Software Quality Journal manuscript No.
(will be inserted by the editor)

Fault Model Driven Testing from FSM with Symbolic Inputs

Omer Nguena Timo · Alexandre Petrenko · S. Ramesh

Received: date / Accepted: date

Abstract Test generation based on one-by-one analysis of potential implementations in fault models is challenging; it is indeed impossible or inefficient to enumerate each and every implementation, even when a fault model defines a finite but a significant number of implementations. We propose an approach for fault model and constraint solving-based testing from a particular type of extended finite state machines called a symbolic input finite state machine (SIFSM). Transitions in SIFSMs are labeled with symbolic inputs, which are predicates on input variables having possibly infinite domains. Its implementations, mutants, are also represented by SIFSMs. The generated tests are complete in a given fault domain which is a set of mutants specified with a so-called mutation machine. We define a well-formed mutation SIFSM for describing various types of faults. Given a mutation SIFSM, we develop methods for evaluating the adequacy of a test suite and generating complete tests. Experimental results with the prototype tool we have developed indicate that the approach is applicable to industrial-like systems.

Keywords Extended FSM · Conformance testing · Mutation testing · Fault model-based test generation · Constraint solving

Omer Nguena Timo
Computer Research Institute of Montreal - CRIM
405 Avenue Ogilvy Suite 101, H3N 1M3 (QC), Montreal, Canada
Tel.: +1-514-840-1234
E-mail: omer.nguena-timo@crim.ca

Alexandre Petrenko
Computer Research Institute of Montreal, CRIM
405 Avenue Ogilvy Suite 101, H3N 1M3 (QC), Montreal, Canada
E-mail: petrenko@crim.ca

S. Ramesh
GM Global R&D - Warren, MI, USA
E-mail: ramesh.s@gm.com

1 Introduction

Detecting implementation errors is a major challenge during the design and the maintenance of systems, which motivates research in testing techniques [21,33], model-checking [8] and runtime verification techniques [16]. Testing techniques not only aim at exercising a system with tests to reveal failures and eventually identify and repair faults causing the failures. They also aim at evaluating the adequacy of tests and their generation to cover artifacts that can conceal faults [7,15,6,3], e.g., statements, branches, interfaces. Mutants which are versions of a specification of a system seeded with suspected faults can be used to generate tests or to determine the adequacy of given tests to reveal the faults. A fault domain [23,30] can be specified with a set of mutants and tests detecting the mutants which do not conform to the specification can be applied to detect faulty implementations of a system.

The classical Mealy FSM model is often used in developing fault model-based testing approaches for detecting faulty implementations [23]. While the Mealy FSM is defined only over finite sets, many real discrete systems have inputs with infinite domains, this renders FSM-based testing approaches inapplicable. As an example, in automotive applications, the behaviors of some controllers [26] depend on the truth values of predicates defined over input variables with infinite domains. Extensions of FSMs with arithmetic operations on variables [24,29,5,19] and other formal notations [12,1,32,22,25,10,2] have been proposed to relax limitations of the classical FSM and used in developing testing methods [12,32,4,1,22,29,11,19,25,10].

We propose to generate tests complete for predefined fault models for specifications of systems represented with deterministic FSM with symbolic inputs (SIFSM), i.e, tests detecting all nonconforming SIFSM implementations modeled in a finite fault domain. SIFSM [29] 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 an SIFSM can be found in [12,14,26,13]. A fault domain specifies a finite set of SIFSM mutants modeling (possibly infinite) number of deterministic implementations of a deterministic system; it is represented with a nondeterministic SIFSM called a mutation machine. Mutants can have more states than the specification up to the number of states of the mutation machine. Mutated transitions of the mutation machine can be seen as incorrect parts of the description of a processing step in a system [30]. The mutation machines can be obtained by using mutation operators used in mutation-based testing [15,3,6], e.g., for introducing faults related to HW/SW integration [30,14]. Our fault model-based testing approach for SIFSM is new; however some contributions address problems similar to that of the paper.

