UDC 004.414.23+004.415.53+004.832.23+ 004.4'233

INTERACTIVE METHOD FOR CUMULATIVE ANALYSIS OF SOFTWARE FORMAL MODELS BEHAVIOR

A. Kolchin

The aim of the proposed method is to simplify and improve the process of models debugging and to increase efficiency of model-based test cases generation. Unlike existing methods of models behavior analysis, which produce as a result only one, usually first-found, path per specified property (which is an evidence of test goal reachability or explanation of some inconsistency during debugging process), the proposed method generates a projection of *all* satisfiable paths, which provokes exposure of *undesired behavior*. For test cases generation, the feature plays a role of interactive path constructor, which prompts all satisfiable behavior alternatives, so user can find a desired path by iteratively specifying points-of-interest. Appropriate novel algorithm for efficient searching is presented.
Key words: testing, debugging, model checking.

Мета методу – спростити та удосконалити процес налагодження моделі та підвищити ефективність генерації тестових сценаріїв. На відзнаку від існуючих методів аналізу поведінки, які у результаті роботи для кожної властивості, що перевіряється, породжують тільки один, часто перший виявлений шлях (у підтвердження досяжності цілі тесту або для пояснення невідповідності при налагодженні), запропонований метод породжує проекцію *усіх* шляхів, що задовольняють заданої властивості, таким чином сприяючи виявленню *небажаного поведінки*. Для генерації тестових сценаріїв, метод служить інтерактивним конструктором трас, який показує всі можливі альтернативи поведінки, так що користувач може знаходити бажаний шлях ітеративна задаючи точки зацікавленості. Запропоновано відповідний новий алгоритм для ефективного пошуку.
Ключові слова: тестування, налагодження, перевірка моделі.

Цель метода – упростить и усовершенствовать процесс отладки модели и повысить эффективность генерации тестовых сценариев. В отличие от существующих методов анализа поведения, которые в результате работы для каждого проверяемого свойства порождают только один, часто первый обнаруженный путь (в подтверждение достижимости цели теста или для объяснения несоответствия при отладке), предложенный метод порождает проекцию *всех* путей, удовлетворяющих заданному свойству, таким образом способствуя выявлению *нежелательного поведения*. Для генерации тестовых сценариев, метод служит интерактивным конструктором трасс, который показывает все возможные альтернативы поведения, так что пользователь может находить желаемый путь итеративно задавая интересующие точки. Предложен соответствующий новый алгоритм для эффективного поиска.
Ключевые слова: тестирование, отладка, проверка модели.

Introduction

Formal models debugging problem. Formal models are actively used in a model-based design flow software development process. The quality of a model is crucial since it plays a role of requirements specification for future implementation. However, creating a formal model is an error-prone process and at the same time debugging is difficult and labor-consuming. Existing tools propose facilities for simulation, plain step-by-step path exploration [1, 2], and for automatic model analysis [3]. The latter essentially performs reachability checking to generate and explore the state space of a model in order to find a path that follows a given criterion. Such path can be interpreted as a test case or as an explanation of some problem during debugging. For example, approach [4] allows a designer to ask “show me a run that puts control at this point with x <= 0”. Accordingly to this request, a temporal logic formula will be constructed, and the witness (or counterexample) path will be produced [3]. In [5] authors propose a method for unrealizability and unsoundness checking of specifications. But in all cases, only one path (as a rule, first-found by searching algorithm) will be produced as a result. The single path, while serves as a reliable evidence of reachability, is unlikely to be sufficient: in order to understand the model one needs somehow to see the overall picture of its behavior. Moreover, it is well-known that it is easy to check desired behavior, but hard to check absence of undesired behavior. For example, let A blocks B, but the property to be checked is that C is reachable after A. The goal path may exist, and test can successfully pass, B may still be reachable in another way, but the design has no requirement that A shall disable B, and this problem is at risk to be undisclosed: while the former property (A $\sim>$ C) is specified, the latter one (A does not block B) is just supposed. In reality, a lot of desired properties remain unspecified explicitly, and thus, the properties are not verified and not tested.

Test cases generation problem. The industry moves toward model-based development, and automated test generation from the model is often considered as a form of requirements-based testing. 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. However, there is some evidence [3] that simply using the coverage provision as a target for automated test generation may be a flawed strategy: coverage metrics are intended to measure the thoroughness of human-generated tests, and do not necessarily lead to good test sets when used in an inverted role as a specification for the tests required. For example, in [6], authors make 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, and generally performed even worse than random testing. Another drawback of the automatic approach is that the

test case generated is not necessarily a good exercise regarding the verified property: a counterexample path is an artifact of search strategy, so it may terminate at the middle of some interesting behavior, may include a lot of redundancy, cause-effect relations are obscure and intricate, and moreover, the path might even not contain a state where antecedent of required property becomes true. Poorly designed tests are known to have a negative impact on test maintenance [7].

