dangl M. SMT-based Software Model Checking: An Experimental Comparison of Four Algorithms. VSTTE 2016: Verified Software. Theories, Tools, and Experiments. (2016). P. 181–198.
19. DeMillo R.A., Lipton R.J., and Sayward F.G. Hints on Test Data Selection: Help for the Practicing Programmer. Computer. Vol. 11, N 4. P. 34–41. (1978).
20. Meng Y., Gay G., Whalen M. Ensuring the Observability of Structural Test Obligations. IEEE Transactions on Software Engineering (Early Access). (2019). https://doi.org/10.1109/tse.2018.2869146
21. Li N., Offut J. An experimental comparison of four unit test criteria: mutation, edge-pair, all-uses and prime path coverage. In: IEEE International Conference on Software Testing, Verification and Validation. P. 220–229. (2009). http://doi.org/10.1109/ICSTW.2009.30
22. Budd T.A. Mutation Analysis of Program Test Data. PhD Thesis, Yale University, New Haven, Connecticut, 1980.
23. Hong H., Ural H. Dependence testing: extending data flow testing with control dependence. LNCS. Vol. 3502. (2005). P. 23–39. doi>10.1007/11430230_3
24. Ntafos S. On required element testing. IEEE Trans. Software Eng. Vol. 10, N 6. P. 795–803. (1984).
25. Letichevsky A.A. Algebraic Interaction Theory and Cyber-Physical Systems. *Journal of Automation and Information Sciences*. Vol. 49. P. 1–19. (2017). https://doi.org/10.1615/JAutomatInfScien.v49.i9.10
26. Tallam S., Gupta N. A concept analysis inspired greedy algorithm for test suite minimization. ACM Softw. Eng. Notes 31(1). P. 35–42. (2006). https://doi.org/10.1145/1108768.1108802
27. Namin A., Andrews J. The influence of size and coverage on test suite effectiveness. *In Proc. of Int. Symp. on Softw. Testing*. P. 57–68. (2009). http://doi.org/10.1145/1572272.1572280
28. Kolchin A., Potiyenko Weigert. Challenges for automated, model-based test scenario generation. *Communications in Computer and Information Science*. (2019). Vol. 1078. P. 182–194.
29. Guba A., et al. A method for business logic extraction from legacy COBOL code of industrial systems. In: Proceedings of the 10th International Conference on Programming UkrPROG2016, CEUR-WS. Vol. 1631. P. 17–25. (2016).

Література

1. Dssouli R., et al. Testing the control-flow, data-flow, and time aspects of communication systems: a survey. *Adv. Comput*. 2017. 107. P. 95–155.
2. Inozemtseva L., Holmes R. Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of ACM ICSE. P. 435–445 (2015). http://doi.org/10.1145/2568225.2568271
3. Chekam T., et. al. An empirical study on mutation, statement and branch coverage fault revelation that avoids unreliable clean program assumption. In: IEEE-ACM 39th International Conference on Software Engineering, 12 p. (2017). http://doi.org/10.1109/ICSE.2017.61
4. Gay G., Staats M., Whalen M., Heimdahl M. The risks of coverage-directed test case generation. IEEE Trans. Softw. Eng. 41. P. 803–819. (2015).
5. Heimdahl M., Devaraj G. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In: IEEE Computer Society, HASE. P. 178–186. (2004).
6. Morell L.J. A theory of fault-based testing. IEEE Trans. Softw. Eng. 16(8). P. 844–857. (1990). https://doi.org/10.1109/32.57623
7. Jia Y., Harman M. An Analysis and Survey of the Development of Mutation Testing. IEEE Trans. Softw. Eng. 37(5). P. 1–31. (2010).
8. Papadakis M. at al. Mutation Testing Advances: An Analysis and Survey. (2019) Advances in Computers, Vol. 112. Elsevier. P. 275 – 378. https://doi.org/10.1016/bs.adcom.2018.03.015
9. Volkov V., et al. A survey of systematic methods for code-based test data generation. Artif. Intell. 2. P. 71–85 (2017) .
10. Fraser G., Arcuri A. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with Evosuite. Empir. Softw. Eng. 20(3). P. 611–639. (2015).
11. Choi Y.M., Lim D.J. Model-Based Test Suite Generation Using Mutation Analysis for Fault Localization (2019). Appl. Sci. 9(17). P. 34–92. https://doi.org/10.3390/app9173492
12. Chekam T. at al. Killing Stubborn Mutants with Symbolic Execution. arXiv:2001.02941. 20 p. (2020).
13. Hessel A., Petterson P. A global algorithm for model-based test suite generation. Electr. Notes Theor. Comp. Sci. (2007). Vol. 190. P. 47–59.
14. Weigert T. at al. Generating Test Suites to Validate Legacy Systems. Lecture Notes in Computer Science (2019). Vol. 11753. P. 3–23. https:/doi.org/10.1007/978-3-030-30690-8_1
15. Kolchin A. A novel algorithm for attacking path explosion in model-based test generation for data flow coverage. Proc. of IEEE 1st Int. Conf. on System Analysis and Intelligent Computing, SAIC. (2018). P. 226–231. https://doi.org/10.1109/SAIC.2018.8516824
16. DeMillo Richard A. and A. Jefferson Offutt. 1991. Constraint-Based Automatic Test Data Generation. IEEE Trans. Software Eng. 17, 9 (1991). P. 900–910. https://doi.org/10.1109/32.92910