Related work include testing approaches from extended FSM and fault models. The work in [25] is aimed at generating distinguishing tests for verifying configurations of an EFSM model, which is a different purpose of that of our work. The work in [10] also uses an EFSM model and a fault model with 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 user 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 [4], a fault domain specifies a (possibly infinite) set of CSP processes refining a reference CSP process; an infinite number of tests can be needed to cover fault domains. In our work, implementation models are obtained by replacing transitions in a reference model with mutated transitions. The contribution in [32] is a gray-box testing approach to detect all unexpected behaviors in implementations. The unexpected behaviors are paths in a nondeterministic symbolic reactive state machines (SRSM) representing the fault domain. SRSM is a kind of SIFSM extended with symbolic outputs; it allows specifying infinite output domains. In our work, we focus not on all unexpected behaviors, but on detecting nonconforming mutants which requires to find just one incorrect behavior to kill a mutant. In [22,14], specifications and implementations of systems are represented with state transition systems. In [22,14] systems are deterministic whereas they are nondeterministic in [12]. The fault model includes a fault domain defined over input equivalence classes for potential implementations. The test generation approach consists in building an abstract FSM and using the W-method to generate tests. Differently from that work, we use a mutation machine to specify a fault domain and not input equivalence classes, this allow us to model fine-grained mutations modeling faults that can hardly be represented in the fault model of [22,14]. For example, a mutation machine can have mutated transitions guarded by predicates splitting some original guard and leading not to all but only to certain target states. Our test generation method need neither equivalent input classes, nor intermediate abstract FSM. It further develops the constraint solving-based testing approach [27,28] that is different from traditional checking experiments, such as the W-method.

The contribution of the paper is three-fold. First, we define well-formed mutation machines specifying implementations in finite fault domains. We present mutation operators which can be applied to seed faults in systems. These operators have some restrictions concerning predicate mutations because predicates cannot be mutated independently. The mutation should not violate the input-completeness property of the mutants. Possible faults must be specified with well-formed mutation machines. The requirement of well-formedness of mutation machines reflects a fundamental assumption used in black box testing of deterministic FSMs that each implementation machine can be modeled by a completely specified deterministic machine. 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 represented by a mutation machine. Finally we propose a method for generating complete test suites. Following the ideas in our previous work on FSM-based testing [27,28], the methods rely on building and resolving constraints specifying the mutants undetected by given tests. However, in this work the constraints differ from those in our previous work; they are represented with Boolean expressions for determining both undetected mutants and the input-completeness property of the mutants. We evaluate the methods with a prototype tool applied to an SIFSM model of a component from the automotive domain.

The paper is an extended and enhanced version of our work in [18]. We provide the proofs of statements. We elaborate an algorithm used in our approach for

determining executions of mutation machines. We update the working example to illustrate symbolic inputs over variables from the integer domain. We perform new experiments on an updated version of the case study with integer and Boolean input variables, highlighting the scalability and the applicability of our approach. The remaining of the paper is organized as follows. Section 2 introduces mutation SIFSM and mutation operations used for its creation. In Section 4 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 5.1. In Section 5.2 we develop a method for complete test suite generation. Section 6 reports some experimental evaluation of the approach. We conclude 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 procedure exists, excluding the predicates that are always *false*. G^* denotes the universe of *input sequences* and ε denotes the empty sequence. Henceforth, an input sequence is called a *test*. 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^*.

Definition 1 (Compatibility, reduction and instance)
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.

Example 1 (Symbolic and concrete inputs)
Let v_1, v_2 be two integer variables and v_3 be a Boolean variable. Let $g_1 = v_1 \geq 4v_2 + 6 \wedge \overline{v_3}$, $g_2 = v_1 \geq 17 \wedge v_2 \leq 2 \wedge \overline{v_3}$ and $g_3 = v_1 \leq 10 \wedge v_2 \leq 1 \wedge \overline{v_3}$ be three predicates over the three variables. Each predicate defines a symbolic input and has more than one instances. The predicate g_2 is a reduction of g_1 while g_3 is not.

We consider an extension of FSM called symbolic input finite state machine (SIFSM) [29], 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.

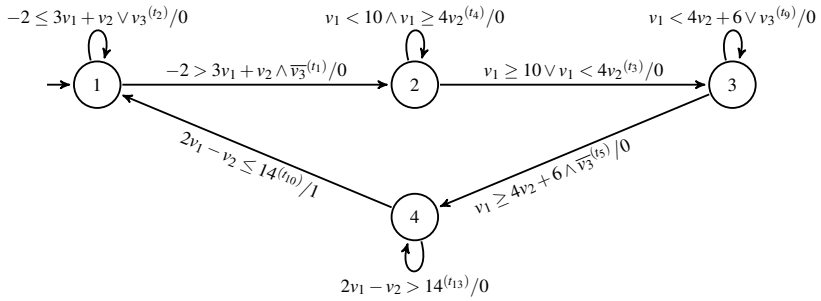

Fig. 1 A specification SIFSM, state 1 is initial. The name assigned to each transition appears in parenthesis as a subscript of the input located before the symbol "/". The outputs are located after the symbol "/".

