Multiple Mutation Testing from Finite State Machines with Symbolic Inputs

Omer Nguena Timo, Alexandre Petrenko, S. Ramesh

Multiple Mutation Testing from Finite State Machines with Symbolic Inputs

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

[1] Computer Research Institute of Montreal, CRIM
Montreal, Canada
{omer.nguena-timo, petrenko}@crim.ca
[2] GM Global R&D
Warren, MI, USA
ramesh.s@gm.com

Abstract. Recently, we proposed a mutation-testing approach from a classical finite state machine (FSM) for detecting nonconforming mutants in a given fault domain specified with a so-called mutation machine. In this paper, we lift this approach to a particular type of extended finite state machines called symbolic input finite state machine (SIFSM), where transitions are labeled with symbolic inputs, which are predicates on input variables possibly having infinite domains. We define a well-formed mutation SIFSM for describing various types of faults. Given a mutation SIFSM, we develop a method for evaluating the adequacy of a test suite and a method for generating tests detecting all nonconforming mutants. Experimental results with the prototype tool we have developed indicate that the approach is applicable to industrial-like systems.

Keywords: Extended FSM · Symbolic inputs · Conformance testing · Mutation testing Fault modelling · Fault model-based test generation · Constraint solving

1 Introduction

Detecting nonconforming implementations is a major challenge during the design and the maintenance of systems, which motivates the elaboration of innovative and efficient testing [16,23], model-checking [6] and runtime verification techniques [12]. Testing techniques [23] not only aim at exercising a system with adequate test cases to reveal failures and ideally to identify and to repair faults causing the failures. They may also target evaluating the adequacy of test cases and the generation of test cases to cover artefacts that can conceal faults [5,11,4,2], e.g., statements, branches, interfaces, requirements, mutants. Mutants which are versions of a specification of a system seeded with undesired faults can be used to generate test cases or to determine the adequacy of given test cases to reveal the faults. A fault domain can be specified with a set of mutants and test cases detecting the mutants which do not conform to the specification can be applied to detect faulty implementations of a system. Classical FSM model is often used in developing fault model based testing approaches for detecting nonconforming implementations. Recently we proposed an approach for this model to evaluate the adequacy of test cases in a given fault domain [19] and to generate test cases detecting all nonconforming mutants [20].

In case the testers need to deal with inputs with infinite domains, the finite input alphabets which is used in classical FSM to represent the inputs of a system becomes ineffective along with FSM-based testing approaches. In the automotive applications, the behaviors of some controllers [18] depend on the truth values of predicates defined over input variables with infinite domains. Extensions of FSMs with symbolic inputs and arithmetic operations on variables have been proposed [3,21,14] to relax limitations of the classical FSM and used in developing testing methods [9,21,14]. Following the same trend, our test generation method by constraint solving from FSM in [19,20] could be enhanced to extended FSM. The work of [8] also uses an EFSM model and a mutation machine to model transition and output faults. Test generation requires (partial) unfolding of the specification, which we completely avoid. A test suite complete for used defined faults can only be generated if they satisfy certain sufficient conditions, which severely restrict types of detectable transition and output faults. Moreover, faults in transition predicates are not considered, as opposed to our approach.

In this paper, we lift the mutation testing approach from classical FSM in [19,20] to symbolic input finite state machine (SIFSM). SIFSM [21] is an extension of FSM with inputs specified with predicates on input variables possibly having infinite domains, which permits a more compact representation of data, data-flow relations and control-flow for determining outputs depending on the values of the predicates and states. Examples of realistic systems which can be specified with SIFSM can be found in [18,10]. The contribution is three-fold. First, we define mutation operations for building well-formed mutation machines specifying mutants in fault domains. New mutation operations may change predicates used in the specification or introduce new predicates. Secondly, we propose a method for evaluating the completeness of a test suite, i.e., the adequacy of a test suite to detect all nonconforming mutants. Finally we propose a method for generating complete test suites. Following the ideas in our previous work [19,20], the methods rely on building and resolving constraints specifying the mutants undetected by given test cases. However, in this work the constraints differ from those in our previous work; they are represented with Boolean expressions for expressing both undetected mutants and the input-completeness property of the mutants. The latter property is formalized with a notion of *cluster* for state. This is needed because predicates cannot be mutated independently. We evaluate the methods with a prototype tool applied to a SIFSM model of a component from the automotive domain.

The remaining of the paper is organized as follows. Section 2 introduces mutation SIFSM and mutation operations used for its creation. In Section 3 we present an approach for determining the mutants undetected by a test, which leads to a method for completeness checking of a given test suite in Section 4. In Section 5 we develop a method for complete test suite generation. Section 6 reports some experimental evaluation of the approach. We summarize our contributions in Section 7.

2 Background

2.1 Preliminaries

Let G denote the universe of *inputs* that are predicates over variables in a fixed set V for which a decision theory, e.g., an SMT solver, exists, excluding the predicates that

are always *false*. G^* denotes the universe of *input sequences* and ε denotes the empty sequence. Later in the paper, a *test* is just an input sequence. Let I_V denote the set of all the valuations of the input variables in the set V, called *concrete inputs*. A set of concrete inputs is called a *symbolic input*; both, concrete and symbolic inputs are represented by predicates in G. Henceforth, we use set-theoretical operations on inputs. In particular, we say that concrete input x satisfies symbolic input g if $x \in g$. We also have that $I_V \subseteq G$. A set of inputs H is a *tautology* if each concrete input $x \in I_V$ satisfies at least one input in it, i.e., $\{x \in g \mid g \in H\} = I_V$.

We define some relations between input sequences in G^*. Given two input sequences $\alpha, \beta \in G^*$ of the same length k, $\alpha = g_1 g_2 \dots g_k, \beta = g'_1 g'_2 \dots g'_k$, we let $\alpha \cap \beta = g_1 \cap g'_1 \dots g_k \cap g'_k$ denote the sequence of intersections of inputs in sequences α and β; α and β are *compatible*, if for all $i = 1, \dots, k$, $g_i \cap g'_i \neq \emptyset$. We say that α is a *reduction* of β, denoted $\alpha \subseteq \beta$, if $\alpha = \alpha \cap \beta$. If α is a sequence of concrete inputs as well as a reduction of β then it is called an *instance* of β; given a finite set of input sequences $E \subseteq G^*$, a set of concrete input sequences is called an instance of the set E, if it contains at least one instance for each input sequence in E.

Given a finite set of outputs O, a *trace* is a sequence of input-output pairs in $(G \times O)^*$. A trace is *concrete* if every input in it is concrete; otherwise it is *symbolic*. Given a trace $\beta \in (G \times O)^*$, the input (resp. output) projection of β, denoted $\beta_{\downarrow G}$ (resp. $\beta_{\downarrow O}$), is a sequence obtained from β by erasing symbols in O (resp. G).

We consider an extension of FSM called symbolic input finite state machine (SIFSM) [21], which operates in discrete time as a synchronous machine reading values of input variables and setting up the values of output variables. Output variables are assumed to have a finite number of valuations and form a finite output alphabet. On the other hand, the set of input valuations can be infinite.