Proposed solution. Unlike existing methods of models behavior analysis, which produce as a result only one witness or counter-example path per specified property (or required coverage item), the proposed method provides analysis of behavior in a cumulative way: it generates aggregate information about *all* satisfiable (with respect to specified request) paths. This distinctive feature plays a role of interactive path constructor, which prompts all satisfiable behavior alternatives, so user can find a desired path by iteratively specifying points-of-interest and observe updates of the prompting on-the-fly. Visualization of the cumulative coverage assists with desired test case generation and simplifies identification of *undesired* behavior among the alternatives.

The paper describes properties of the cumulative analysis and appropriate efficient searching algorithm.

The cumulative analysis approach

Behavior of a model can be considered as a whole set of paths outgoing from initial state. Often the set growth up exponentially with the model size and in general case it could be infinite due to possible loops. User-defined property (request or test goal) considered in the proposed method stipulates a path property in form of a set of control flow locations, optionally ordered and extended with conditions over variables and restrictions on number of location visits (for example, zero visits means ‘do not consider paths passing through this point’). A path satisfies request if it includes all of its check-points appropriately. The idea of the described method is to provide a designer with information about set of all admissible paths. The naïve solution – generation of all paths – is obviously unrealizable task, and, even if the set is finite, it is not observable due to it's huge size. The described method proposes a compromise: it proceeds from the assumption that the main interesting events are represented with different structure items in a model design, and thus, it will be informative for a designer to observe a *projection* of a path on the design's structure (flowchart). Consequently, the proposed method strongly relies on a control flow graph of a model under analysis: it is efficient only for models with structured finite flow, in which interesting events and decisions are distributed along individual branches. The produced paths are represented then as a projection on the graph. The main value-added property of the method is an ability to efficiently generate a projection of *all satisfiable paths*, so user may observe them *all at once* (by highlighting of appropriate flow graph elements in a visualization panel). It makes feasible a complex cause-effect analysis of model behavior during debugging process and simplifies undesired behavior identification; for test cases generation, the method allows a designer to find a desired path by iteratively specifying points-of-interest, while projection of admissible paths will be generated and updated on-the-fly. The proposed method develops [8], where it was used as a directed search strategy [9], and later it became [10] a test goal specification.

Examples. Let’s consider a model with two attributes ‘x’ and ‘y’, transitions are presented at the table and control flow in fig. 1–3. Set of possible paths is the following: {init,a,c,e,g}, {init,a,c,e,h}, {init,a,c,f,h}, {init,a,c,f,i}, {init,b,d,e,g}, {init,b,d,f,i}. Initially (when coverage criteria are not specified yet), all transitions are reachable, and thus, highlighted.

Table. Transitions for model 1

Transition	Precondition	Postcondition
a	-	x := 1
b	-	x := 0
c	x=1	-
d	x=0	-
e	-	y := 1
f	-	y := 0
g	y=1	-
h	x=1	-
i	y=0	-

Example in fig.1 shows a highlighting which corresponds to user-defined property ‘g’. Branch is depicted as dashed line if there is no feasible path from the startpoint leading through the branch and ‘g’ simultaneously. In the example, there is only two satisfiable paths: {init,a,c,e,g} and {init,b,d,e,g}, so dashed lines marks ‘f’, ‘h’ and ‘i’.

Note that property ‘f’ and ‘g’ will result in empty coverage since ‘g’ has a guard ‘y=0’, while post-condition of ‘f’ is ‘y:=1’.

Example in fig. 2 corresponds to property ‘not a’. Lines through ‘c’ and ‘h’ become dashed, meaning that path leading to them is possible only after ‘a’. Example in fig.3 corresponds to property ‘b’ and ‘f’. There is only one possible path which can satisfy this request – {init, b, d, f, i}.

The main property of the cumulative analysis is the following: a control flow element (branch or statement) will be highlighted if and only if it is a part of some path which satisfies given request. This result could be achieved using existing search machinery provided out-of-the-box by model checkers, for example, the global algorithm with partial coverage items [2]. For this purpose a set of coverage items shall consist of $\{\bigcup_{c\in C} \mathrm{cov}+c\}$, where cov is the coverage request and C is a set of all control flow vertices. However, such approach has efficiency shortcomings: the specified request will include many infeasible coverage items; the algorithm will be forced to make redundant visits of model states. To make the cumulative analysis feasible, its searching algorithm must have an ability for early recognition of the redundancy of states exploration and to provide more efficient analysis of the state of space.

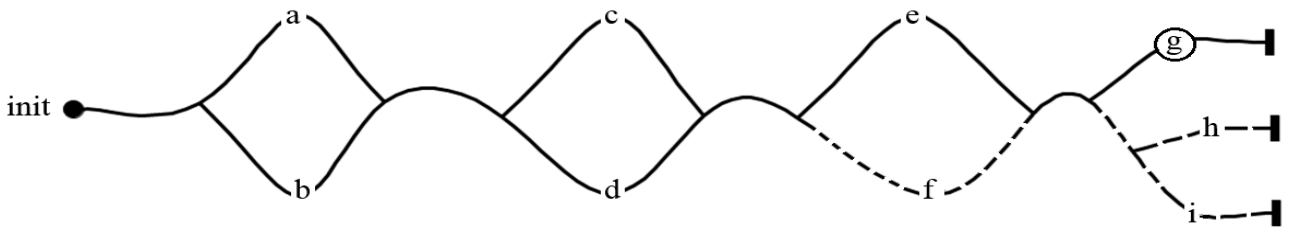

Fig. 1. Coverage example 1: required criterion is ‘g’