Definition 2 (Symbolic input finite state machine)
A *symbolic input finite state machine* $\mathcal{S}$ (SIFSM 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.

Definition 3 (Executions of an SIFSM and triggering of executions) An *execution* of $\mathcal{S}$ from state s is a sequence $e = t_1 t_2 \ldots t_n$ of transitions $(t_i = (s_i, g_i, o_i, s_{i+1}))_{i=1..n}$ forming a path from s in the state transition diagram of $\mathcal{S}$. We use $inp(e)$, $out(e)$ and $tgt(e)$ to denote the input sequence $g_1 g_2 \ldots g_n$, the output sequence $o_1 o_2 \ldots o_n$ and the target s_{n+1} of e, respectively. We say that an input sequence α *triggers* e if α is a reduction of the input of the execution e, i.e., $\alpha \subseteq inp(e)$.

We let $out_{\mathcal{S}}(s, \alpha) = \{out(e) \mid e \text{ is an execution of } \mathcal{S} \text{ in s and } \alpha \subseteq inp(e)\}$ denote the set of the output sequences which can be produced by $\mathcal{S}$ in response to input sequence α at state s.

We let $\Omega(s)$ denote the set of all symbolic input sequences defined in state s, i.e., $g_1 g_2 \ldots g_n \in \Omega(s)$ if there is an execution e from s such that $g_1 g_2 \ldots g_n = inp(e)$. We denote by $\Omega_{\mathcal{S}}$ the set of input sequences defined in the initial state of $\mathcal{S}$.

Transitions from the same state with compatible inputs are said to be *compatible*.

Definition 4 (Deterministic and nondeterministic executions) An execution with compatible transitions is called *nondeterministic*, otherwise it is *deterministic*.

Definition 5 (Deterministic, completely specified and initially connected SIFSM) The machine $\mathcal{S}$ is *deterministic* (DSIFSM), if for every state s, $T(s)$ does not have compatible transitions; otherwise $\mathcal{S}$ is a *nondeterministic* SIFSM (NSIFSM). The machine $\mathcal{S}$ is *completely specified*, if for each state s, the disjunction over all predicates from $G(s)$ is a tautology. The machine $\mathcal{S}$ is *initially connected*, if for any state $s \in S$, there exists an execution from s_0 to s.

Clearly, a DSIFSM has only deterministic executions, while an NSIFSM can have both. Only executions corresponding to cyclic paths in the NSIFSM can be nondeterministic; the executions corresponding to acyclic paths are deterministic.

Example 2 (An example of an SIFSM)
Fig. 1 presents the state transition diagram of an example of SIFSM. The SIFSM has three input variables v_1, v_2 and v_3 and two outputs in $\{0,1\}$. The input variables v_1 and v_2 are Integers and v_3 is a Boolean input variable. Each transition is represented with a labeled arc. The name assigned to a transition appears in parenthesis as a subscript of the input located before the symbol "/"; the output is located after the symbol "/". The machine is deterministic, completely specified and initially connected.

Henceforth, all SIFSMs are initially connected and completely specified.

We define distinguishability and equivalence relations between states of SIFSM. Intuitively, states that produce different output sequences in response to some concrete input sequence are distinguishable.

Definition 6 (Distinguishability and equivalence relations)
Let p be a state of an SIFSM $\mathcal{P}$ and s be a state of an SIFSM $\mathcal{S}$ over the same set of input variables V. Given an input sequence $\alpha \subseteq \alpha_1 \cap \alpha_2$, such that $\alpha_1 \in \Omega(p)$ and $\alpha_2 \in \Omega(s)$, p and s are *distinguishable* (with input sequence α), denoted $p \not\simeq_\alpha s$, if $out(p,\alpha) \neq out(s,\alpha)$, otherwise they are *equivalent*, i.e., if $out(p,\alpha) = out(s,\alpha)$ for all $\alpha \subseteq \alpha_1 \cap \alpha_2$, $\alpha_1 \in \Omega(p)$, and $\alpha_2 \in \Omega(s)$. An SIFSM $\mathcal{S}$ is *reduced* if its states are pairwise distinguishable.