Definition 1. *A* symbolic input finite state machine $\mathcal{S}$ *(or machine, for short) is a 5-tuple* (S, s_0, V, O, T)*, where*

- S *is a finite set of* states *with the* initial *state* s_0*,*
- V *is a finite set of* input variables *over which inputs in* G *are defined,*
- O *is a finite set of* outputs*,*
- $T \subseteq S \times G \times O \times S$ *is a finite* transition relation*,* $(s, g, o, s') \in T$ *is a* transition*.*

The semantics of SIFSM is defined by a Mealy state machine with a possibly infinite input set, where the state and output sets remain finite. The set of transitions outgoing from state s is denoted by $T(s)$. We say that input g is *defined* in state s if g is the input of a transition in $T(s)$. Then, $G(s)$ denotes the sets of all the inputs defined in s. We say that transition (s, g, o, s') is *triggered* by input g' if g' is a reduction of g. Several transitions in $T(s)$ are *nondeterministic* if they can be triggered by the same input. If a set of transitions $T(s)$ includes nondeterministic transitions, the set is said to be *nondeterministic*; otherwise it is *deterministic*.

An *execution* of $\mathcal{S}$ from state s is a sequence of transitions $t_1 t_2 \dots t_n$ forming a path from s in the state transition diagram of $\mathcal{S}$. A *deterministic execution* is an execution such that its set of transitions is deterministic; otherwise, i.e., if for some state s and some transition in the execution there exists another transition such that both transitions belong to $T(s)$ and are triggered by an identical input, the execution is *nondeterministic*.

A *symbolic trace of* $\mathcal{S}$ *in state* s is the projection of an execution from s on the input-output pairs in $(G \times O)$. A trace obtained from a symbolic trace in s by substituting every inputs by an instance of it is called a *concrete trace of* $\mathcal{S}$ *in* s. Let $Tr_{\mathcal{S}}(s)$ (resp. $STr_{\mathcal{S}}(s)$) denote the set of all concrete (resp. symbolic) traces of $\mathcal{S}$ in state s and $Tr_{\mathcal{S}}$ (resp. $STr_{\mathcal{S}}$) denote the set of concrete (resp. symbolic) traces of $\mathcal{S}$ in the initial state.

We say that an input sequence *triggers* an execution of $\mathcal{S}$ (in state s) if it is a reduction of the input projection of a trace of the execution of $\mathcal{S}$ (in state s). Given an input sequence α, let $out_{\mathcal{S}}(s, \alpha)$ denote the set of all output sequences which can be produced by $\mathcal{S}$ in response to α at state s, that is $out_{\mathcal{S}}(s, \alpha) = \{\beta_{\downarrow O} \mid \beta \in STr_{\mathcal{S}}(s) \text{ and } \alpha \subseteq \beta_{\downarrow G}\}$. We observe that $out_{\mathcal{S}}(s, \alpha) = out_{\mathcal{S}}(s, \gamma)$ whenever γ is a reduction of input α.

The machine $\mathcal{S}$ is *deterministic* (DSIFSM), if for every state s, $T(s)$ is deterministic; otherwise $\mathcal{S}$ is a *nondeterministic* SIFSM (NSIFSM). Clearly, a DSIFSM has only deterministic executions, while an NSIFSM can have both. State s of $\mathcal{S}$ is *completely specified*, if $G(s)$ is a tautology, i.e., each concrete input $x \in I_V$ satisfies at least one input defined at s. The machine $\mathcal{S}$ is *completely specified*, if each state is completely specified. The machine $\mathcal{S}$ is *initially connected*, if for any state $s \in S$ there exists an execution from s_0 to s. Henceforth, we assume that all SIFSM are initially connected and completely specified.

We adapt several relations introduced in [19,20] for FSM to SIFSM and use trace-based definitions of the relations introduced in [21]. Given states s_1, s_2 of a SIFSM $\mathcal{S} = (S, s_0, V, O, T)$, s_1 and s_2 are *(trace-) equivalent*, $s_1 \simeq s_2$, if $Tr_{\mathcal{S}}(s_1) = Tr_{\mathcal{S}}(s_2)$; s_1 and s_2 are *distinguishable*, $s_1 \not\simeq s_2$, if $Tr_{\mathcal{S}}(s_1) \neq Tr_{\mathcal{S}}(s_2)$; s_2 is *trace-included* into (is a reduction of) s_1, $s_2 \leq s_1$, if $Tr_{\mathcal{S}}(s_2) \subseteq Tr_{\mathcal{S}}(s_1)$. $\mathcal{S}$ is *reduced* if any pair of its states is distinguishable. Given two distinguishable states s_1 and s_2, there exists a sequence $\alpha \in G^*$ such that $out_{\mathcal{S}}(s_1, \alpha) \neq out_{\mathcal{S}}(s_2, \alpha)$; α is called a *distinguishing input sequence* for states s_1 and s_2, this is denoted $s_1 \not\simeq_{\alpha} s_2$.

We also use relations between machines. Given SIFSM $\mathcal{S} = (S, s_0, V, O, T)$ and $\mathcal{P} = (P, p_0, V, O, N)$, $\mathcal{P} \leq \mathcal{S}$ if $p_0 \leq s_0$; $\mathcal{P} \simeq \mathcal{S}$ if $p_0 \simeq s_0$; $\mathcal{P} \not\simeq_{\alpha} \mathcal{S}$ if $p_0 \not\simeq_{\alpha} s_0$ with $\alpha \in G^*$; and $\mathcal{P} \not\simeq \mathcal{S}$ if $\mathcal{P} \not\simeq_{\alpha} \mathcal{S}$ for some distinguishing input sequence α for p_0 and s_0. Later, we use equivalence relation between machines as a conformance relation between implementation and specification machines.

Given a NSIFSM $\mathcal{S} = (S, s_0, V, O, T)$, a machine $\mathcal{P} = (P, p_0, V, O, N)$ is a *submachine* of $\mathcal{S}$ if $p_0 = s_0$, $P \subseteq S$ and $N \subseteq T$.

2.2 Mutation machine

Let $\mathcal{S} = (S, s_0, V, O, N)$ be a DSIFSM, called the specification machine.

