Test Generation by Constraint Solving and FSM Mutant Killing

Alexandre Petrenko, Omer Timo, S. Ramesh

Test Generation by Constraint Solving and FSM Mutant Killing

Alexandre Petrenko [1], Omer Nguena Timo [1], S. Ramesh [2]

[1] Computer Research Institute of Montreal, CRIM
Montreal, Canada
{petrenko, omer.nguena}@crim.ca

[2] GM Global R&D
Warren, MI, USA
ramesh.s@gm.com

Abstract. The problem of fault model-based test generation from formal models, in this case Finite State Machines, is addressed. We consider a general fault model which is a tuple of a specification, conformance relation and fault domain. The specification is a deterministic FSM which can be partially specified and not reduced. The conformance relation is quasi-equivalence, as all implementations in the fault domain are assumed to be completely specified FSMs. The fault domain is a set of all possible deterministic submachines of a given nondeterministic FSM, called a mutation machine. The mutation machine contains a specification machine and extends it with mutated transitions modelling potential faults. An approach for deriving a test suite which is complete (sound and exhaustive) for the given fault model is elaborated. It is based on our previously proposed method for analyzing the test completeness by logical encoding and SMT-solving. The preliminary experiments performed on an industrial controller indicate that the approach scales sufficiently well.

Keywords: FSM, Conformance testing, Mutation testing, Fault modelling, Fault model-based test generation, Test coverage, Fault coverage analysis

1 Introduction

Fault model-based testing receives constantly growing interests of both researchers and test practitioners. Fault models are defined in the literature in a variety of ways [16]. The work [10] proposes to define a fault model as a tuple of a specification, conformance relation and fault domain. In the context of testing from finite state machines, the specification is a certain type of FSM. A conformance relation is specific to the FSM type and for completely specified deterministic machines it is equivalence, while for partially specified machines it is quasi-equivalence. The fault domain is a set of implementation machines, aka mutants, each of which models some faults, such as output, transfer and transition faults.

In the traditional checking experiment theory the fault domain is the universe of all machines with a given number of states and input and output alphabets of the specification, see, e.g., [8, 11, 12, 7, 13]. Checking experiments are in fact sound and exhaustive, i.e., complete tests. However, their size for realistic specifications is often considered too big for practical applications. To us, this is a price to pay for considering the universe of all FSMs. Intuitively, choosing a reasonable subset of this fault domain might be the way to mitigate the test explosion effect. As an example, if one considers the fault domain of mutants that model output faults, a test complete for this fault model is simply a transition tour. The fault domains intermediate to these two domains have not yet received in our opinion sufficient attention.

To define a fault domain which is a subset of the universe of all FSMs, one could explicitly enumerate mutants as in program or model-based mutation testing, see, e.g., [1, 2, 3, 21] or avoid this enumeration by defining a fault domain as a set of all possible submachines of a given nondeterministic FSM, called a mutation machine [4, 9, 6]. The mutation machine contains as a submachine a specification machine, additional transitions model potential faults. Several methods were developed for test generation using this fault model [4, 9, 6, 22]. All these methods are adaptations of classical checking experiments for a fault domain defined by a mutation machine. A checking experiment is in fact a complete test suite, however, the use of the state identification approach imposes limitations on the fault model. First, the specification machine must be completely specified and reduced, so that state identifiers exist. Second, the mutation machine was defined only for such specification machines. The existing methods are not applicable for partial specification machines and mutation machines derived from them. Finally, the state identification approach does not support iterative test generation with a mutation machine allowing the tester to terminate the process when a complete test suite for the given fault model is not yet obtained, but facing the scalability problems he is forced to make a compromise between fault coverage and test length.

Addressing the above limitations, in our recent work [20], we have developed a method for analyzing the test completeness for a fault model using a mutation machine. The analysis approach is based on logical encoding and SMT-solving, it avoids enumeration of mutants while still offering a possibility to estimate the test adequacy (mutation score). This method paves a road to a test generation approach which uses the results of the analysis to find tests which kill mutants survived a current test suite and iterates until a test suite complete for a given fault model with a mutation machine is obtained or the tester decides to terminate it earlier. Elaboration of the iterative test generation approach which is based on the test completeness analysis and does not require the specification machine to be complete and reduced is the main goal of this paper.

The remaining of this paper is organized as follows. Section 2 defines a specification model as well as a fault model. In Section 3, we develop an approach for complete test suite generation for a given fault model with a mutation machine. Section 4 reports some results of experimental evaluation of the approach. Section 5 summarizes our contributions and indicates future work.

2 Background

2.1 Finite State Machines

A *Finite State Machine* (FSM) M is a 5-tuple (S, s_0, I, O, T), where S is a finite set of states with initial state s_0; I and O are finite non-empty disjoint sets of inputs and outputs, respectively; T is a transition relation $T \subseteq S \times I \times O \times S$, (s, i, o, s') is a transition.

M is *completely specified* (complete FSM) if for each tuple $(s, x) \in S \times I$ there exists transition $(s, x, o, s') \in T$; M is *partially specified* (partial FSM), if for some $(s, x) \in S \times I$ there is no transition, we say in this case that input x is not specified in state s. Let P_M denote the set of all pairs (s, x) for which M has no transitions.

M is *deterministic* (DFSM) if for each $(s, x) \in S \times I$ there exists at most one transition $(s, x, o, s') \in T$; if there are several transitions for some $(s, x) \in S \times I$ then it is *nondeterministic* (NFSM).

An *execution* of M from state s is a sequence of transitions forming a path from s in the state transition diagram of M. The machine M is *initially connected*, if for any state $s \in S$ there exists an execution from s_0 to s. Henceforth, we assume that all FSMs are initially connected. An execution is *deterministic* if each transition (s, x, o, s') in it is the only transition for $(s, x) \in S \times I$; otherwise, i.e., if for some transition (s, x, o, s') in the execution there exists in it a transition (s, x, o', s'') such that $o \neq o'$ or $s' \neq s''$, the execution is *nondeterministic*. Clearly, a DFSM has only deterministic executions, while an NFSM can have both.