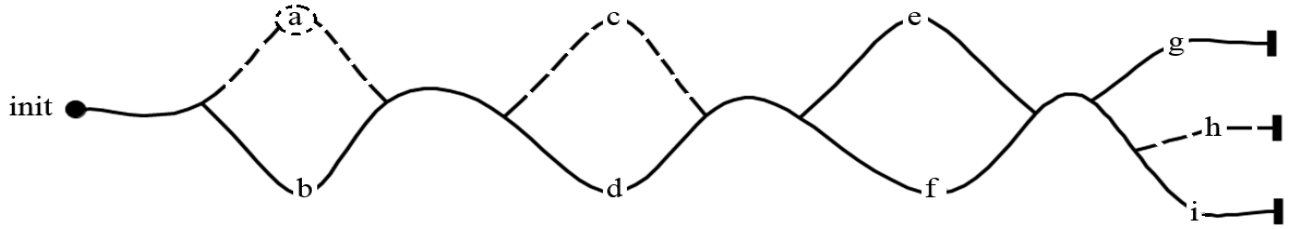

Fig. 2. Coverage example 2: required criterion is ‘not a’

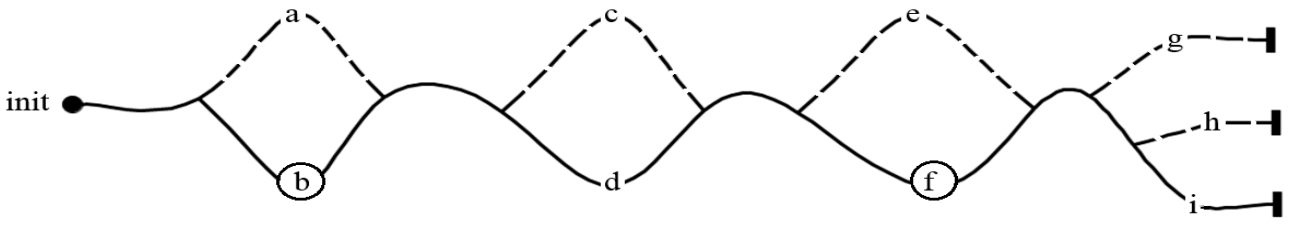

Fig. 3. Coverage example 3: required criterion is ‘b’ and ‘f’

Searching algorithm

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, 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 (model) state of an EFSM is a tuple <cfg, var> where $cfg \in C$ and var is a mapping from V to values. The initial state is <c_0, v_0> where v_0 is the initial mapping. A model state transition is of the form <c, v> $\xrightarrow{t}$ <c′, v′> and is possible if there is a transition <c, g, t, u, c′>∈ T where the guard g is satisfied for the valuation of v, and the result of updating v according to u is v′. A path is a proper sequence of states leading from initial state: <c_0,v_0> $\xrightarrow{t_0}$ <c_1,v_1> $\xrightarrow{t_1}$ …. A state <c_i,v_i> is reachable, if there is a path leading from initial state <c_0,v_0> to <c_i,v_i>, S denotes the whole set of reachable (model) states, and P – whole set of reachable paths.

In the algorithm representation, for the description simplicity reasons, the user-defined property is considered as just a set of control flow locations.

Definition. A path p satisfies property R if it includes all its elements, i.e., the following holds:

$$c \in R \Rightarrow \exists s: s \in p \wedge s.cfg = c \quad \text{(path satisfiability condition)}$$

The problem of state space traversal for reachability checking of required coverage criteria is a research topic of increasing importance, see e.g., [1–4, 7–12]. As a rule, existing algorithms are designed for searching of one path per each required coverage criterion [2, 4, 10]. In opposite, the proposed cumulative behavior analysis requires searching of all paths (indeed, their projection), which satisfy given coverage criterion. A brute-force attempt to reuse and adopt algorithms developed for systems modeled as EFSM for the cumulative behavior analysis needs is shown in fig. 4.

```
(01)procedure search(A, R)
(02)begin
(03)  Q:=∅;
(04)  for all q∈C do
(05)       VISITED:=∅; WAIT:={(A.initial, {c0 if c0∈{R∪q} or ∅ otherwise},
A.initial)};
(06)    while WAIT≠∅ ∧ q∉Q do
(07)      select (s, r, p) from WAIT;
(08)      if¬∃(si,ri,pi):(si,ri,pi)∈VISITED ∧ si.cfg=s.cfg ∧ si.var=s.var ∧ r=ri
then do
(09)        for all (t, s'): s --t--> s' do
(10)          if s'.cfg∈{R∪q} then r':= r∪s'.cfg; else r':= r;
(11)          if r'={R∪q} then
(12)            for all x: (x ∈ {p∪s'}) Q := Q∪x.cfg;
(13)          add (s', r', p∪s') to WAIT;
(14)        od
(15)      od
(16)    od
(17)    add (s, r, p) to VISITED;
(18)  od
(19)  return Q;
(20)end  □
```