The relation $\simeq$ serves as a conformance relation between SIFSMs.

Definition 7 (Submachine of an SIFSM)
Given SIFSMs $\mathcal{M} = (M, m_0, V, O, T)$ and $\mathcal{P} = (P, p_0, V, O, E)$, $\mathcal{P}$ is a *submachine* of $\mathcal{M}$ if $P \subseteq M$, $p_0 = m_0$ and $E \subseteq T$.

3 Fault Modeling with Mutation Machine

3.1 Mutation Machine

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

Definition 8 (Specification and mutation machine)
A NSIFSM $\mathcal{M} = (M, m_0, V, O, T)$ is a *mutation machine* of $\mathcal{S}$, if $\mathcal{S}$ is a submachine of $\mathcal{M}$. Then $\mathcal{S}$ is called the *specification* machine for $\mathcal{M}$.

We use mutation machines to compactly represent possible implementations of specifications. Any completely specified deterministic submachine of a mutation machine, except the specification, is called a *mutant*. A mutant is obtained by seeding the specification with faults. Seeding the specification with faults consists in replacing transitions in the specification by compatible transitions representing the faults. Each mutant models a (potentially infinite) set of implementation SIFSMs which use syntactically different, but equivalent transition predicates.

The set of possible faults are represented with transitions of the mutation machine that are not transitions of the specification. They can be introduced with different types of mutation operations. These operations include, but are not limited to, changing (adding) target states or outputs, merging/splitting inputs of transitions, replacing input variables with default values, swapping occurrences of input variables in input predicates, substituting an input variable for another, modifying arithmetic/-logical operations in input predicates. These operations introduce faults which cannot be represented in classical FSM; some of these faults are considered in [30, 15, 3, 6]. Note that merging and splitting of inputs are not considered in [29]. We allow for mutation machines to have more states than the specification. Additional states can be used to represent for instance faults reflecting various interpretations of operating modes of a system; e.g., a designer can duplicate some control state and specify different behaviors in the two versions of the state, thus introducing extra control states.

The transitions of the mutants and the specifications are all defined in mutation machine. We distinguish several types of transitions in a mutation machine.

Definition 9 (Unaltered, mutated, suspicious, trusted and untrusted transitions) A transition of $\mathcal{M}$ that is also a transition of $\mathcal{S}$ is called *unaltered*, otherwise it is a *mutated* transition. A transition of $\mathcal{M}$ is *suspicious* if it is compatible with another transition of $\mathcal{M}$. An unaltered transition is *trusted* if it is not suspicious; otherwise it is *untrusted*.

So, mutated transitions can be viewed as alternatives for unaltered transitions and they represent faults in implementations. Suspicious transitions including untrusted and mutated transitions can be replaced in the specification or a mutant to obtain another mutant. The specification, as well as every mutant, contains all the trusted transitions because no alternative (fault) was defined for them. Let $Susp(\mathcal{M})$ denote the set of all suspicious transitions of $\mathcal{M}$ and $Untr(\mathcal{S})$ denote the set of untrusted transitions. By definition, $Untr(\mathcal{S}) = Susp(\mathcal{M}) \cap N$.

Example 3 (Mutation machine, deterministic and nondeterministic executions) Fig. 2 presents an example of a nondeterministic SIFSM $\mathcal{M}_1$ which is a mutation machine for the specification machine $\mathcal{S}_1$ in Fig. 1. The mutation machine has five mutated transitions depicted with dashed lines. The solid lines represent the unaltered transitions of the specification machine. Names of transitions are presented in brackets and parentheses for mutated and unaltered transition, respectively. The transitions t_5 and t_6 are compatible, t_5 and t_7 are compatible; but t_6 and t_7 are not compatible. The executions $t_1t_3t_5t_{10}$ and $t_1t_3t_7t_6$ are deterministic and the execution $t_1t_3t_7t_5$ is nondeterministic because it includes two compatible transitions. 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 also contained in the specification; so they are unaltered and untrusted.