A *trace* of M in state s is a string of input-output pairs which label an execution from s. Let $Tr_M(s)$ denote the set of all traces of M in state s and Tr_M denote the set of traces of M in the initial state. Given sequence $\beta \in (IO)^*$, the *input* (*output*) *projection* of β, denoted $\beta{\downarrow_I}$ ($\beta{\downarrow_O}$), is a sequence obtained from β by erasing symbols in O (I). Given a trace β in state s the input projection $\beta{\downarrow_I}$ is an input sequence *defined in* state s. We use $\Omega_M(s)$ to denote the set of all the input sequences defined in state s and Ω_M to denote the set of all the input sequences defined in state s_0. Clearly, if M is complete then $\Omega_M = I^*$.

We say that an input sequence *triggers* an execution of M (in state s) if it is the input projection of a trace of the execution of M (in state s).

Given input sequence $\alpha \in \Omega_M$, let $out_M(s, \alpha)$ denote the set of all output sequences which can be produced by M in response to α at state s, that is $out_M(s, \alpha) = \{\beta{\downarrow_O} \mid \beta \in Tr_M(s) \text{ and } \beta{\downarrow_I} = \alpha\}$.

We define several relations between states in terms of traces. Given states s_1, s_2 of an FSM $M = (S, s_0, I, O, T)$, s_1 and s_2 are (*trace-*) *equivalent*, $s_1 \simeq s_2$, if $Tr_M(s_1) = Tr_M(s_2)$; s_2 is *trace-included* into (is a *reduction* of) s_1, $s_2 \leq s_1$, if $Tr_M(s_2) \subseteq Tr_M(s_1)$. M is *reduced* if any pair of its states are distinguishable, i.e., for every $s_1, s_2 \in S$ there exists $\alpha \in \Omega_M(s_1) \cap \Omega_M(s_2)$ such that $out_M(s_1, \alpha) \neq out_M(s_2, \alpha)$, α is called a *distinguishing* sequence for states s_1 and s_2, this is denoted $s_1 \not\simeq_\alpha s_2$ or simply $s_1 \not\simeq s_2$.

We also use relations between machines. Given FSMs $M = (S, s_0, I, O, T)$ and $N = (P, p_0, I, O, N)$, $M \leq N$ if $s_0 \leq p_0$; $N \simeq M$ if $s_0 \simeq p_0$; $N \not\simeq M$ if $s_0 \not\simeq p_0$.

In this paper, we assume that a specification machine is a DFSM which could be complete or partial, but all the implementation machines are complete DFSMs. This

implies that we should use the quasi-equivalence relation [17] as a conformance relation between implementation and specification machines. Given a DFSM $M = (S, s_0, I, O, T)$ and DFSM $N = (P, p_0, I, O, N)$, N is *quasi-equivalent* to M if M is a reduction of N.

Given a complete FSM $M = (S, s_0, I, O, T)$, a machine $N = (S', s_0, I, O, N)$ is a *submachine* of M if $S' \subseteq S$ and $N \subseteq T$. The set of all complete deterministic submachines of M is denoted $Sub(M)$. Obviously, each machine in $Sub(M)$ is a reduction of M; moreover, if M is deterministic then $Sub(M)$ contains just M.

2.2 Fault Model

We define the so-called mutation machine for a given specification machine by generalizing the definition previously given only a complete specification FSM [4, 9, 6, 20, 22] to allow the latter to be partially specified.

Definition 1. Let $A = (S, s_0, I, O, N)$ be a specification DFSM with the set of state-input pairs P_A for which A has no transitions. A complete NFSM $M = (S, s_0, I, O, T)$ is a *mutation* machine of A, if $\{(s, x, o, s') \mid (s, x) \in P_A, o \in O, s' \in S\} \subseteq T$ and A is a submachine of M.

In this definition we interpret inputs which are not specified in some states of A as don't care inputs. This implies that in a conforming implementation these inputs may cause transitions with an arbitrary output to any state in S.

The transitions T of the mutation machine can be classified as follows. The transitions in the set $\{(s, x, o, s') \mid (s, x, o, s') \in T, (s, x) \in P_A\}$ are called *don't care* transitions; we let DNC_M denote this set. The transitions in $T \cap N$ are common for M and A, these are *unaltered* transitions. Transitions in the set $T \setminus (N \cup DNC_M)$, are *mutated* transitions. Given $(s, x) \in S \times I$, we let T_{sx} denote the set of transitions from state s and input x in M. If T_{sx} is a singleton then its transition is called a *trusted* transition. The set T_{sx} is called a *suspicious* set of transitions if it is not a singleton, transitions in a suspicious set are called *suspicious*. Notice that don't care transitions are also treated as suspicious, since they can either compensate faults represented by mutated transitions and form a conforming mutant or expose wrong outputs.

We assume that all possible implementation machines for the specification machine A constitute the fault domain $Sub(M)$, the set of all deterministic submachines of the mutation machine M of A. A submachine $B \in Sub(M)$, $B \neq A$ is called a *mutant*. All mutants share all the trusted transitions, they may differ in suspicious and don't care transitions. In fact, transitions in suspicious sets are alternative and only one can be present in a deterministic mutant. Similarly, each mutant has a single transition for each pair $(s, x) \in P_A$, as all mutants are complete machines. A mutant B is *conforming* if it is quasi-equivalent to A, otherwise, it is *nonconforming*. We say that input sequence $\alpha \in \Omega_A$ *detects* or kills the mutant B if $B \not\simeq_\alpha A$.

The tuple $<A, \simeq, Sub(M)>$ is a fault model [10]. For a given specification machine A quasi-equivalence relation partitions the set $Sub(M)$ into conforming and nonconforming implementations. In this paper, we do not require the FSM A to be complete and reduced. A conforming mutant may therefore have fewer states than the specification A; on the other hand, we assume that no fault creates new states in implementations,

hence mutants with more states than the specification FSM are not in the fault domain *Sub*(*M*).

Consider the example in Fig. 1.