Fig. 4. A reachability analysis algorithm adopted for the cumulative behavior coverage

In essence, the described algorithm is an ordinary reachability analysis algorithm, which computes a set of control flow elements (statements) with respect to the path satisfiability condition. The search is placed into a loop at the lines (04)–(18) which enumerates all control flow locations of a model. In the loop, it uses two data structures `WAIT` and `VISITED` to hold combined states waiting to be examined and states already examined respectively. The combined states are represented in the form (`s`, `r`, `p`), where `s` is a current model state, `r` – set of required coverage items which have already been covered along the path, and `p` – the current path (for description simplicity it is handled as a set of states). Initially `VISITED` is empty and `WAIT` holds the initial combined state (s_0, c_0, s_0). The lines (07) – (17) are repeated until `WAIT` is empty or chosen control flow location becomes covered. At (07) a combined state is taken from `WAIT`, then at the line (08) it is compared versus passed earlier states, and, if the state is a new one, then lines (09) – (14) generate its successors. Resulting set `Q` is replenished at line (12) if current path satisfies the required coverage. After the state successors are generated, in order to avoid duplicate examination, it is placed to `VISITED` at (17).

A well-known problem with algorithms like the one described is the time consumed to explore the state space, and the space required to represent `WAIT` and `VISITED`. State space explored by the algorithm in fig. 4 has size defined by the number of model states |S| in product with the number of possible coverage sets $2^{|R|}$ and with the number of control flow locations |C|. Obvious optimization is to check inclusion $r \subseteq r_i$ rather than equality $r=r_i$ during states comparison, but the performance remains infeasible, and thus, the algorithm is impractical for the proposed approach.

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 coverage. The improving is based on two modifications: for each state, the novel cumulative reachability searching algorithm will store information about reachable sets of (1) partial items of the required coverage, and (2) reached but non-covered control flow items. For this purpose the algorithm needs auxiliary attributes of a state – special sets 'reached' and 'wanted' to store information about reachable partial coverage items and non-covered control flow locations respectively. Note that the sets will be computed on-the-fly, and at the moment of states comparison it is unknown whether the sets 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 to a tuple <`cfg`, `var`, `reached`, `wanted`, `idems`> where `reached` and `wanted` stores information

about prospects of the search, it is used to prune analyzed behavior branches that will not be able to contribute to the coverage; set `idems` keeps track of equivalent states, it is used for terminated paths resuming. Also the external loop for control flow locations enumeration is removed; their coverage is now controlled inside of the main loop. The new algorithm consists of two procedures – `search` (fig. 5) and `propagate` (fig. 6).

```
(01)procedure search(A, R)
(02)begin
(02)  A.initial.wanted←∅; A.initial.reachable←∅; A.initial.idems←∅;
(04)    WAIT:={(A.initial, {c0 if c0∈R or ∅ otherwise}, A.initial.cfg,
A.initial)};
(05)  VISITED:=∅; Q:=∅;
(06)  while WAIT≠∅ do
(07)    select (s, r, q, p) from WAIT;
(08)      if ∃(si,ri,qi,pi):(si,ri,qi,pi)∈VISITED ∧ si.cfg=s.cfg ∧ si.var=s.var
then do
(09)      s.reached := si.reached; s.wanted := si.wanted;
(10)      for all c: (c∈si.reached) propagate(c, p, reach_mrk);
(11)      for all c: (c∈{si.wanted\Q}) propagate(c, p, want_mrk);
(12)      if((r⊆ri ∧ {q\Q}⊆qi) ∨
(13)         {r∪si.reached}⊃R ∨
(14)         {si.wanted∪q}⊆Q)then do
(15)        add (s, r, q, p) to si.idems;
(16)        continue;
(17)      od
(18)    od
(19)    add (s, r, q, p) to VISITED;
(20)    if s.cfg∉Q then propagate(s.cfg, p, want_mrk);
(21)    for all (t, s'): s --t--> s' do
(22)      s'.reached:=∅; s'.wanted:=∅; s'.idems:=∅;
(23)      if s'.cfg∈R then
(24)        do r':= r∪s'.cfg; propagate(s'.cfg, p, reach_mrk); od else r':= r;
(25)      q':= q∪s'.cfg;
(26)      if r'=R then
(27)        for all x: (x ∈ {p∪s'}) Q := Q∪x.cfg;
(28)      add (s', r', q', p∪s') to WAIT;
(29)    od
(30)  od
(31)  return Q;
(32)end  □
```

Fig. 5. The cumulative search algorithm: procedure `search`

```
(33)procedure propagate(c, p, flag)
(34)begin
(35)  for all x: x∈p do
(36)    if flag = want_mrk ∧ c∉x.wanted ∨ flag = reach_mrk ∧ c∉x.reached then
do
(37)      for all v:v∈x.idems  add v to WAIT;
(38)      x.idems:=∅;
(39)    od
(40)    if flag = reach_mrk then x.reached := x.reached∪c;
(41)    if flag = want_mrk then x.wanted := x.wanted∪c;
(42)  od
(43)end
```

Fig. 6. The cumulative search algorithm: procedure `propagate`

The main change is at the extension of the combined states comparison decision: the current searching state is now considered as non-perspective (and thus, it will be terminated) if it can not contribute to the sought coverage because (1) it can not reach the required items or (2) everything it can cover is already covered. The extension is formulated at the lines (13) and (14). However, as it was mentioned before, the decision is not irrevocable: the idem-

state can be refined later by the `propagate` procedure at the lines (40) or (41), and the terminated state will be placed to the set `WAIT` again at the line (37).