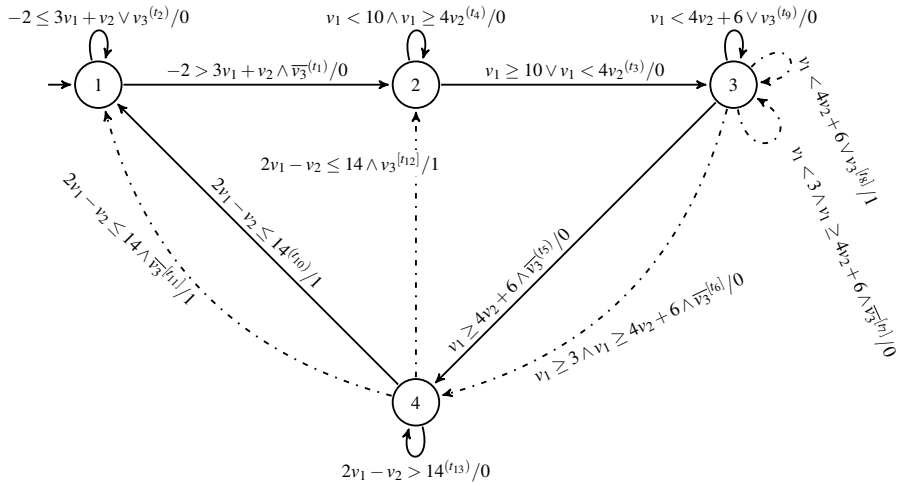

Fig. 2 A mutation machine $\mathcal{M}_1$ for the specification machine in Fig. 1, state 1 is initial. Names assigned to mutated transitions represented with dashed lines appear in brackets; solid lines represent unaltered transitions contained in the specification machine.

The set of all mutants in $\mathcal{M}$ is called a *fault domain* for $\mathcal{S}$, denoted $Mut(\mathcal{M})$. If $\mathcal{M}$ is deterministic then $Mut(\mathcal{M})$ contains just $\mathcal{S}$. A *fault subdomain* for $\mathcal{S}$ is a subset of $Mut(\mathcal{M})$. Each mutant $\mathcal{P}$ has all the trusted transitions of $\mathcal{M}$ and a unique set of suspicious transitions $Susp(\mathcal{P})$ which we can use to identify $\mathcal{P}$. Each mutant is a completely specified deterministic SIFSM, while a state of the mutation machine may have compatible alternative transitions which belong to different mutants. This motivates the following definition.

Definition 10 (Cluster of a state and suspicious state)
Given state s of $\mathcal{M}$, a subset of $T(s)$ is called a *cluster* of s if it is deterministic and the inputs of its transitions constitute a tautology. State s is said to be *suspicious* if it has more than one cluster.

Let $Susp(s)$ denote the set of all suspicious transitions in state s and $Z(s)$ denote the set of all clusters of s. Different types of transitions can appear in a given cluster. We use S_{susp} to denote the set of all suspicious states of $\mathcal{M}$. A transition can be in a mutant or the specification only together with all the transitions in a cluster containing the transition. Each state in each machine in $Mut(\mathcal{M})$ or the specification has exactly one cluster because each machine is deterministic and completely specified. The number of mutants contained in $Mut(\mathcal{M})$ is the product of the sizes of the clusters of the states minus one, i.e., $|Mut(\mathcal{M})| = \Pi_{s \in S}|Z(s)| - 1$.

Example 4 (Clusters and mutants) Consider the mutation machine $\mathcal{M}_1$ in Fig. 2. Only states 3 and 4 have more than one cluster; so they are suspicious states. The four clusters of state 3 are $\{t_5, t_9\}$, $\{t_5, t_8\}$, $\{t_6, t_7, t_9\}$ and $\{t_6, t_7, t_8\}$. State 4 has two clusters $Z(4) = \{\{t_{10}, t_{13}\}, \{t_{11}, t_{12}, t_{13}\}\}$. The mutation machine contains eight mutants including seven mutants and the specification machine. One of them is shown in Fig. 3.

A mutation machine for a completely specified Mealy machine contains mutated transitions sharing the inputs of a specification machine [27]. In case of SIFSMs, symbolic inputs (i.e., predicates on input variables) on mutated transitions may differ from that of the specification, as a result, a mutated transition may not belong to a completely specified deterministic SIFSM. To exclude such transitions we require a mutation SIFSM to be well-formed.

Definition 11 (Well-formed mutation machine)
We say that a mutation machine is *well-formed* if each of its mutated transitions belongs to a cluster.

Example 5 (Well-formed mutation machine)
The mutation machine in Fig. 2 is well-formed. But if we remove transition t_{12} from the machine, the resulting mutation machine is not well-formed.