Definition 2. *A NSIFSM* $\mathcal{M} = (S, s_0, V, O, T)$ *is a* mutation machine *of* $\mathcal{S}$*, if* $\mathcal{S}$ *is a submachine of* $\mathcal{M}$*.*

Transitions of $\mathcal{M}$ that are also transitions of S are called *unaltered*, while the others, in the set $T \setminus N$, are *mutated* transitions. A transition of $\mathcal{M}$ is *suspicious* if it belongs to a nondeterministic set of transitions, let *Susp*(s) denote the set of all suspicious transitions in state s and *Susp*$(\mathcal{M})$ denote the set of all suspicious transitions of $\mathcal{M}$. An unaltered transition is *trusted* if it is not suspicious; otherwise it is *untrusted* and belongs to the set *Untr*$(\mathcal{S})$ = *Susp*$(\mathcal{M}) \cap N$. Given state s, a subset of $T(s)$ is called a *cluster* of s if

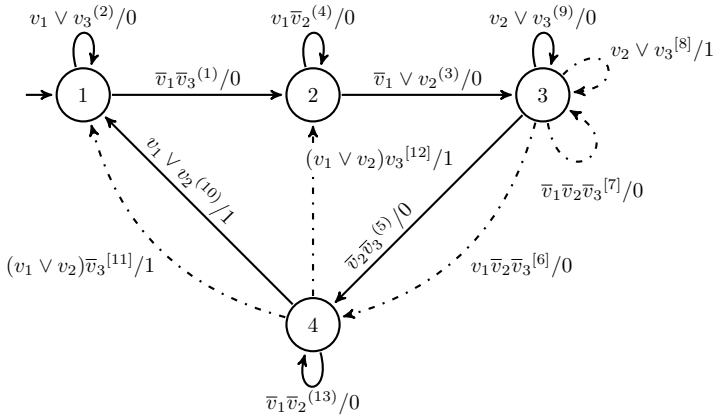

Fig. 1: A mutation SIFSM, state 1 is initial.

it is deterministic and the inputs of its transitions constitute a tautology, in other words, the transitions of the cluster have a complete system of guards so that each concrete input enables a transition. Let $Z(s)$ denote the set of all clusters of s. State s is said to be *suspicious* if $|Z(s)| > 1$. We use S_{susp} to denote the set of all suspicious states of $\mathcal{M}$.

In a mutation machine, untrusted transitions can be seen as the result of applying mutation operations transforming the specification into mutants. Mutation operations may also be considered as fault seeding in the specification. For an untrusted transition to belong to a mutant, it must participate in clusters. We say that a mutation machine is *well-formed* if each of its suspicious transitions belongs to a cluster. In what follows, we consider only well-formed mutation machines.

We assume that only completely specified deterministic submachines of $\mathcal{M}$ are possible implementation machines for the specification machine $\mathcal{S}$. The set of all such submachines is called a *fault domain* for $\mathcal{S}$, denoted *Sub*$(\mathcal{M})$. If $\mathcal{M}$ is deterministic then *Sub*$(\mathcal{M})$ contains just $\mathcal{S}$. Since each implementation machine in *Sub*$(\mathcal{M})$ is deterministic, each state of an implementation machine has only one cluster. The size of *Sub*$(\mathcal{M})$ is the product of the sizes of the clusters of the states, i.e., $|\textit{Sub}(\mathcal{M})| = \prod_{s \in S} |Z(s)|$. A DSIFSM $\mathcal{P} \in \textit{Sub}(\mathcal{M})$, such that $\mathcal{P} \neq \mathcal{S}$, is called a *mutant*. Each mutant $\mathcal{P}$ has all the trusted transitions of $\mathcal{M}$ and the set of suspicious transitions *Susp*$(\mathcal{P})$. It holds that for all $\mathcal{P}, \mathcal{P}' \in \textit{Sub}(\mathcal{M})$, if $\mathcal{P} \neq \mathcal{P}'$ then *Susp*$(\mathcal{P}) \neq$ *Susp*$(\mathcal{P}')$.

Fig. 1 presents an example of an NSIFSM which is a well-formed mutation machine with three Boolean input variables v_1, v_2 and v_3 and two outputs in $\{0, 1\}$. The mutation machine has five mutated transitions depicted with dashed lines. The solid lines represent the unaltered transitions of the specification machine. Identifiers of transitions are presented in brackets and parentheses for mutated and unaltered transition, respectively. There are eight suspicious transitions $t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}$ and t_{12}; three of them t_5, t_9 and t_{10} are untrusted. The states 3 and 4 are suspicious. The four clusters of state 3 are $\{t_5, t_9\}$, $\{t_5, t_8\}$, $\{t_6, t_7, t_9\}$ and $\{t_6, t_7, t_8\}$. The only two clusters

for state 4 are $\{t_{10}, t_{13}\}$ and $\{t_{11}, t_{12}, t_{13}\}$. The mutation machine includes seven mutants and the specification machine. Execution $t_1t_3t_7t_5$ is nondeterministic because it includes two nondeterministic transitions t_7 and t_5. Execution $t_1t_3t_7t_6$ is deterministic and involves four mutants determined with the following sets of suspicious transitions $\{t_9, t_7, t_6, t_{10}\}$, $\{t_9, t_7, t_6, t_{11}, t_{12}\}$, $\{t_8, t_7, t_6, t_{10}\}$ and $\{t_8, t_7, t_6, t_{11}, t_{12}\}$.

Let e be an execution of $\mathcal{M}$ and *Susp*(e) denote the set of suspicious transitions in e. We say that a (possibly nondeterministic or partially specified) submachine $\mathcal{P}$ is *involved* in e if *Susp*$(e) \subseteq$ *Susp*$(\mathcal{P})$. An execution of any submachine of $\mathcal{M}$ is an execution of $\mathcal{M}$, but only deterministic executions of $\mathcal{M}$ are executions of submachines in *Sub*$(\mathcal{M})$. $\mathcal{P} \in$ *Sub*$(\mathcal{M})$ is the only mutant involved in e if *Susp*$(e) =$ *Susp*$(\mathcal{P})$.

Since the specification and mutants are completely specified and deterministic SIFSM, we use equivalence as a conformance relation for testing. A mutant $\mathcal{P}$ is *nonconforming (faulty)* if $\mathcal{P} \not\simeq \mathcal{S}$, otherwise, it is called a *conforming* mutant. We say that a distinguishing input sequence $\alpha \in G^*$ such that $\mathcal{P} \not\simeq_\alpha \mathcal{S}$ *detects* or *kills* the mutant $\mathcal{P}$.

The tuple $\langle \mathcal{S}, \simeq, \textit{Sub}(\mathcal{M}) \rangle$ is a fault model following [19,20,21]. For a given specification machine $\mathcal{S}$ the equivalence partitions the set *Sub*$(\mathcal{M})$ into conforming implementations and nonconforming ones. In this paper, we do not require the DSIFSM $\mathcal{S}$ to be reduced, this implies that a conforming mutant may have fewer states than the specification $\mathcal{S}$; on the other hand, we assume that no fault creates new states in implementations, hence mutants with more states than the specification are not in *Sub*$(\mathcal{M})$.

2.3 Mutation Operations for Building Well-formed Mutation Machines

Mutation operations permit seeding different types of faults including output, transition and other types of faults which cannot be represented with classical FSM. Considering for instance nondeterministic Simulink/Stateflow models, priorities which are automatically assigned to transitions based on the graphical layout may vary upon changes in the layout [22]. The variation of the priorities causes transition faults which can be represented with mutated transitions. We consider mutation operations adding mutated transitions to well-formed mutation machines to build new well-formed mutation machines. Every mutated transition introduced by a mutation operation must belong to a cluster of a state. Let $\mathcal{M} = (S, s_0, V, O, T)$, $\mathcal{M}' = (S, s_0, V, O, T')$ be two well-formed mutation machines, $s \in S$ be a state, $A \subseteq T(s)$ be a subset of unaltered transitions from state s in $\mathcal{M}$ and $B \subseteq T'(s)$ be a subset of mutated transitions from state s in $\mathcal{M}'$. We say that $\mathcal{M}'$ is a mutation of $\mathcal{M}$ w.r.t A and B if the following four conditions hold: $A \cap B = \emptyset$, $T' = T \cup B$, the union of the inputs of the transitions in A is equivalent to the union of the inputs of the transitions in B and there are $t \in A$ and $t' \in B$ having compatible guards but different outputs or target states. We specify a mutation operation with a tuple $(\mathcal{M}, A, B)$ such that there exists a mutation of $\mathcal{M}$ w.r.t. A and B. The set B can be obtained from the transitions in A by changing target states or outputs, merging/splitting inputs of transitions, replacing variables with default values, swapping occurrences of variables in inputs, substituting a variable for another, modifying arithmetic/logical operations in guards. These operations introduce faults which cannot be represented in classical FSM; some of these faults are considered in [11,4,2]. Any well-formed mutation machine for a specification can be obtained by iterative application of mutation operations on the specification.