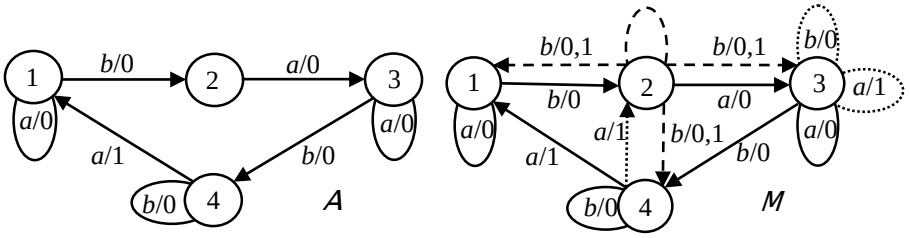

Fig. 1. A specification machine *A* and mutation machine *M*, where mutated transitions are depicted with dotted lines, don't care transitions with dashed lines; state 1 is the initial state.

The specification machine *A* in Fig. 1 is a partial DFSM, where input *b* is not specified in state 2, hence $P_A = \{(2, b)\}$. The machine is not reduced, since state 3 is quasi-equivalent to state 2. All the existing methods for test generation using mutation machines [4, 9, 6, 22] cannot be applied for such a machine, as they are based on the assumption that the specification machine is a complete and reduced machine, as required by the state identification approach.

The mutation machine *M* in Fig. 1 has three mutated transitions, one representing an output fault and the other two transfer faults. It also has 14 suspicious transitions, eight of them are don't care transitions.

The mutation machine *M* represents mutants as its deterministic submachines. Their number is given by the following formula:

$$|Sub(M)| = \prod_{(s,x)\in S\times I} |T_{sx}|$$

In our running example, the number of mutants is $8 \times 2 \times 2 \times 2 = 64$.

In the extreme case, considered in classical checking experiments a fault domain is the universe of all machines with at most *n* states, the number of states in the specification machine, and the alphabets of it. The corresponding mutation machine becomes in this case a chaos machine with all possible transitions between each pair of states. We use *Chaos*(*A*, *n*) to denote such a mutation machine for *A*. The number of FSMs it represents is the product of the numbers of states and outputs to the power of the product of the numbers of states and inputs.

3 Mutation testing

A finite set of finite input sequences $E \subset \Omega_A$ is a *test suite* for *A*. A test suite is said to be *complete* w.r.t. the fault model $\langle A, \simeq, Sub(M)\rangle$ if for each nonconforming mutant $B \in Sub(M)$ it contains a test detecting *B*.

In the case where $M = Chaos(A, n)$ a complete test suite is called n-complete. This notion coincides with the classical notion of checking experiments for the fault domain consisting of FSMs with at most n states [5, 7, 10].

In the domain of program mutation testing, such a test suite is often called adequate for a program relative to a finite collection of programs (in our case the set $Sub(M)$), see, e.g., [3].

For deterministic FSMs tests that kill a given mutant FSM can be obtained from the product of the two machines, see, e.g., [2, 1, 17]. This approach can also be used to check whether a given test kills mutants, but it requires mutant enumeration.

In this work, we develop an approach for complete test suite generation for the fault model $<A, \simeq, Sub(M)>$, where A can be a partial or complete FSM not necessary reduced. It is based on mutant killing, but does not check mutants one by one, thus avoiding their full enumeration.

3.1 Distinguishing automaton

Tests detecting mutants of the specification can be determined using a product of the specification and mutation machines obtained by composing their transitions as follows.

Definition 2 [20]. Given a specification machine $A = (S, s_0, I, O, N)$ and a mutation machine $M = (S, s_0, I, O, T)$ of A, a finite automaton $D = (C \cup \{\nabla\}, c_0, I, D, \nabla)$, where $C \subseteq S \times S$, and ∇ is an accepting (sink) state is the *distinguishing* automaton for A and M, if it holds that

- $c_0 = (s_0, s_0)$
- For any $(s, t) \in C$ and $x \in I$, $((s, t), x, (s', t')) \in D$, if there exist $(s, x, o, s') \in N$ and $(t, x, o', t') \in T$, such that $o = o'$ and $((s, t), x, \nabla) \in D$, if there exist $(s, x, o, s') \in N$ and $(t, x, o', t') \in T$, such that $o \neq o'$.

Notice that there is no outgoing transition with an input from a state of the distinguishing automata if and only if the state includes a state of A for which the input not specified.

We illustrate the definition using the specification and mutation machines in Fig. 1. Fig. 2 presents the distinguishing automaton for A and M.

The accepting state defines the language L_D of the distinguishing automaton D for A and M and possesses the following properties. First, $L_D \subset \Omega_A$, then all input sequences detecting each and every mutant belong to this language.

Theorem 1. Given the distinguishing automaton D for A and M, if $B \not\simeq_\alpha A$ for some $B \in Sub(M)$, then there exists $\beta \in L_D$, such that $B \not\simeq_\beta A$ and β is a prefix of α.

Notice that for any nonconforming mutant there exists an input sequence of length at most n^2, where n is the number of states of the specification machine, since a distinguishing automaton has no more than n^2 states.

An input sequence $\alpha \in L_D$ triggers an execution in the distinguishing automaton D which is defined by an execution in the specification machine A and some execution in

the mutation machine M triggered by α. The latter to represent a mutant must be deterministic. Such a deterministic execution of the mutation machine M defining an execution of the distinguishing automaton D to the sink state is called α-*revealing*. An input sequence triggering revealing executions enjoys a nice property of being able to detect mutants. Moreover all its extensions also detect at least the same mutants.

Theorem 2 [20]. Given an input sequence $\alpha \in \Omega_A$ such that $\alpha \in L_D$, an α-revealing execution includes at least one mutated transition, moreover, each mutant which has this execution is detected by the input sequence α.

Given an input sequence $\alpha \in L_D$, the question arises how all the mutants (un)detected by this input sequence can be characterized. We address this question in the next section.