Henceforth, we consider only well-formed mutation machines.

3.2 Fault Model and Complete Test Suite

In this paper, we focus on conformance testing of implementations against their specifications both modeled by DSIFSM. The implementations are represented with mutants in a mutation machine. The conformance relation is defined as the equivalence relation between the initial states of a mutant and the specification.

Let $\mathcal{M}$ be a mutation machine for a specification machine $\mathcal{S} = (S, s_0, V, O, N)$.

Definition 12 (Conforming and nonconforming mutants) A mutant $\mathcal{P} = (P, p_0, V, O, E)$ in $Mut(\mathcal{M})$ is conforming to $\mathcal{S}$, if $p_0 \simeq s_0$, otherwise, it is nonconforming. We say that an input sequence distinguishing for $\mathcal{P}$ and $\mathcal{S}$ *detects* or *kills* $\mathcal{P}$.

An execution in a mutant producing an unexpected output sequence reveals a fault; then, the mutant is detected by any input sequence triggering the execution. This justifies the following definition concerning executions of submachines of $\mathcal{M}$.

Definition 13 (Revealing execution) Let $\mathcal{P}$ be a nonconforming (possibly nondeterministic) submachine of a mutation machine and $\alpha \in G^*$ be an input sequence. An execution e_1 of $\mathcal{P}$ is α*-revealing* (or simply revealing) if there exists an execution e_2 of $\mathcal{S}$ such that $out(e_2) \neq out(e_1)$ and α triggers both e_1 and e_2 and this does not hold for any prefix of α.

Revealing executions of mutants are always deterministic, revealing executions of nondeterministic submachines can be deterministic or nondeterministic.

Example 6 (Nonconforming submachine)
The DSIFM $\mathcal{P}_1$ in Fig. 3 is a submachine of the mutation machine $\mathcal{M}_1$ in Fig. 2. $\mathcal{P}_1$ is nonconforming. The input sequence $\alpha_1 = (3 * v_1 + v_2 < -3 \wedge \overline{v_3})(v_1 \geq 8 \wedge v_2 > 3)(-7 \leq v_1 < 2 \wedge v_2 \leq -4 \wedge \overline{v_3})(6 \leq v_1 \leq 7 \wedge v_2 == 0 \wedge \overline{v_3})$ triggers executions $e_1 = t_1 t_3 t_5 t_{10}$ in the DSIFSM in Fig. 1 and execution $e_2 = t_1 t_3 t_7 t_6$ in the DSIFSM in Fig. 3. The output sequence $out_{\mathcal{S}_1}(e_1) = 0001$ is not equal to $out_{\mathcal{P}_1}(e_2) = 0000$. Thus e_2 is α-revealing, meaning that $\mathcal{P}_1$ is nonconforming to $\mathcal{S}_1$ and α_1 is a distinguishing input sequence.

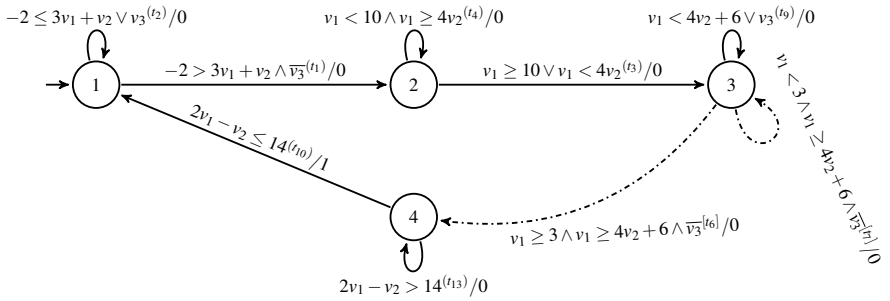

Fig. 3 A nonconforming submachine $\mathcal{P}_1$ w.r.t. the specification $\mathcal{S}_1$ in Fig. 1, state 1 is initial.

We consider a general fault model defined as the tuple $\langle \mathcal{S}, \simeq, \mathit{Mut}(\mathcal{M}) \rangle$ following [27,28,29]. It specifies the faults to be detected in implementations obtained by seeding the specification with the faults. For a given specification machine $\mathcal{S}$ the conformance relation partitions the set $\mathit{Mut}(\mathcal{M})$ into conforming mutants and nonconforming ones. Implementations represented with conforming mutants cannot to be detected and nonconforming mutants gather faults causing revealing executions. We want to generate tests detecting all implementations represented with nonconforming mutants in a fault model.