3 Boolean Expressions Specifying Mutants (un)Detected by Tests

Let $\langle \mathcal{S}, \simeq, \mathit{Sub}(\mathcal{M}) \rangle$ be a fault model. In the context of testing SIFSM, we consider that a test is just an input sequence. Tests detecting mutants can be determined using a distinguishing automaton obtained by composing the transitions of the specification and mutation machines as follows.

Definition 3. *Given a DSIFSM* $\mathcal{S} = (S, s_0, V, O, N)$ *and a mutation machine* $\mathcal{M} = (S, s_0, V, O, T)$ *of* $\mathcal{S}$*, a finite automaton* $\mathcal{D} = (D \cup \{\nabla\}, d_0, G, \Theta, \nabla)$*, where* $D \subseteq S \times S$*,* ∇ *is an accepting (sink) state and* $\Theta \subseteq D \times G \times D$ *is the transition relation is the* distinguishing automaton *for* $\mathcal{S}$ *and* $\mathcal{M}$*, if it holds that*

- $d_0 = (s_0, s_0)$ *is the initial state in* D
- *For any* $(s, t) \in D$
  - $((s, t), g \cap h, (s', t')) \in \Theta$*, if there exist* $(s, g, o, s') \in N$*,* $(t, h, o', t') \in T$*, such that* $o = o'$ *and* $g \cap h \neq \emptyset$
  - $((s, t), g \cap h, \nabla) \in \Theta$*, if there exist* $(s, g, o, s') \in N$*,* $(t, h, o', t') \in T$*, such that* $o \neq o'$ *and* $g \cap h \neq \emptyset$

Fig. 2 presents the distinguishing automaton for the mutation and specification machines in Fig. 1. Multiple transitions are represented with a single arc labeled with multiple inputs.

An execution of $\mathcal{D}$ starting at the initial state d_0 and ending at the sink state ∇ is said to be *accepted*. The language of $\mathcal{D}$, $L_{\mathcal{D}}$ is the set of tests labeling accepted executions of $\mathcal{D}$. Any nonconforming mutant in $\mathit{Sub}(\mathcal{M})$ can be detected by a test in $L_{\mathcal{D}}$.

Theorem 1. *Given the distinguishing automaton* $\mathcal{D}$ *for* $\mathcal{S}$ *and* $\mathcal{M}$*,* $\mathcal{P} \not\simeq \mathcal{S}$ *for some* $\mathcal{P} \in \mathit{Sub}(\mathcal{M})$ *if and only if* $\mathcal{P} \not\simeq_\alpha \mathcal{S}$ *for some* $\alpha \in L_{\mathcal{D}}$*.*

A test $\alpha \in L_{\mathcal{D}}$ triggers several executions in the distinguishing automaton defined by executions of the specification and mutation machine $\mathcal{M}$ which are the respective projections of the distinguishing automaton's executions; a deterministic execution of $\mathcal{M}$ defining an execution of the distinguishing automaton $\mathcal{D}$ to the sink state is called α*-revealing* if it is triggered by any prefix of the test α. An α-revealing execution may belong to several mutants. As discussed above, given a deterministic execution e of $\mathcal{M}$ which has the set of suspicious transitions $\mathit{Susp}(e)$, a mutant $\mathcal{P}$ is involved in the execution e, if $\mathit{Susp}(e) \subseteq \mathit{Susp}(\mathcal{P})$. Since an α-revealing execution defines an accepted execution of the distinguishing automaton, each involved mutant is killed. Thus the sets of suspicious transitions in all α-revealing executions represent all the mutants killed by test α; on the other hand, it does not detect mutants which are not involved in these executions. To elaborate a mutant killing test generation procedure we need first to determine all the sets of suspicious transitions of the revealing executions for a given test. Let E_α be the finite set of α-revealing executions of $\mathcal{M}$. We use Boolean expressions for encoding of suspicious transitions of executions in E_α. A solution of a Boolean expression c over a set of variables is an assignment to *True* or *False* of every variable which makes c *True*. A solution of c can be obtained with solvers [7,13] which return *null* in case c has no solution. Given the set of suspicious transitions $\mathit{Susp}(\mathcal{M})$,