Let's consider an example of a model and a progress of the searching algorithm. The model with appropriate control flow graph and cumulative coverage highlighting is shown in fig. 7. Points-of-interest, which form the coverage criterion, are 'a' and 'e' (encircled); in this and subsequent examples there are no variables and the guards are trivial.

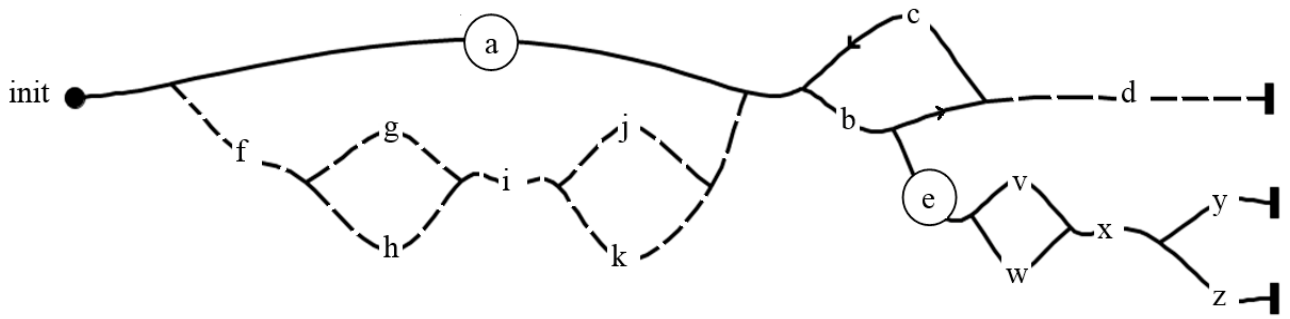

Fig. 7. Coverage example 4

In this example, path {init,a,b,c,b} will be terminated by condition at the line (13), because the 'reach' set of a state at the point 'b' is empty, but during processing of the path {init,a,b,e} it will be refined because the `propagate` procedure will add the element 'e' to the 'reach' set, and then path {init,a,b,c,b,e} will be constructed and the element 'c' will be added to `Q`. Paths {init,f,g,i,j,b}, {init,f,g,i,k,b}, {init,f,h,i} will be terminated by condition at the line (13), because appropriate 'reach' sets will miss 'a'; paths {init,a,b,e,w,x} and {init,a,b,c,b,e} by condition at the line (12) because `r`-sets are equal and the set {`q\Q`} is empty due to the earlier paths {init,a,b,e,v,x,y} and {init,a,b,e,v,x,z}.

The next example (fig. 8) shows state space reducing which is inspired by the condition at the line (14). Here the required coverage criterion is `R`={f}. During traversal, after exploring paths {init,a,b,d,f,e} and {init,a,c,d,f,e}, the subsequent path {init,f,a} will be terminated is spite of the conditions $r \subseteq r_i$ and $r \supset R$ do not hold at the state 'a' (r_i={∅}, while current `r`={f}), because the earlier paths leave nothing to cover ('wanted' set of the state a_i is empty), and thus, the condition $\{a_i.wanted \cup q\} \subseteq Q$ at the line (14) holds.

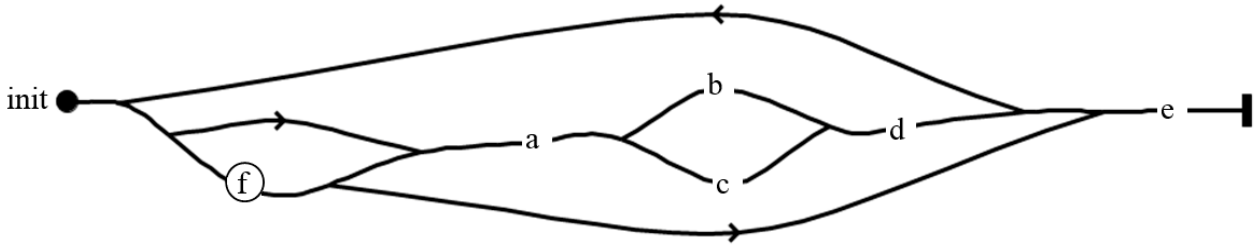

Fig. 8. Coverage example 5

Theorem. The cumulative search algorithm has the following main properties:

1. Termination. Its asymptotic time is $O(\max(|T|, |S|) \cdot |C| \cdot 2^{|R|})$.
2. Soundness. The set `Q` will consist only of control flow locations, which are on satisfiable paths:

$$c \in Q \Rightarrow \exists p, s: p \in P \wedge s \in p \wedge s.cfg = c \wedge (r \in R \Rightarrow \exists x: x.cfg = r \wedge x \in p)$$

3. Completeness. After completion, the set `Q` will include all control flow locations, which are on satisfiable paths:

$$\exists p, s, c: p \in P \wedge s \in p \wedge s.cfg = c \wedge (r \in R \Rightarrow \exists x: x.cfg = r \wedge x \in p) \Rightarrow c \in Q$$

Proof sketch.