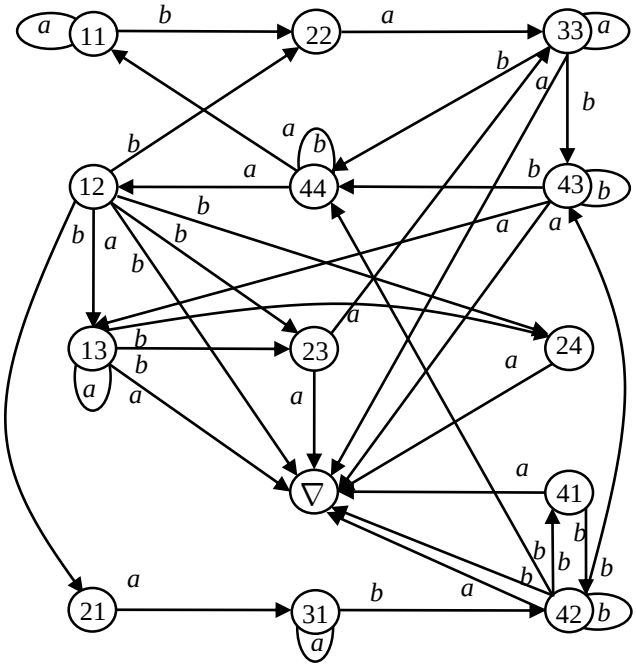

Fig. 2. The distinguishing automaton D for the specification A and mutation M machines in Fig.1, state 11 is the initial state.

3.2 Characterisation of mutants (un)detected by an input sequence

Consider an input sequence $\alpha \in \Omega_A$ whose prefixes trigger α-revealing executions. These executions characterize mutants detected by α, since each of them defines a distinct set of suspicious transitions involved in the execution. Based on these sets we can build a constraint on transition sets of mutants undetected by α. This can be achieved by using a distinguishing automaton constrained to a given input sequence.

Let *Pref*(α) be the set of all prefixes of α. We define a linear automaton (*Pref*(α), ε, *I*, D_α), such that each prefix of α is a state, and (β, *x*, β*x*) ∈ D_α if β*x* ∈ *Pref*(α).

Definition 3 [20]. Given a specification machine $A = (S, s_0, I, O, N)$, input sequence α ∈ Ω_A, and mutation machine $M = (S, s_0, I, O, T)$, a finite automaton $D_\alpha = (C_\alpha \cup \{\nabla\}, c_0, I, D_\alpha, \nabla)$, where $C_\alpha \subseteq Pref(\alpha) \times S \times S$, and ∇ is a designated sink state is the α-*distinguishing* automaton for A and M if it holds that

- $c_0 = (\varepsilon, s_0, p_0)$
- For any $(\beta, s, t) \in C_\alpha$ and $x \in I$, such that $\beta x \in Pref(\alpha)$, $((\beta, s, t), x, (\beta x, s', t')) \in D_\alpha$, if there exist $(s, x, o, s') \in N$, $(t, x, o', t') \in T$, such that $o = o'$ and $((\beta, s, t), x, \nabla) \in D$, if there exist $(s, x, o, s') \in N$, $(t, x, o', t') \in T$, such that $o \neq o'$.

For α = *bababa* in our running example, the α-distinguishing automaton for A and M is shown in Fig. 3.

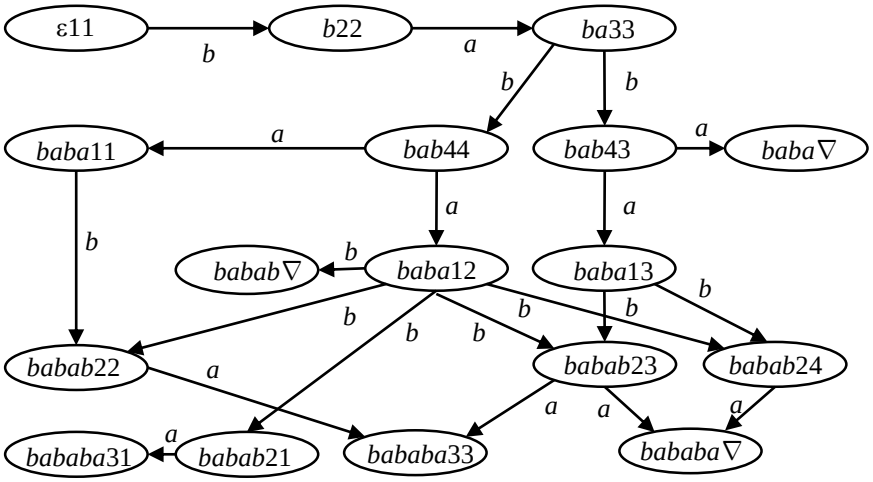

Fig. 3. The α-distinguishing automaton D_α for the specification A machine and mutation machine M in Fig. 1, where α = *bababa.*

There are eleven executions of the mutation machine listed below which are defined by five executions of the α-distinguishing automaton reaching the sink state in Fig. 3. Suspicious transitions are in bold font and the others are trusted transitions. Transitions of the specification are underlined. Three executions, namely executions 9, 10, and 11, are non-deterministic. The first eight executions belong to mutants detected by *bababa*.

1. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 3)(3, *a*, 0, 3)**
2. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 2)(2, *b*, 1, 2)**
3. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 2)(2, *b*, 1, 1)**
4. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 2)(2, *b*, 1, 3)**
5. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 2)(2, *b*, 1, 4)**
6. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 3)(3, *a*, 1, 3)(3, *b*, 0, 3)(3, *a*, 1, 3)**
7. (1, b, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 2)(2, *b*, 0, 3)(3, *a*, 1, 3)**

8. (1, *b*, 0, 2)(2, a, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 2)(2, *b*, 0, 4)(4, *a*, 1, 2)**
9. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 3)(3, *a*, 1, 3)(3, *b*, 0, 4)(4, *a*, 1, 1)**
10. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 3)(3, *a*, 1, 3)(3, *b*, 0, 4)(4, *a*, 1, 2)**
11. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 2)(2, *b*, 0, 4)(4, *a*, 1, 1)**

Three prefixes of *bababa*, namely *baba*, *babab* and *bababa* belong to L_D and trigger α–revealing executions in the mutation machine. Any deterministic execution of the mutation machine with the input sequence *baba* is α–revealing if it uses the two suspicious transitions involved in the first execution, i.e., (3, *b*, 0, 3) and (3, *a*, 0, 3). We recall that trusted transitions are used in every mutant submachine. Hence, every mutant which has both these suspicious transitions is detected by *baba*. Considering all eight executions, any mutant which has any of the eight sets of suspicious transitions is nonconforming and detected by *bababa*. Conversely, any mutant undetected by *bababa* must have mutated transitions such that do not form any of the sets defined by the listed executions. This property can be formalized as a constraint on suspicious transitions using conditional operators {=, ≠} and logical operators {∧, ∨} for constraint formulas.