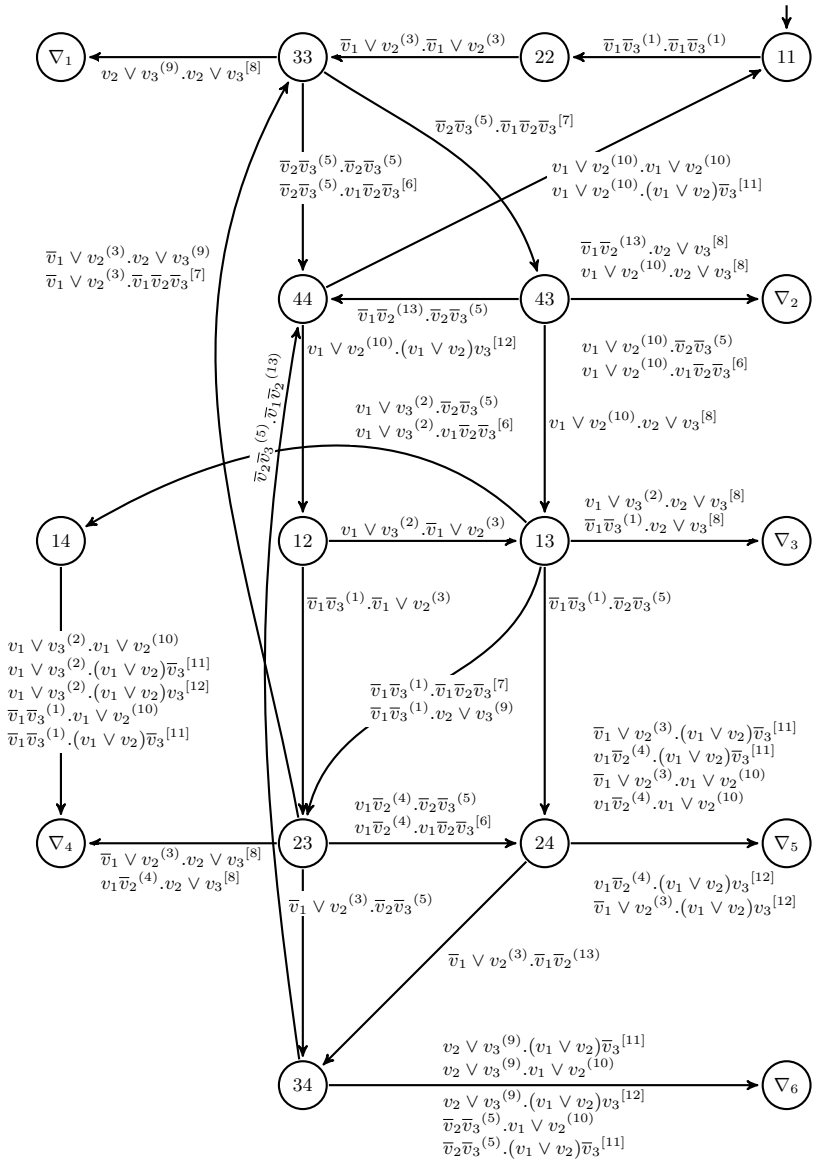

Fig. 2: The distinguishing automaton for machines in Fig. 1, state 11 is initial.

we introduce $|\mathit{Susp}(\mathcal{M})|$ Boolean variables each of which represents a suspicious transition of the mutation machine. From now on we will use t to refer to both a suspicious transition and the variable which represents it. Then the conjunction $c_e \stackrel{\text{def}}{=} \bigwedge_{t \in \mathit{Susp}(e)} t$ of variables of transitions in $\mathit{Susp}(e)$ specifies the submachines involved in the revealing execution e. Moreover, the disjunction of conjunctions of all executions in E_α gives

Boolean expression $c_\alpha \stackrel{\text{def}}{=} \bigvee_{e \in E_\alpha} c_e$ specifying all the submachines which are involved in all executions in E_α and killed by the test α. As usual, the disjunction over the empty set is *False* and the conjunction over the empty set is *True*. Boolean expression c_α is satisfiable whenever $E_\alpha \neq \emptyset$, since an α-revealing execution of $\mathcal{M}$ is a projection of the execution of the distinguishing machine. A witness solution of c_α provides all the variables evaluated to *True* and defines a corresponding subset of *Susp*$(\mathcal{M})$ which together with the trusted transitions of $\mathcal{M}$ determines (the transition relation of) a submachine of $\mathcal{M}$ involved in α-revealing executions.

Let $\alpha = (\overline{v}_1\overline{v}_2\overline{v}_3)(\overline{v}_1\overline{v}_2v_3)(\overline{v}_1\overline{v}_2\overline{v}_3)(v_1\overline{v}_2\overline{v}_3)(\overline{v}_1v_2\overline{v}_3)(v_1v_2v_3)$ be a test case. It triggers four executions in the distinguishing automaton in Fig. 2. These executions are defined by four executions of mutation machine in Fig. 1 including $e_1 = t_1t_3t_5t_{10}t_1t_3$, $e_2 = t_1t_3t_5t_{11}t_1t_3$, $e_3 = t_1t_3t_7t_5$ and $e_4 = t_1t_3t_7t_6$. The executions e_3 and e_4 are defined by the two executions to the sink state of the distinguishing automaton. Execution e_3 is not α-revealing because it is nondeterministic. Only execution e_4 is α-revealing and includes the two suspicious transitions in *Susp*$(e_4) = \{t_7, t_6\}$. Thus $c_{e_4} = t_7t_6$ and $c_\alpha = c_{e_4}$. The solutions of c_α determine the submachines involved in e_4.

We denote by *Generate_a_submachine*(c) a function which either determines such a submachine from a solution of c it obtained after calling a solver or returns *null* if c has no solution. Nondeterministic and partially specified submachines are not mutants. To exclude such submachines as well as the specification from any solution, clusters in suspicious states has to be considered.

Let s be a suspicious state, $Z(s) = \{Z_1, Z_2, \ldots, Z_n\}$ be the set of its clusters. Then the conjunction of variables of a cluster Z_i expresses the requirement that all these transitions must be present together to ensure that a submachine with the cluster Z_i is completely specified in state s. Moreover, since all mutants are deterministic, only one cluster in $Z(s)$ can be chosen, therefore, the transitions are restricted by the expressions determining clusters. Each cluster Z_i is exclusively determined by Boolean expression $z_i \stackrel{\text{def}}{=} (\bigwedge_{t \in Z_i} t) \wedge (\overline{\bigvee_{t \in \mathit{Susp}(s) \setminus Z_i} t})$ which permits the selection of all the suspicious transitions in Z_i and the exclusion of the remaining suspicious transitions leaving s, i.e., the exclusion of the other clusters.

Lemma 1. *Let $Z_i, Z_j \in Z(s)$ be two clusters of state s. Every solution of z_i is not a solution of z_j.*

Then each state s in S_{susp} yields the expression $c_s \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} z_i$ of which all the solutions determine all the clusters in $Z(s)$.

Lemma 2. *Every solution of c_s determines a cluster in $Z(s)$ and every cluster in $Z(s)$ is determined by a solution of c_s.*

Each solution of $\bigwedge_{s \in S_{\mathit{susp}}} c_s$ determines the set of clusters of suspicious states either in the specification or in a mutant. Each such cluster in the specification has at least one untrusted transition in *Untr*$(\mathcal{S})$. Excluding the specification can be expressed with the negation of the conjunction of the variables of all the untrusted transitions $\overline{\bigwedge_{t \in \mathit{Untr}(\mathcal{S})} t}$. Any of its solutions excludes at least one cluster in the specification and therefore cannot determine the specification. The Boolean expression $c_{\mathit{clstr}} \stackrel{\text{def}}{=} \bigwedge_{s \in S_{\mathit{susp}}} c_s \wedge \overline{\bigwedge_{t \in \mathit{Untr}(\mathcal{S})} t}$

Procedure Build_expression $(TS, \mathcal{D})$;
Input : *TS*, a test suite
Input : $\mathcal{D}$, the distinguishing automaton of mutation machine $\mathcal{M}$ and specification $\mathcal{S}$
Output : c_{TS}, a Boolean expression defining submachines of $\mathcal{M}$ involved in revealing executions for tests in *TS*

$c_{TS} := False$;
for *each* $\alpha \in TS$ **do**
  Using $\mathcal{D}$, determine E_α, the set of α-revealing executions of $\mathcal{M}$;
  $c_\alpha := False$;
  for *each* $e \in E_\alpha$ **do**
    $c_e := \bigwedge_{t \in Susp(e)} t$;
    $c_\alpha := c_\alpha \vee c_e$;
  end
  $c_{TS} := c_{TS} \vee c_\alpha$;
end
Return c_{TS};

Algorithm 1: Building c_{TS}

excludes nondeterministic and partially specified submachines and the specification, which means that c_{clstr} specifies only all mutants in the fault domain $Sub(\mathcal{M})$.