Statement 1. In order to prove termination, it is necessary to show that set `WAIT` should eventually become empty during evaluation of the procedure `search`. The set consists of combined states presented as `(s,r,q,p)`; in the path termination decision at the lines (08) and (12) possible sets of compared elements $s \in S$, $r \in 2^{|R|}$ and $q \in C$ are

finite and path p is ignored at all, that is why the WAIT set will not contain similar states which differs by p only. The WAIT set is extended at the lines (28), where a new combined state is reached, and (37), where idem-state is refined. Number of model transitions and states is finite, and the number of refinements, according to condition at the line (36) and updates of the sets at (40) and (41), at worst is $|S|\cdot|C|$. Summary time of other computations, including the propagate procedure, is obviously no more than $O(\max(|T|, |S|)\cdot|C|\cdot 2^{|R|})$.

Statement 2. The control flow locations are added to the resulting set only at the line (27), according to the lines (23), (24) and to the condition at (26), only from satisfiable paths.

Statement 3. By induction on the number of iterations of the loop at the lines (6)–(30) that have completed their work (at the lines (16), (30)), it is sufficient to show that if some supposed flow element X shall be covered on the path reachable from the current state (selected from WAIT at the line (07)), then the following alternatives only are possible:

1) X is already covered (i.e., $X\in Q$),

2) there exist a sequence $\{p,s_w,t_s\}$, where p is a current path leading to s_w, $s_w\in$ WAIT, t_s is a finite path leading from s_w to a state, where all required coverage items become covered, i.e., the whole path becomes satisfiable, and there is state s_x, such that s_x.cfg=X and $s_x\in t_s$ or $s_x\in p$ or $s_x= s_w$.

3) X is on a path, which is reachable from the current state, but it was terminated by condition at the line (12),

4) current path p is terminated at the state s by condition at the line (13), and there exist a finite sequence $\{p,s_1,s^i_1,t_1,\ldots,s_n,s^i_n,t_n,s_w,t_w,s_c\}$, where s_k is a terminated state, s^i_k – its corresponding idem-state (i.e. $s_k\in s^i_k$.idems), t_k – finite paths leading from s^i_k to s_{k+1}, $s_w\in$ WAIT, s_1=s, $n\geq 1$, and there is a state s_c which is reachable from s_w via finite path t_w, such that s_c.cfg$\in$R$\wedge s_c$.cfg$\notin s^i_1$.reach.

5) current path p is terminated at the state s by condition at the line (14), and there exist a finite sequence $\{p,s_1,s^i_1,t_1,\ldots,s_n,s^i_n,t_n,s_w,t_w,s_c\}$, where s_k is a terminated state, s^i_k – its corresponding idem-state (i.e. $s_k\in s^i_k$.idems), t_k – finite paths leading from s^i_k to s_{k+1}, $s_w\in$ WAIT, s_1=s, $n\geq 1$, and there is a state s_c which is reachable from s_w via finite path t_w, such that s_c.cfg$\notin$Q$\wedge\ s_c$.cfg$\notin s^i_1$.wanted.

Really, each case can lead to the desired goal, i.e., to $X\in Q$:

Case 1. Trivial.

Case 2. Since $s_w\in$ WAIT, its immediate successors, according to the lines (06), (07) and (21) will be eventually generated and placed to the set WAIT at the line (28) or, in case $t_s=\varnothing$, accordingly to the lines (23), (24), (26), the X will be added to the resulting set Q at the line (27).

Case 3. The condition of the line (12) supposes existence of another state, which has the same or even larger set of reachable paths, consequently, path termination by this condition does not affect coverage of the element.

Case 4. Since the supposed state $s_w\in$ WAIT, it will be eventually selected at the line (07) and processed by the main loop of the search procedure. Here only the following cases are possible:

a) $t_w\neq\varnothing$. In this case line (28) will add all immediate successors of the s_w to the WAIT, and the assumed by the case 4 path will remain in the form $\{p,s_1,s^i_1,t_1,\ldots,s_n,s^i_n,t_n',s_w',t_w',s_c\}$, where $t_n'=\{t_n,s_w\}$ and $t_w=\{s_w, t_w'\}$.

b) $t_w=\varnothing$. Then there exits an immediate successor s_c, and execution of the line (24) will lead to propagation of the element s_c.cfg and to consequent refinement of the state s^i_n, and thus, state s_n, regarding to the line (40), will be added to the WAIT set, and the path assumed by the case 4 will take the form $\{p,s_1,s^i_1,t_1,\ldots,s_{n-1},s^i_{n-1},t_{n-1},s_w',s_c'\}$ for $n\geq 2$, otherwise (for n=1) the case under consideration will lead to the case 2, i.e., s_1 is placed to the WAIT again and existence of $\{p,s_1,t_s\}$ remains.

Case 5. Similar to case 4, the case can lead to the case 2. Proof omitted.