Given a pair $(s, x) \in S \times I$ such that T_{sx} is a suspicious set of transitions, we introduce an auxiliary variable z_{sx} which takes values from the indexes of the transitions of the mutation machine in T_{sx}.

Each α–revealing execution e of the mutation machine involving the set of suspicious transitions $\{t_1, t_2, \ldots, t_n\}$ yields a clause $c_e = ((z_{s_1x_1} \neq t_1) \vee (z_{s_2x_2} \neq t_2) \vee \ldots \vee (z_{s_nx_n} \neq t_n))$ where s_i and x_i are the source state and the input of transition t_i for $1 \leq i \leq n$. The clause c_e is satisfied whenever $z_{s_ix_i}$ is not t_i for some $1 \leq i \leq n$. A solution of c_e excludes at least one transition in e.

In the running example, the sets of suspicious transitions indexed with an integer identifier are:

T_{2b} = {(2, *b*, 0, 2)$_3$, (2, *b*, 1, 2)$_4$, (2, *b*, 0, 1)$_5$, (2, *b*, 1, 1)$_6$, (2, *b*, 0, 3)$_8$, (2, *b*, 1, 3)$_9$, (2, *b*, 0, 4)$_{10}$, (2, *b*, 1, 4)$_{11}$}, T_{3a} = {(3, *a*, 0, 3)$_{12}$, (3, *a*, 1, 3)$_{13}$}, T_{3b} = {(3, *b*, 0, 3)$_{14}$, (3, *b*, 0, 4)$_{15}$} and T_{4a} = {(4, *a*, 1, 1)$_{17}$, (4, *a*, 1, 2)$_{18}$}. The remaining trusted transitions are indexed as follows: (1, *a*, 0, 1)$_1$, (1, *b*, 0, 2)$_2$, (2, *a*, 0, 2)$_7$, (4, *b*, 0, 2)$_{16}$. So we consider four variables z_{2b}, z_{3a}, z_{3b} and z_{4a} whose domains are T_{2b}, T_{3a}, T_{3b}, and T_{4a}, respectively. To simplify the presentation we use transition identifiers in constraints. The constraint formula for *bababa* consists of a preamble and eight clauses. The preamble $(z_{2b} \in T_{2b}) \wedge (z_{3a} \in T_{3a}) \wedge (z_{3b} \in T_{3b}) \wedge (z_{4a} \in T_{4a})$ specifies the domains of the variables, thus constraining the possible solutions of the conjunction of the eight clauses to transitions of the mutation machine.

$((z_{2b} \in T_{2b}) \wedge (z_{3a} \in T_{3a}) \wedge (z_{3b} \in T_{3b}) \wedge (z_{4a} \in T_{4a}) \wedge ((z_{3a} \neq 12) \vee (z_{3b} \neq 14)) \wedge ((z_{4a} \neq 18) \vee (z_{2b} \neq 4) \vee (z_{3b} \neq 15)) \wedge ((z_{4a} \neq 18) \vee (z_{2b} \neq 6) \vee (z_{3b} \neq 15)) \wedge ((z_{4a} \neq 18) \vee (z_{2b} \neq 9) \vee (z_{3b} \neq 15)) \wedge ((z_{4a} \neq 18) \vee (z_{2b} \neq 11) \vee (z_{3b} \neq 15)) \wedge ((z_{3a} \neq 13) \vee (z_{3b} \neq 14)) \wedge ((z_{4a} \neq 18) \vee (z_{2b} \neq 8) \vee (z_{3a} \neq 13) \vee (z_{3b} \neq 15)) \wedge ((z_{4a} \neq 18) \vee (z_{2b} \neq 10) \vee (z_{3b} \neq 15))).$

Clearly, the constraint always has a solution where values of variables determine all the unaltered transitions, but to find nonconforming mutants we need a solution if it exists which has at least one mutated transition. To this end, we add the constraint $((z_{3a} \neq 12) \vee (z_{3b} \neq 15) \vee (z_{4a} \neq 17))$ excluding the solution defining the specification machine

augmented with an arbitrary don't care transition, called a completed specification machine.

The final constraint formula for *bababa* is $C(bababa)$= (($z_{2b} \in T_{2b} \wedge z_{3a} \in T_{3a} \wedge z_{3b} \in T_{3b} \wedge z_{4a} \in T_{4a}$) $\wedge$ (($z_{3a} \neq 12$) $\vee$ ($z_{3b} \neq 14$)) $\wedge$ (($z_{4a} \neq 18$) $\vee$ ($z_{2b} \neq 4$) $\vee$ ($z_{3b} \neq 15$)) $\wedge$ (($z_{4a} \neq 18$) $\vee$ ($z_{2b} \neq 6$) $\vee$ ($z_{3b} \neq 15$)) $\wedge$ (($z_{4a} \neq 18$) $\vee$ ($z_{2b} \neq 9$) $\vee$ ($z_{3b} \neq 15$)) $\wedge$ (($z_{4a} \neq 18$) $\vee$ ($z_{2b} \neq 11$) $\vee$ ($z_{3b} \neq 15$)) $\wedge$ (($z_{3a} \neq 13$) $\vee$ ($z_{3b} \neq 14$)) $\wedge$ (($z_{4a} \neq 18$) $\vee$ ($z_{2b} \neq 8$) $\vee$ ($z_{3a} \neq 13$) $\vee$ ($z_{3b} \neq 15$)) $\wedge$ (($z_{4a} \neq 18$) $\vee$ ($z_{2b} \neq 10$) $\vee$ ($z_{3b} \neq 15$)) $\wedge$ (($z_{3a} \neq 12$) $\vee$ ($z_{3b} \neq 15$) $\vee$ ($z_{4a} \neq 17$))).