Considering the example mutation machine, we determine the Boolean expressions for the suspicious states 3 and 4. For the four clusters of state 3 $Z_{3_1} = \{t_5, t_9\}$, $Z_{3_2} = \{t_5, t_8\}$, $Z_{3_3} = \{t_6, t_7, t_9\}$ and $Z_{3_4} = \{t_6, t_7, t_8\}$ we build Boolean expressions $z_{3_1} = t_5 t_9(\overline{t_6 t_7 t_8})$, $z_{3_2} = t_5 t_8(\overline{t_6 t_7 t_9})$, $z_{3_3} = t_6 t_7 t_9(\overline{t_5 t_8})$ and $z_{3_4} = t_6 t_7 t_8(\overline{t_5 t_9})$. Then $c_3 = (z_{3_1} \vee z_{3_2} \vee z_{3_3} \vee z_{3_4})$. Similarly for state 4, we build $c_4 = (z_{4_1} \vee z_{4_2})$ where $z_{4_1} = t_{10} t_{13}(\overline{t_{11} t_{12}})$ and $z_{4_2} = t_{11} t_{12} t_{10} \overline{t_{13}}$. Finally, $c_{clstr} = c_3 \wedge c_4 \wedge (\bar{t}_5 \vee \bar{t}_9 \vee \bar{t}_{10})$.

A solution of $c_\alpha \wedge c_{clstr}$ defines a subset of $Susp(\mathcal{M})$ which together with the trusted transitions of $\mathcal{M}$ determines (a transition relation of) a mutant detected by α. All solutions thus determine all mutants detected by the test α. For a non-trivial mutation machine, a sheer number of killed mutants makes their enumeration impracticable. Hence, instead of determining killed mutants, we determine a (conforming or nonconforming) mutant which survives the test α. The *negation* of c_α, $\overline{c}_\alpha$ determines the transition relations of not only all mutants which survive α but also other submachines which are not mutants. Considering the running example, a partially specified submachine having the suspicious transitions t_9 and t_8 is determined by the solution of $\overline{c_\alpha}$ which assigns *True* t_9 and t_8; such a submachine is not a mutant and it does not belong to $Sub(\mathcal{M})$. To eliminate them as well as the specification, we use c_{clstr} as before. Finally, each mutant which survives test α is determined by a solution of the expression $\overline{c}_\alpha \wedge c_{clstr}$.

Theorem 2. *Test $\alpha \in G^*$ does not detect a mutant $\mathcal{P}$ if and only if there is a solution of $\overline{c_\alpha} \wedge c_{clstr}$ which determines $\mathcal{P}$.*

4 Checking Completeness of a Test Suite

Given a fault model $\langle \mathcal{S}, \simeq, \mathcal{M} \rangle$, a *fault subdomain* for $\mathcal{S}$, *FD* is a subset of $Sub(\mathcal{M})$. A *test suite*, *TS* is a set of tests. *TS* is *complete* for fault subdomain *FD* if it detects all the

nonconforming mutants in *FD*. Let us define $c_{TS} \stackrel{\text{def}}{=} \bigvee_{\alpha \in TS} c_\alpha$, a Boolean expression which determines the submachines involved in revealing executions for the tests in *TS*. Procedure *Build_expression* for building c_{TS} is presented in Algorithm 1.

Let c_{fd} be a Boolean expression specifying only all mutants in a fault subdomain *FD*. It can be formulated as the conjunction c_{clstr} with another (possibly always True) Boolean expression over the variables of suspicious transitions, which excludes mutants from $Sub(\mathcal{M})$ to obtain *FD*. A fault subdomain can always be refined with an expression specifying the mutants to be excluded. Later, in checking the completeness of a test suite for a given *FD*, we will be excluding conforming mutants.

Theorem 3. *Test suite TS is complete for fault subdomain FD if and only if* $\overline{c_{TS}} \wedge c_{fd}$ *has no solution or each of its solutions determines a conforming mutant.*

The fault domain $Sub(\mathcal{M})$ is specified with c_{clstr}, which leads to Corollary 1.

Corollary 1. *Test suite TS is complete for* $Sub(\mathcal{M})$ *if and only if* $\overline{c_{TS}} \wedge c_{clstr}$ *has no solution or each of its solutions determines a conforming mutant.*

Based on Theorem 3, checking the completeness of a test suite for a fault subdomain *FD* amounts to its iterative refinement by excluding conforming mutants as solutions to $\overline{c_{TS}} \wedge c_{fd}$ while no nonconforming mutant is found. In particular, the negation of the conjunction of variables of all suspicious transitions of a conforming mutant added to c_{fd} excludes it from *FD*. This method is formalized in Algorithm 2 which presents Procedure *Check_completeness* for checking the completeness of a test suite *TS* for a fault subdomain specified by the input parameter c_{fd} which is refined each time a conforming mutant is generated. The procedure *Check_completeness* also takes as inputs a test suite *TS* and the distinguishing automaton for the mutation and specification machines. It returns a witness test detecting a mutant surviving *TS* in case *TS* is not complete; otherwise the witness test is empty, which indicates that *TS* is complete. It also returns an updated expression of c_{fd} specifying a reduced fault domain which is used to generate tests that make *TS* a complete test suite in Section 5. Procedure *Check_completeness* proceeds as follows. It calls *Build_expression* for building c_{TS}, the Boolean expression which determines the submachines involved in revealing executions for tests in *TS*. Initialy, the fault domain is specified with the conjunction of c_{fd} with the negation of c_{TS} which determines all mutants surviving *TS*. The execution is iterative and each step consists in generating a mutant surviving *TS*, checking the conformance of the mutant and removing from the current fault domain the mutant in case it is conforming.

Procedure *Check_completeness* makes calls to *Generate_a_submachine* to select a mutant in a fault domain specified with Boolean expression c_{fd}. *Generate_a_submachine* returns *null* in case the fault domain is empty. The execution of *Check_completeness* stops when *Generate_a_submachine* returns a nonconforming mutant or *null*. In case *null* is returned, the test suite is declared complete and *Check_completeness* returns the empty test; otherwise the test suite is declared incomplete and *Check_completeness* returns a non empty witness test detecting a nonconforming mutant. In both cases *Check_completeness* returns an expression specifying the reduced fault domain at the end of the execution. In the next section, we will check the completeness of generated tests (e.g., the witness tests) for the reduced fault domains in determining complete test suites for fault domains specified with mutation machines.