The induction shows (due to the space limitation, the proof is omitted) that the listed cases will remain the only possible. Since at the first iteration of the loop the set WAIT will contain initial state, and the supposed element X is reachable, and after the last iteration of the loop the set WAIT will be empty (and thus, the only possible case remained is 1), then at the end of the algorithm work the X will be in Q.

□

The new searching algorithm summary. The proposed algorithm fulfills a projection of all satisfiable paths on the control flow graph. The state space traversal is improved: a state will not be explored until it cannot increase the sought coverage, however, the exploration of a postponed state may be resumed due to the state refinement option. This

feature allows to avoid unnecessary exploration and to speed up termination, in many practical cases the state space reducing is exponential. Let's consider a demonstrable example (see fig. 9). Let the required coverage items set R is $\{w,a_1,\ldots,a_n\}$. While the asymptotic complexity is $O(|S|\cdot|C|\cdot 2^{|R|})=O(n^2\cdot 2^n)$, the algorithm will terminate in $O(n^2)$ steps: the 'reach' sets of examined states will not contain coverage element 'w', and thus (by condition at the line (13)), paths $\{init,a_1,\ldots,a_{n-2},b_{n-1},c_{n-1}\}$, $\{init,a_1,\ldots,a_{n-3},b_{n-2},c_{n-2}\}$, …, $\{init,b_1,c_1\}$ will be terminated.

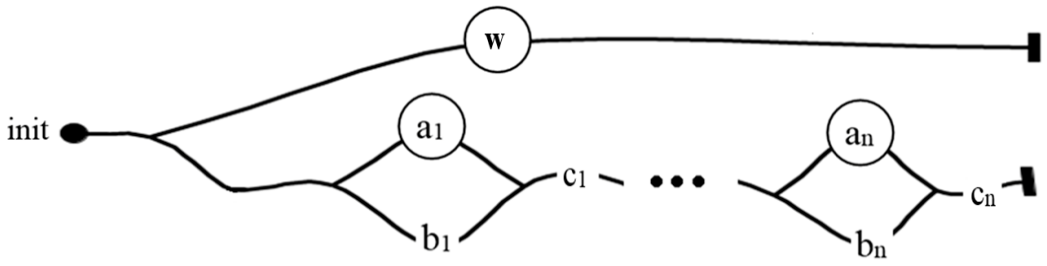

Fig. 9. Coverage example 6

Note that number of steps to termination remains $O(n^2)$ even without w in R, i.e., for the case $R=\{a_1,\ldots,a_n\}$, because paths $\{init,a_1,\ldots,a_{n-2},b_{n-1},c_{n-1}\}$, $\{init,a_1,\ldots,a_{n-3},b_{n-2},c_{n-2}\}$, …, $\{init,b_1,c_1\}$ will be terminated due to lack of a_{n-1}, a_{n-2}, …,a_1 in the 'reach' sets of corresponding states c_{n-1}, c_{n-2}, …,c_1, i.e., $a_k \notin \{r_k \cup c_k.\texttt{reached}\}$.

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`.

The algorithm can be used for test cases generation since the combined state structure already includes the needed path; also it is easy to tune the search for branch coverage criteria – the specified coverage items should be defined as pairs of points instead of single statements.

Implementation notes.

1. For models with well-structured control flow graph, organization of the '`wanted`' and '`reached`' sets could be computed statically, e.g., before actual state space traversal.

2. In order to avoid duplication of identical '`vars`', '`wanted`' and '`reached`' sets in different combined states structures appropriate pointers can be used.

3. It is useful to distinguish elements of set `Q`, which lies on slices (backward/forward) w.r.t. given property R, such technique significantly reduces model size to be observed.

4. The '`wanted`' set can be replaced with a Boolean flag, which will only indicate the presence of 'something uncovered' instead of storing a complete set of individual wanted items. This replacement is a kind of classic time/memory tradeoff – the memory saving may lead to redundant searching time.

5. In order to reduce the number of visited checking procedure calls, prioritize refined states lower than new ones during the selection from `WAIT`.

6. The described method is compatible with the dynamic abstraction method [11] which reduces a state-space of the model behavior analysis using relaxed checking of the states equivalence.

7. Experiments with the implementation of the algorithm demonstrate that for a model, which contains ~1000 transitions, the cumulative coverage computation time is less than one second. The high performance plays crucial role for the interactive operation and makes the proposed approach usage really practical.

Conclusions

The proposed interactive strategy suggests a change of the human role in test cases generation: from step-by-step exploration of one path to observation of all admissible paths w.r.t. specified property and choosing from the proposed alternatives. While such strategy simplifies the searching process, increases the level of descriptive adequacy between desired behavior and test case obtained, and thus, enhances quality of test cases, it also assists in model debugging and revealing of undesired behavior. The approach shows its usefulness for legacy code analyzing and its behavior understanding [12]. However, the proposed approach results are informative only for models with structured control flow, in which points-of-interest are distributed along individual branches.