The constraint $C(\alpha)$ characterizing the mutants undetected by an input sequence α is the conjunction of a clause excluding all completed specification machines and the clauses generated for every α–revealing execution. Any existing constraint solver, e.g., Z3 [15], could be used for satisfiability checking. A solution of $C(\alpha)$ if it exists is an assignment of the auxiliary variables. The mutant defined by such a solution includes the transitions specified by the solution along with all the trusted transitions of the mutation machine. Any nonconforming mutant detected by α cannot be defined by any solution of $C(\alpha)$, only conforming mutants can.

Algorithm 1 presents a procedure that builds a constraint C_{TS} for a given test suite *TS* out of the constraints for each test in the test suite.

Algorithm 1. ***Build_Constraint***
Input: *TS*, a test suite
A, a specification machine
M, a mutation machine for *A*
Output: C_{TS}, a constraint specifying mutants undetected by tests in *TS*

1. C_{TS} = True
2. **For each** input sequence $\alpha \in TS$,
   (a) Determine the α-distinguishing automaton D_α and set c_α = True
   (b) **For each** α-revealing execution *e* of the mutation machine, set $c_\alpha = c_\alpha \wedge c_e$, where c_e is the disjunction of constraints excluding the suspicious transitions in *e*
   (c) $C_{TS} = C_{TS} \wedge c_\alpha$
3. Add to C_{TS} the constraints restricting the value of the variables to their domain and excluding *A*
4. **Return** C_{TS}

Theorem 3. Let C_{TS} be a constraint specifying undetected mutants by the test suite *TS*. *TS* is complete w.r.t. the fault model $\langle A, \simeq, Sub(M)\rangle$ if and only if C_{TS} is unsatisfiable or every solution of C_{TS} defines a conforming mutant.

In the running example, to solve the constraint formula $C(bababa)$, we use the SMT solver Z3 [15] which finds the solution $z_{2b} = 11$, $z_{3a} = 13$, $z_{3b} = 15$, $z_{4a} = 17$. The solution defines a mutant with all trusted transitions, one don't care transition $(2, b, 1, 4)_{11}$ and

one mutated transition $(3, a, 1, 3)_{13}$. The mutant is presented in Fig. 4. The mutant is nonconforming, which can be verified with the help of a distinguishing automaton obtained for the specification machine and the mutant also shown in Fig.4.

Notice that the solver could find another solution of the $C(bababa)$, namely, $z_{2b} = 11$, $z_{3a} = 12$, $z_{3b} = 15$, $z_{4a} = 17$ which defines a conforming mutant.

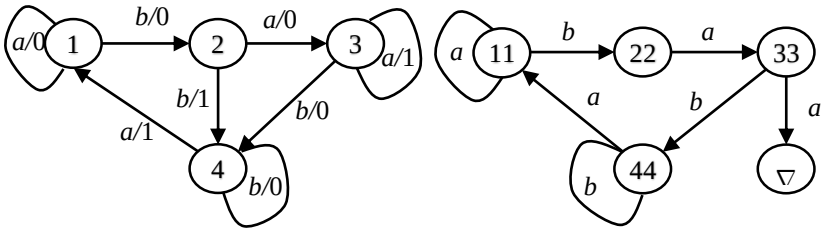

Fig. 4. A nonconforming mutant defined by a solution of the constraint for $TS = \{bababa\}$ and the distinguishing automata for the mutant and the specification A.

Given an initial test suite TS_{init}, the question arises how to augment TS_{init} with new input sequences to detect all nonconforming mutants. We elaborate a complete test suite generation procedure in the next section.

3.3 Complete Test Suite Generation

We are given a test suite $TS_{init} \subseteq \Omega_A$ and a fault model $\langle A, \simeq, Sub(M) \rangle$. We want to add test cases to TS_{init} to obtain a complete test suite. Constraints defined in the previous section can be used to analyse the completeness of a test suite, elaborated in our previous work [20]. If the constraint for a test suite has no solution or the first solution computed by a solver defines a nonconforming mutant, we can immediately assert the incompleteness of the test suite. If however the solution defines a conforming mutant, the search continues such that the mutant will not be found again in a new round of satisfiability checking. This process iterates until no new solution is found or the generated solution defines a nonconforming mutant. A witness nonconforming mutant can be used to determine a test case which detects the mutant. A new test can kill other nonconforming mutants. Hence the constraints generated by this test should be added to the ones of the current test suite. The search terminates when the current constraint is unsatisfiable which indicates that the test suite is complete. The test generation procedure is formalized in Algorithm 2.

The algorithm has two loops. The coverage analysis loop includes statements in lines 7 to 9 and the test generation loop includes statements at lines 4 to 13. The former loop is nested in the latter. When the current test suite is not complete, the execution of the test generation loop augments (in line 4) it with a new test case and updates the constraint of the current test suite with that of the generated test case. Then the coverage analysis loop is executed checking the completeness of the updated test suite, searching for a new nonconforming mutant. To this end, constraints excluding conforming mutants defined by the found solutions are iteratively added to the current constraint. The procedure terminates when the resulting constraint is unsatisfiable, indicating that the

current test suite is complete or the solver generates a solution defining a nonconforming mutant. In this case, the execution of the test generation loop augments the current test suite with a new test killing the mutant. It is determined by finding the shortest path to the sink state of the distinguishing automaton of the mutant and the specification.

Algorithm 2. ***TestSuiteGen***
Inputs: TS_{init}, an initial test suite
A, a specification machine
M, a mutation machine for A
Output: TS, a complete test suite

1. $TS = \emptyset$
2. C = True
3. $TS_{new} = TS_{init}$
4. $TS = TS \cup TS_{new}$
5. Let C_{new} = ***Build_Constraint***(TS_{new}, A, M) be a constraint specifying mutants undetected by TS_{new}
6. Let $C = C \wedge C_{new}$
7. Check the satisfiability of C by calling a solver
8. ***if*** C is "*unsatisfiable*" ***then*** terminate with "TS is a complete test suite"
9. *Let* N be a mutant defined by a solution of C obtained by a solver
10. ***if*** N is conforming ***then*** set $C = C \wedge C_N$ where C_N is the constraint that excludes N and ***Goto*** 7
11. Determine D, the distinguishing automaton of A and N
12. Let $TS_{new} = \{\alpha\}$, where $\alpha \in L_D$
13. ***Goto*** 4

Theorem 4. Procedure *TestSuiteGen* always terminates with a complete test suite.