Procedure Check_completeness $(c_{fd}, TS, \mathcal{D})$;
Input/Output : c_{fd} a boolean expression specifying a fault domain
Input **:** *TS*, a (possibly empty) test suite
Input **:** $\mathcal{D}$, the distinguishing automaton of $\mathcal{M}$ and $\mathcal{S}$
Output **:** $\alpha \neq \varepsilon$, a test case revealing a nonconforming mutant surviving the test suite; $\alpha = \varepsilon$, if *TS* is complete

$c_{TS} :=$ *Build_expression*$(TS, \mathcal{D})$;
$c_{fd} := \overline{c_{TS}} \wedge c_{fd}$;
$c_{\mathcal{P}} :=$ *False*;
$\alpha := \varepsilon$;
repeat
  $c_{fd} := c_{fd} \wedge \overline{c_{\mathcal{P}}}$;
  $\mathcal{P} :=$ *Generate_a_submachine*(c_{fd}) ;
  if $\mathcal{P} \neq$ *null* **then**
    Build $\mathcal{D}_{\mathcal{P}}$, the distinguishing automaton of $\mathcal{S}$ and $\mathcal{P}$;
    if $\mathcal{D}_{\mathcal{P}}$ *has no sink state* **then**
      $c_{\mathcal{P}} := \bigwedge_{t \in Susp(\mathcal{P})} t$;
    else
      Set α to an input sequence in $L_{\mathcal{D}_{\mathcal{P}}}$;
    end
  end
until $\alpha \neq \varepsilon$ *or* $\mathcal{P} =$ *null*;
return (c_{fd}, α)

Algorithm 2: Checking the completeness of a test suite for a fault domain

In checking the completeness of the initial test suite $\{\alpha\}$ for the example mutation machine and test α, *Check_completeness* takes as input $c_{fd} = c_{clstr}$, $TS = \{\alpha\}$ and the distinguishing automaton in Fig. 2. Then, it determines $c_{TS} = c_\alpha = t_7 t_6$, sets $c_{fd} = \overline{c_{TS}} \wedge c_{clstr}$, $c_{\mathcal{P}} =$ *False* and $\alpha = \varepsilon$ and starts executing the loop. In the first iteration, the call of *Generate_a_submachine* with input c_{fd} has generated the mutant with the suspicious transitions t_8, t_{11}, t_{12}. The mutant is nonconforming and killed by the test $\beta = (\overline{v}_1 \overline{v}_3)(\overline{v}_1 \vee v_2)(v_2 \vee v_3)$ labeling a path to the sink state in the distinguishing automaton for the mutant. Then the execution of *Check_completeness* terminates with outputs c_{fd} and non empty test β, which indicates that the test suite $\{\alpha\}$ is not complete.

5 Complete Test Suite Generation

In case an initial (possibly empty) test suite does not detect all the nonconforming mutants in a fault domain, we want to generate tests which together with the initial tests constitute a complete test suite for the fault domain. This can be done iteratively by adding a new test detecting a nonconforming mutant surviving the incomplete test suite, obtaining a new test suite which in turn can be augmented in case it is not complete. This complete test suite generation method is formalized in Algorithm 3 with procedure *Complete_test_gen* which takes as inputs an initial test suite TS_{init} and a fault domain represented with a mutation machine. At every step, the procedure adds a current test

Procedure Complete_test_gen (TS_{init}, $\langle \mathcal{S}, \simeq, Sub(\mathcal{M}) \rangle$);
Input : TS_{init}, an initial (possibly empty) test suite
Input : $\langle \mathcal{S}, \simeq, Sub(\mathcal{M}) \rangle$, a fault model
Output : *TS*, a complete test suite for $\langle \mathcal{S}, \simeq, \mathcal{M} \rangle$
Compute c_{clstr}, the boolean expression which determines all mutants in $Sub(\mathcal{M})$;
Compute $\mathcal{D}$ the distinguishing automaton for $\mathcal{S}$ and $\mathcal{M}$;
$c_{fd} := c_{clstr}$;
$TS := \emptyset$;
$TS_{curr} := TS_{init}$;
repeat
  $TS := TS \cup TS_{curr}$;
  (c_{fd}, α) := *Check_completeness*(c_{fd}, TS_{curr}, $\mathcal{D}$);
  $TS_{curr} := \{\alpha\}$;
until $(\alpha = \varepsilon)$;
return *TS* is complete;

Algorithm 3: Generation of a complete test suite from initial test suite TS_{init}

suite to the set *TS* of already analyzed tests and makes a call of *Check_completeness* to analyze the completeness of a current test suite w.r.t. a current fault domain. In case of completeness, *Check_completeness* returns the empty test, which triggers the termination of *Complete_test_gen* with *TS* as a complete test suite for the initial fault domain; otherwise, *Check_completeness* returns a witness test detecting a nonconforming mutant and a reduced fault domain obtained by removing the nonconforming mutant and possibly other conforming mutants, as discussed in the previous section. Then *Complete_test_gen* proceeds to a next iteration step after it has set the current fault domain and the current test suite to the reduced fault domain and the witness test.

Theorem 4. *Procedure* Complete_test_gen *always terminates and returns a complete test suite for the fault domain specified with a fault model.*

Procedure *Complete_test_gen* always terminates because the execution of its only loop always terminates. This is because the initial fault domain consisting of a finite number of mutants is reduced at every iteration step of the loop and *Check_completeness* returns the empty test when executed with the empty fault domain as an input.

Considering the running example, Table 1 summarizes data computed in executing *Complete_test_gen* to generate a complete test suite from initial test suite $TS_{init} = \{\alpha\}$. The iteration step appears at the first column. Data are initialized at the end of step *init*. In each step *Complete_test_gen* makes a call to *Check_completeness* which computes the executions revealed by TS_{curr} determined in the previous step and updates c_{fd}. Three iteration steps were sufficient to obtain the complete test suite $\{\alpha, \beta, \gamma\}$ having three tests for the detection of the seven nonconforming mutants, which shows that the method permits generating fewer tests than the nonconforming mutants. Notice that we generate symbolic tests; their concrete instances should be used to execute against black box implementations. In our working example, for simplicity, all input variables are Boolean, which however, can represent comparisons of integer variables with some constants. The obtained concrete tests could then be just rewritten by replacing every Boolean variable by an instance of the corresponding comparison.

Table 1: Execution of Procedure *Complete_test_gen* with the initial test α.

	In *Check_completeness*				End of the step	
step	Revealing. Exec	$c_{TS_{curr}}$	c_{fd}	Surv. mut.	*TS*	TS_{curr}
init	N/A	N/A	c_{clstr}	N/A	$\emptyset$	α
1	$t_1t_3t_7t_6$	t_7t_6	$c_{fd} \wedge \overline{t_7t_6}$	t_8, t_{11}, t_{12}	α	β
2	$t_1t_3t_8$	t_8	$c_{fd} \wedge \overline{t_8}$	t_9, t_{11}, t_{12}	α, β	γ
3	$t_1t_3t_5t_{12}t_3t_5t_{10}$, $t_1t_3t_5t_{10}t_1t_3t_8$	$(t_5t_{11}t_{12}) \vee (t_5t_8t_{10})$	$c_{fd} \wedge \overline{(t_5t_{11}t_{12}) \vee (t_5t_8t_{10})}$	$\emptyset$	α, β, γ	ε

$\alpha = (\overline{v}_1\overline{v}_2\overline{v}_3)(\overline{v}_1\overline{v}_2v_3)(\overline{v}_1\overline{v}_2\overline{v}_3)(v_1\overline{v}_2\overline{v}_3)(\overline{v}_1v_2\overline{v}_3)(v_1v_2v_3)$
$\beta = (\overline{v}_1\overline{v}_3)(\overline{v}_1 \vee v_2)(v_2 \vee v_3)$
$\gamma = (\overline{v}_1\overline{v}_3)(\overline{v}_1 \vee v_2)(\overline{v}_2\overline{v}_3)(v_1v_3 \vee v_2v_3)(\overline{v}_1\overline{v}_3)(\overline{v}_1\overline{v}_2\overline{v}_3)(v_2\overline{v}_3)$

Table 2: Experimental results with the prototype tool

#Mutants	8191	163839	1105919	9400319
#Tests	14	15	18	18
Time (sec.)	30	90	100	296

6 Prototype Tool and Experimental Results

We implemented in JAVA a prototype tool consisting of three main modules. The first module for parsing mutation machines in text format was developed using ANTLR 4.1 [15]. The second module is concerned with building clusters, distinguishing automata and Boolean expressions for undetected mutants; it uses as a back-end the solver Z3 [13] for solving of non Boolean expressions obtained by combining predicates in building clusters and automata. We integrated the solver in the tool using a Z3 API. The third module is responsible of solving Boolean expressions for mutants, extracting mutants and generating new tests. The module also uses solver Z3 though it may also use a SAT solver [7] since it deals with the resolution of Boolean expressions only.