17. Boonstoppel P., Cadar C., Engler D. RWset: attacking path explosion in constraint-based test generation. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS. Vol. 4963. P. 351–366. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3 27
18. Beyer D., Dangl M. SMT-based Software Model Checking: An Experimental Comparison of Four Algorithms. VSTTE 2016: Verified Software. Theories, Tools, and Experiments. (2016). P. 181–198.
19. DeMillo R.A., Lipton R.J., and Sayward F.G. Hints on Test Data Selection: Help for the Practicing Programmer. Computer. Vol. 11, N 4. P. 34–41. (1978).
20. Meng Y., Gay G., Whalen M. Ensuring the Observability of Structural Test Obligations. IEEE Transactions on Software Engineering (Early Access). (2019). https://doi.org/10.1109/tse.2018.2869146
21. Li N., Offut J. An experimental comparison of four unit test criteria: mutation, edge-pair, all-uses and prime path coverage. In: IEEE International Conference on Software Testing, Verification and Validation. P. 220–229. (2009). http://doi.org/10.1109/ICSTW.2009.30
22. Budd T.A. Mutation Analysis of Program Test Data. PhD Thesis, Yale University, New Haven, Connecticut, 1980.
23. Hong H., Ural H. Dependence testing: extending data flow testing with control dependence. LNCS. Vol. 3502. (2005). P. 23–39. doi>10.1007/11430230_3
24. Ntafos S. On required element testing. IEEE Trans. Software Eng. Vol. 10, N 6. P. 795–803. (1984).
25. Letichevsky A.A. Algebraic Interaction Theory and Cyber-Physical Systems. *Journal of Automation and Information Sciences*. Vol. 49. P. 1–19. (2017). https://doi.org/10.1615/JAutomatInfScien.v49.i9.10
26. Tallam S., Gupta N. A concept analysis inspired greedy algorithm for test suite minimization. ACM Softw. Eng. Notes 31(1). P. 35–42. (2006). https://doi.org/10.1145/1108768.1108802
27. Namin A., Andrews J. The influence of size and coverage on test suite effectiveness. In Proc. of Int. Symp. on Softw. Testing. P. 57–68. (2009). http://doi.org/10.1145/1572272.1572280
28. Kolchin A., Potiyenko, Weigert. Challenges for automated, model-based test scenario generation. Communications in Computer and Information Science. (2019). Vol. 1078. P. 182–194.
29. Guba A., et al. A method for business logic extraction from legacy COBOL code of industrial systems. In: Proceedings of the 10th International Conference on Programming UkrPROG2016, CEUR-WS. Vol. 1631. P. 17–25. (2016).

Received 02.03.2020

About the authors:

Kolchin Alexander,
PhD, senior scientist.
Publications in Ukrainian journals – 32.
Publications in foreign journals – 14.
http://orcid.org/0000-0001-7809-536X,

Potiyenko Stepan,
PhD, senior scientist.
Publications in Ukrainian journals – 20.
Publications in foreign journals – 8.

Weigert Thomas,
PhD, professor.
Publications in Ukrainian journals – 2.
Publications in foreign journals – 59.

Authors' place of work:

V.M. Glushkov Institute of Cybernetics of National Academy of Sciences of Ukraine,
Akademician Glushkov Avenue, 40, Kyiv, Ukraine, 03187,
E-mail: kolchin_av@yahoo.com.

Uniquesoft LLC, Palatine, IL, USA
E-mail: thomas.weigert@uniquesoft.com