Proof. The procedure *TestSuiteGen* terminates as the test generation loop terminates. It does so because a mutation machine has a finite number of submachines and the solution defining a particular mutant is generated at most once. According to Theorem 2, the computed tests are revealing input sequences. On termination of the procedure the final constraint characterizing undetected mutants excludes all conforming mutants and it is unsatisfiable. Based on Theorem 3 we have that the test suite returned by the procedure is complete.

To generate a complete test suite for the running example we consider the initial test suite TS_{init} = {*bababa*}, used in Section 3.2. Procedure *TestSuiteGen* first goes into the test generation loop which builds $C_{new} = C$(*bababa*), updates *TS* and *C* to {*bababa*} and *C*(*bababa*) defined in Section 3.2. Then the coverage analysis loop finds the solution of *C* defining the nonconforming mutant in Fig 4. The test *baa* is then generated by determining the only shortest path to the sink state in the distinguishing automaton for the specification and the mutant, so TS_{new} becomes {*baa*}. Then the constraint C_{new} =

$C(baa) = (z_{3a} \neq 13)$ is generated in line 5, *TS* becomes {*bababa*, *baa*} and $C = C(bababa) \wedge C(baa)$. The solution of *C* defines a nonconforming mutant killed by the test *babaaba*, the input sequence of the shortest path to the sink state the distinguishing automaton for the new nonconforming mutant and the specification, then TS_{new} becomes {*babaaba*}. The constraint $C_{new} = C(babaaba) = (((z_{3a} \neq 12) \vee (z_{3b} \neq 14)) \wedge ((z_{3a} \neq 13) \vee (z_{3b} \neq 14)) \wedge ((z_{4a} \neq 18) \vee (z_{3b} \neq 15)))$ is generated; it includes one clause for each of the three α-revealing executions triggered by *babaaba*. *TS* becomes {*bababa*, *baa*, *babaaba*} and $C = C(bababa) \wedge C(baa) \wedge C(babaaba)$. *C* is satisfiable and the procedure iteratively generates only conforming mutants augmenting *C* each time with a constraint excluding the last conforming mutant. No additional tests are generated. The conjunction of *C* with the eight constraints excluding eight conforming mutants is unsatisfiable. The procedure returns a complete test suite *TS* = {*bababa*, *baa*, *babaaba*}. Notice that it generated only ten out of total 64 mutants.

The procedure *TestSuiteGen* also generates a complete test suite starting when the initial test suite contains just an empty input sequence.

In the next section we present experimental results obtained with a prototype tool.

4 Experimental results

We have developed a prototype tool implementing the proposed method for complete test suite generation. In this section we present the tool and some experimental results using it.

4.1 Prototype tool

The prototype tool is composed of four modules: an I/O module, a completeness checking module, a test generation module and a module for solver execution. The I/O module converts input data into an internal representation for processing and obtained results into a human-readable format. To this end, it implements an ANTLR-based parser [19] to interpret the mutation machine specified in a text format; it also parses the output of SMT solver Z3 [15] to extract a solution and builds a mutant. The completeness checking module builds α-distinguishing automata, determines revealing executions of the mutation machine and generates constraints for the solver. The test generation module iteratively calls the former module. The prototype can also be used with other SMT solvers compatible with the SMT-LIB 2.0.

For the experiments we use a desktop computer with the following settings: 3.4 Ghz Intel Core i7-3770 CPU, 16.0 GB of RAM, Z3 4.3.2, and ANTLR 4.5.1.

4.2 Test Generation for an Automotive Controller

We consider as a case study an automotive controller of the air quality system (HVAC), which we also used in our previous work [18, 20]. The functionality of the controller is to set an air source position depending on its current state and input from the environment.

The controller initially specified as a hierarchical Simulink Stateflow model is converted into an FSM with 14 states, 24 inputs and 24 x 14 = 336 transitions.

Several mutation machines were used in the experiments. The first one M_{hvac} was obtained by adding 46 mutated transitions to the specification machine (details are available in [20]). The formula in Section 2 gives the number of mutants $3^{12} \times 2^{17} = 69{,}657{,}034{,}752$.

The other mutation machines were built by adding more mutated transitions to M_{hvac}. In particular, 20, 100, 428, 764 and 1000 mutated transitions were randomly added, resulting in five more mutation machines, M_{+20}, M_{+100}, M_{+428}, M_{+764} and M_{+1000}. Tab. 1 presents the numbers of mutated transitions, mutants, generated tests and the computation time. Each generated mutant was non-conforming, so their number coincides with that of the tests, conforming mutants were never generated. The third column of Tab. 1 represents the average values for 30 mutation machines randomly generated by adding 20 mutated transitions to M_{hvac}.

Tab. 1. Experimental results for randomly generated mutation machines

	M	M_{+20}	M_{+100}	M_{+428}	M_{+764}	M_{+1000}
Mut. Trans.	46	66	146	474	810	1046
Mutants	6.9×10^{10}	3.6×10^{16}	9.8×10^{38}	7.8×10^{108}	4.9×10^{160}	3.2×10^{194}
Tests	29	48	119	381	520	689
Seconds	0.57	0.7	1.8	8.6	58	154

The experimental results indicate that the approach scales sufficiently well on a typical automotive controller even with the large number of mutants.

5 Conclusions

In this paper we focused on generation of a complete test suite detecting all nonconforming implementations in a fault domain defined by a mutation machine. A mutation machine is a nondeterministic FSM, interpreted as a compact representation of a set of deterministic implementations of a system represented by a partially or completely specified FSM. Each deterministic submachine of the mutation machine models an implementation.

We proposed a method for generating a complete test suite which avoids complete enumerations of nonconforming mutants. The method iteratively builds constraints specifying mutants undetected by an incomplete (possibly empty) test suite and uses a solution of constraints generated by a solver to determine an undetected mutant from which a new test case is selected and derives an augmented constraint for a next iteration step until the obtained constraint becomes unsatisfiable.