In our experiments, we use a desktop computer with the following settings: 3.4Ghz Intel Core i7-3770 CPU, 16.0 GB of memory (RAM), Windows 7 (64 bits).

We use the prototype on an industrial-like SIFSM model obtained by transforming a Simulink/Stateflow model [18] of an automotive controller. To regulate the air quality in a vehicle, the controller sets an air source position to 0 or 1 depending on its current state and truth values of predicates on integer and Boolean input variables. The transformation required flattening and determinizing the original hierarchical Simulink/Stateflow model. The determinization is based on priorities assigned to nondeterministic transitions as it is done by Simulink [22]. We obtained a SIFSM with 13 states, 62 transitions and 22 input variables. Then we have manually introduced faults (transition faults, output faults, swapping of variables, replacing variables with constants), obtaining a mutation machine with $2^{13} - 1 = 8191$ mutants. Our tool generates, within 30 seconds, a complete test suite with 14 tests detecting the mutants. Finally, we generate complete tests from automatically generated mutation machines with a generator executing randomly selected mutation operations. Table 2 presents the numbers of mutants in the mutation machines, the number of tests in the generated complete test suites. The

maximal length of the tests is 8. We observed that the test generation is fast when the mutation operations introduce a small number of nonconforming mutants, which is a realistic assumption [11] for applying our method.

7 Conclusion

We lifted the multiple mutation testing approach developed for classical (Mealy) FSM to symbolic input finite state machine (SIFSM). SIFSM extends classical FSM with predicates defined over input variables with possibly infinite domains.

We defined well-formed mutation machines for SIFSM as a fault model for compact representation of a fault domain consisting of several faulty implementations (mutants) of a specification machine. Then we defined mutation operations for building well-formed mutation machines. Based on the machine equivalence and distinguishability relations, we have defined tests detecting nonconforming mutants and developed a multiple mutation testing approach from SIFSM. The proposed approach leveraging on that developed for classical FSM includes a method for checking the completeness of test suites, i.e., their adequacy to detect all nonconforming mutants in a fault domain, and a method for complete test suite generation avoiding mutant enumeration. The novelty of the proposed approach is that it can analyze and enhance completeness of symbolic tests w.r.t. user defined fault models for a specification with infinite input domains.

The experiments with a prototype tool we have developed indicate that our methods can be applied to industrial-like models of systems.

Our current work focuses on extending the approach to FSM with outputs determined by arithmetic operations over input and output variables [17], to FSM extended with timing predicates [1,14] and to C program.

Acknowledgements. This work is supported in part by GM, NSERC of Canada and MESI (Ministère de l'Économie, Science et Innovation) of Gouvernement du Québec.

References

1. Batth, S.S., Vieira, E.R., Cavalli, A., Uyar, M.U.: Specification of timed efsm fault models in sdl. In: Proceedings of the 27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems. pp. 50–65. Springer-Verlag (2007)
2. Bessayah, F., Cavalli, A., Maja, W., Martins, E., Valenti, A.W.: A fault injection tool for testing web services composition. In: Proceedings of 5th International Academic and Industrial Conference on Testing – Practice and Research Techniques. pp. 137–146. Springer Berlin Heidelberg (2010)
3. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: 30th ACM/IEEE Design Automation Conference. pp. 86–91 (1993)
4. Delamaro, M.E., Maldonado, J.C., Pasquini, A., Mathur, A.P.: Interface mutation test adequacy criterion: An empirical evaluation. Empir. Softw. Eng. 6(2), 111–142 (2001)
5. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer 11(4), 34–41 (Apr 1978)

6. D'silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(7), 1165–1178 (2008)
7. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proceedings of 7th International Conference on Theory and Applications of Satisfiability Testing. Lecture Notes in Computer Science, vol. 2919, pp. 333–336. Springer (2004)
8. El-Fakih, K., Kolomeez, A., Prokopenko, S., Yevtushenko, N.: Extended finite state machine based test derivation driven by user defined faults. In: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation. pp. 308–317 (2008)
9. El-Fakih, K., Yevtushenko, N., Bozga, M., Bensalem, S.: Distinguishing extended finite state machine configurations using predicate abstraction. Journal of Software Engineering Research and Development 4(1), 1 (2016)
10. Huang, W.l., Peleska, J.: Exhaustive model-based equivalence class testing. In: Proceedings of the 25th IFIP WG 6.1 International Conference on Testing Software and Systems. pp. 49–64. Springer Berlin Heidelberg (2013)
11. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37(5), 649–678 (sep 2011)
12. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293–303 (2009)
13. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg (2008)
14. Nguena Timo, O., Rollet, A.: Conformance testing of variable driven automata. In: Proceedings of 8th IEEE International Workshop on Factory Communication Systems. pp. 241–248 (2010)
15. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2nd edn. (2013)
16. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. Int. J. Softw. Tools Technol. Transf. 11(4), 339–353 (2009)
17. Petrenko, A.: Checking experiments for symbolic input/output finite state machines. In: Workshops Proceedings of 9th International Conference on Software Testing, Verification and Validation. pp. 229–237 (2016)
18. Petrenko, A., Dury, A., Ramesh, S., Mohalik, S.: A method and tool for test optimization for automotive controllers. In: Workshops Proceedings of 6th IEEE International Conference on Software Testing, Verification and Validation. pp. 198–207 (2013)
19. Petrenko, A., Nguena Timo, O., Ramesh, S.: Multiple mutation testing from fsm. In: Proceedings of the 6th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems. pp. 222–238. Springer International Publishing (2016)
20. Petrenko, A., Nguena Timo, O., Ramesh, S.: Test generation by constraint solving and fsm mutant killing. In: Proceedings 28th IFIP WG 6.1 International Conference on Testing Software and Systems. pp. 36–51. Springer International Publishing (2016)
21. Petrenko, A., Simao, A.: Checking experiments for finite state machines with symbolic inputs. In: Proceedings of 27th IFIP WG 6.1 International Conference on Testing Software and Systems. pp. 3–18. Springer International Publishing (2015)
22. Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F.: Defining and translating a safe subset of simulink/stateflow into lustre. In: Proceedings of the 4th ACM international conference on Embedded software. pp. 259–268. ACM (2004)
23. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability 22(5), 297–312 (2012)