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 [2, 4, 8, 11]. In this respect, the new efficient searching algorithm specialized for the cumulative analysis is proposed. It stores and dynamically refines knowledge about sought coverage 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. While the asymptotic complexity is not improved and memory

consumption is even increased, in many practical cases the algorithm terminates much earlier; the analyzed state space potentially can be reduced exponentially, and thus, it significantly increases the maximum size of models for which the proposed analysis can be algorithmically computed.

References

1. Marinescu R., Kaijser H. and oth. Analyzing Industrial Architectural Models by Simulation and Model-Checking. Formal Techniques for Safety-Critical Systems. 2014. P. 189–205.
2. Hessel A., Petterson P. A global algorithm for model-based test suite generation. Electr. Notes Theor. Comp. Sci. 2007. Vol. 190. P. 47–59.
3. Rushby J. Automated test generation and verified software. Verified Software: Theories, Tools, Experiments. 2008. P. 161–172.
4. Pasareanu C., Visser W., and oth. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Automated Software Engineering. 2013. Vol. 20(3). P. 391–425.
5. Konighofer R., Hofferek G., Bloem R. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. Int. J. on Software Tools for Technology Transfer. 2013. Vol. 15. P. 563–583.
6. Heimdahl M., Devaraj G., Weber R. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In IEEE Computer society, HASE. 2004. P. 178–186.
7. Palomba F., Panichella A., and oth. Automatic Test Case Generation: What if Test Code Quality Matters? In proc. of Int. Symp. on Software Testing and Analysis. 2016. P. 130–141.
8. Kolchin A.V. Instrumental Tool Design for Asynchronous System Formal Model Testing, Cand. Sci. (Phys.-Math.) Dissertation, Kiev. 2009. 142 P. (in Russian: Razrabotka instrumentalnykh sredstv dlya proverki formalnykh modeley asinkhronnykh sistem: Dis.... kand. fiz.-mat. nauk).
9. Kolchin A. Directed search in verification of formal models. Theory and Application of Software System Development (TAAPSD'2007). 2007. P. 256–258.
10. Kolchin A., Kotlyarov V., Drobintsev P. A method of the test scenario generation in the insertion modelling environment. Control systems and machines. 2012. P. 43–48.
11. Kolchin A.V. An automatic method for the dynamic construction of abstractions of states of a formal model. Cybernetics and Systems Analysis. 2010. Vol. 46. Issue 4. P. 583–601.
12. Guba A., Kolchin O., Potiyenko S. A method for business logic extraction from legacy COBOL code of industrial systems. Proceedings of the 10th International Conference of Programming UkrPROG'2016 (Kyiv, Ukraine), CEUR-WS Vol. 1631. 2016. P. 17–25.

Литература

1. Marinescu R., Kaijser H. and oth. Analyzing Industrial Architectural Models by Simulation and Model-Checking. Formal Techniques for Safety-Critical Systems. 2014. P. 189–205.
2. Hessel A., Petterson P. A global algorithm for model-based test suite generation. Electr. Notes Theor. Comp. Sci. 2007. Vol. 190. P. 47–59.
3. Rushby J. Automated test generation and verified software. Verified Software: Theories, Tools, Experiments. 2008. P. 161–172.
4. Pasareanu C., Visser W., and oth. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Automated Software Engineering. 2013. Vol. 20(3). P. 391–425.
5. Konighofer R., Hofferek G., Bloem R. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. Int. J. on Software Tools for Technology Transfer. 2013. Vol. 15. P. 563–583.
6. Heimdahl M., Devaraj G., Weber R. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In IEEE Computer society, HASE. 2004. P. 178–186.
7. Palomba F., Panichella A., and oth. Automatic Test Case Generation: What if Test Code Quality Matters? In proc. of Int. Symp. on Software Testing and Analysis. 2016. P. 130–141.
8. Колчин А.В. Разработка инструментальных средств для проверки формальных моделей асинхронных систем: Дис. ... канд. физ.-мат. наук [Текст]. Киев. 2009. 140 с.
9. Колчин А.В. Направленный поиск в верификации формальных моделей. Тези доп. Міжнар. конф. «Теоретичні та прикладні аспекти побудови програмних систем TAAPSD'2007». Бердянск. НаУКМА, Нац. ун-т імені Тараса Шевченка, Ін-т програмних систем НАН України. 2007. С. 256–258.
10. Колчин А.В., Котляров В.П., Дробинцев П.Д. Метод генерации тестовых сценариев в среде инсерционного моделирования. Управляющие системы и машины. 2012. № 6. С. 43–48,63.
11. Колчин А.В. Автоматический метод динамического построения абстракций состояний формальной модели. Кибернетика и системный анализ. 2010. № 4. С. 70–90.
12. Губа А.А., Колчин А.В., Потиенко С.В. Метод извлечения логики поведения из промышленного программного кода на языке Кобол. Проблеми програмування. 2016. № 2–3. С. 17–25.

About author:

Kolchin Alexander,
PhD, senior scientist
30 Ukrainian papers, 11 – foreign.
http://orcid.org/0000-0001-7809-536X.

Affiliation:

V.M. Glushkov Institute of cybernetics of National Academy of Sciences of Ukraine, Kiev.
Phone: (044) 2008422.

E-mail: kolchin_av@yahoo.com.