While it enumerates all conforming mutants, which exist mostly when the specification is partial or unreduced FSM, it does not generate all nonconforming ones. The experimental results with a prototype tool which uses the SMT solver Z3 indicate that the number of generated nonconforming mutants reaches only a small percentage of all mutants represented by a mutation machine.

Novelty of the contributions of this paper are as follows. The proposed approach allows one to construct checking experiments for FSMs which are not necessarily complete and reduced without using any state identification facility such as characterization sets, distinguishing sequences, and state identifiers, as opposed to traditional checking experiment approaches. Thus, we demonstrate that it is possible to construct checking experiments using logical encoding and constraint solving instead of classical methods based on state identification [4, 9, 6, 22]. Moreover, test completeness is guaranteed for a predefined subset of the universe of all FSMs with a given number of states, represented by a mutation machine. Compared to all previous work on the use of mutation machine [4, 9, 6, 20, 22], we have generalized its definition to make it applicable to partially defined specification machines. The method proposed in [22] is only applicable to mutation machines which satisfy the following assumption. If a transition of the specification machine becomes suspicious in the mutation machine then the latter has all possible (thus chaotic) suspicious transitions from the start state of the transition caused by the same input. The method also requires the specification machine be completely specified. Compared to that work, our method is applicable to arbitrary mutation machines, while the specification machine is allowed to be partially specified.

Another interesting feature of the approach is that it is iterative and allows the tester to obtain an incomplete test suite for which fault coverage can be estimated (as discussed in [20]) when facing the scalability problems he is forced to make a compromise between fault coverage and test length.

The experiments indicate that the proposed approach may scale sufficiently well, though, more experiments with industrial size specifications are needed. Our current work focuses on extending the approach to FSMs with symbolic inputs and outputs [23] and eventually to a more general type of EFSM [14].

Acknowledgements. This work is supported in part by GM, NSERC and MEIE of Gouvernement du Québec.

References

1. Pomeranz, I., Sudhakar M. R.: Test generation for multiple state-table faults in finite-state machines. IEEE Transactions on Computers 46 (7), pp. 783-794 (1997)
2. Poage, J.F., McCluskey, Jr., E. J.: Derivation of optimal test sequences for sequential machines. In: Proceedings of the IEEE 5th Symposium on Switching Circuits Theory and Logical Design, pp. 121-132 (1964)
3. DeMilli, R. A., Offutt, J.A.: Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17(9), pp. 900-910 (1991)
4. Grunsky, I.S., Petrenko, A.: Design of checking experiments with automata describing protocols. Automatic Control and Computer Sciences. Allerton Press Inc. USA. No. 4 (1988)
5. Hennie, F. C.: Fault detecting experiments for sequential circuits. In: Proceedings of the IEEE 5th Annual Symposium on Switching Circuits Theory and Logical Design. Princeton, pp. 95-110 (1964)
6. Koufareva, I., Petrenko, A., Yevtushenko, N.: Test generation driven by user-defined fault models. In: Proceedings of the 12th International Workshop on Testing of Communicating Systems, pp. 215-233 (1999)

7. Lee, D., Yannakakis, M.: Principles and methods of testing finite-state machines - a survey. Proceedings of the IEEE, vol. 84, No. 8, pp. 1090-1123 (1996)
8. Moore, E.F.: Gedanken experiments on sequential machines. In: Automata Studies. Princeton University Press, pp. 129-153 (1956)
9. Petrenko, A., Yevtushenko, N.: Test suite generation for a FSM with a given type of implementation errors. In: Proceedings of IFIP 12th International Symposium on Protocol Specification, Testing, and Verification, pp. 229-243 (1992)
10. Petrenko, A., Yevtushenko, N., Bochmann, G. v.: Fault models for testing in context. In Formal Description Techniques IX, Springer, pp. 163-178 (1996)
11. Vasilevskii, M.P.: Failure diagnosis of automata. Cybernetics. Plenum Publishing Corporation. New York. 4, pp. 653-665 (1973)
12. Chow T.S.: Testing software design modeled by finite-state machines. Transactions on Software Engineering, 4(3), IEEE, pp. 178–187 (1978)
13. Vuong, S.T., Ko, K.C.: A novel approach to protocol test sequence generation. Global Telecommunications Conference 3, IEEE, pp. 2 -5 (1990)
14. Petrenko, A., Boroday, S., Groz, R.: Confirming configurations in EFSM testing. Transactions on Software Engineering 30(1), IEEE, pp. 29-42 (2004).
15. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In Proc. of Tools and Algorithms for the Construction and Analysis of Systems, Springer, pp. 337-340 (2008)
16. Bochmann, G. v., et al.: Fault models in testing. In Proceedings of the IFIP TC6/WG6. 1 Fourth International Workshop on Protocol Test Systems. North-Holland Publishing Co, pp. 17-30 (1991)
17. Petrenko, A., Yevtushenko, N.: Testing from partial deterministic FSM specifications. Transactions on Computers 54(9), IEEE, pp. 1154-1165 (2005)
18. Petrenko, A., Dury, A., Ramesh, S., Mohalik, S.: A method and tool for test optimization for automotive controllers. In ICST Workshops, IEEE, pp. 198–207 (2013)
19. Parr, T.: The definitive ANTLR 4 reference, vol 2. Pragmatic Bookshelf, Raleigh (2013)
20. Petrenko, A, Nguena Timo, O., Ramesh, S.: Multiple mutation testing from FSM. In Proceedings of the 35th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems, pp. 222-238 (2016)
21. Belli, F., Budnik, C. J., Hollmann, A., Tuglular, T., Wong, W. E.: Model-based mutation testing - Approach and case studies. Science of Computer Programming, 120, pp. 25-48 (2016)
22. El-Fakih, K., Dorofeeva, R., Yevtushenko N., and Bochmann, G. v.: FSM-based testing from user defined faults adapted to incremental and mutation testing. Programming and Computer Software, 38, pp. 201-209 (2012)
23. Petrenko, A.: Checking experiments for symbolic input/output finite state machines. In ICST Workshops, IEEE, pp. 229-237 (2016)