Definition 14 (Complete test suite for fault domain) A *test suite* is a finite set of tests. A test suite is *complete* for the specification $\mathcal{S}$ w.r.t. a fault domain $\mathit{Mut}(\mathcal{M})$ if it detects all the nonconforming mutants in the fault domain.

Our goal is to generate complete test suites in an incremental way, i.e., augmenting an existing (possibly empty) test suite with additional tests. Therefore, we need first to address the problem of checking the completeness of a test suite. A simple minded way of checking the completeness of a given test suite is to enumerate and execute each and every mutant with the tests. For a nontrivial mutation machine, a sheer number of mutants makes their enumeration impracticable. Hence, instead of considering all the mutants, we try to determine only mutants one-by-one which survive the tests. Each surviving nonconforming mutant could then be used to determine an additional test until a test suite becomes complete. In the next section, we specify the mutants which survived a given set of tests with a Boolean expression, all together avoiding their enumeration. An individual mutant could then be determined by solving the expression using a solver [17].

4 Specifying Test-Surviving Mutants

Given the fault domain and tests, we want to build a Boolean expression such that each of its solutions determines a mutant of the fault domain undetected by the tests. Solving the resulting expression allows to check the completeness of given tests. Boolean expressions are built over the suspicious transitions occurring in revealing executions in mutants, i.e., executions whose output sequences differ from that of the execution of the specification with a given test. Our first step is to determine

such executions. Next, we build an expression to encode all the submachines of the mutation machine producing these executions. Finally, we add a constraint to make sure that the Boolean expression specifies only mutants among these submachines. The negation of the resulting expression encodes mutants which survived the given tests.

4.1 Determining Revealing Executions in Mutants

Tests detecting mutants trigger revealing executions in mutants. One way to obtain revealing executions of mutants contained in a mutation machine is to enumerate each and every mutant and then determine the revealing executions for each mutants. Enumerating all the mutants is inefficient. We propose to determine revealing executions of the mutation machine and to select among them only the deterministic ones for building Boolean expressions. This is because the executions of the mutants are included in the executions of the mutation machine and the executions of the mutants are always deterministic. Both deterministic and nondeterministic revealing executions of a mutation machine can be determined using a distinguishing automaton obtained by composing the transitions of the specification and mutation machines as follows.

Definition 15 (Distinguishing automaton for $\mathcal{S}$ and $\mathcal{M}$)
Given a DSIFSM $\mathcal{S} = (S, s_0, V, O, N)$ and a mutation machine $\mathcal{M} = (M, m_0, V, O, T)$ of $\mathcal{S}$, a finite automaton $\mathcal{D} = (D \cup \{\nabla\}, d_0, G, \Theta, \nabla)$, where $D \subseteq S \times M$, ∇ 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

- $d_0 = (s_0, m_0)$ is the initial state in D
- For any $(s,m) \in D$
  - $((s,m), g \cap h, (s', m')) \in \Theta$, if there exist $(s, g, o, s') \in N$, $(m, h, o', m') \in T$, such that $o = o'$ and $g \cap h \neq \emptyset$
  - $((s,m), g \cap h, \nabla) \in \Theta$, if there exist $(s, g, o, s') \in N$, $(m, h, o', m') \in T$, such that $o \neq o'$ and $g \cap h \neq \emptyset$

The language of $\mathcal{D}$, $L_{\mathcal{D}}$ is the set of tests labeling executions of $\mathcal{D}$ from d_0 to the sink state ∇.

The distinguishing automaton $\mathcal{D}$ has at most $|S||M| + 1$ states and at most $|M|\mu k$ transitions, where μ represents the maximal number of mutated transitions from any state of $\mathcal{M}$ and k represents the maximal number of unaltered transitions in N from any state of $\mathcal{S}$.

Example 7 (A distinguishing automaton for the specification and the mutation machine) Fig. 4 presents the distinguishing automaton for the mutation and specification machines in Fig. 2. For the sake of the presentation, the sink state occurs six times and every occurrence is numbered. Each state has one or many self-loop transitions which we do not depict. Each transition is labeled with two transitions determining it and its input is the conjunction of the inputs of the two transitions. The first transition is in the specification and the second is in the mutation machine. Multiple transitions

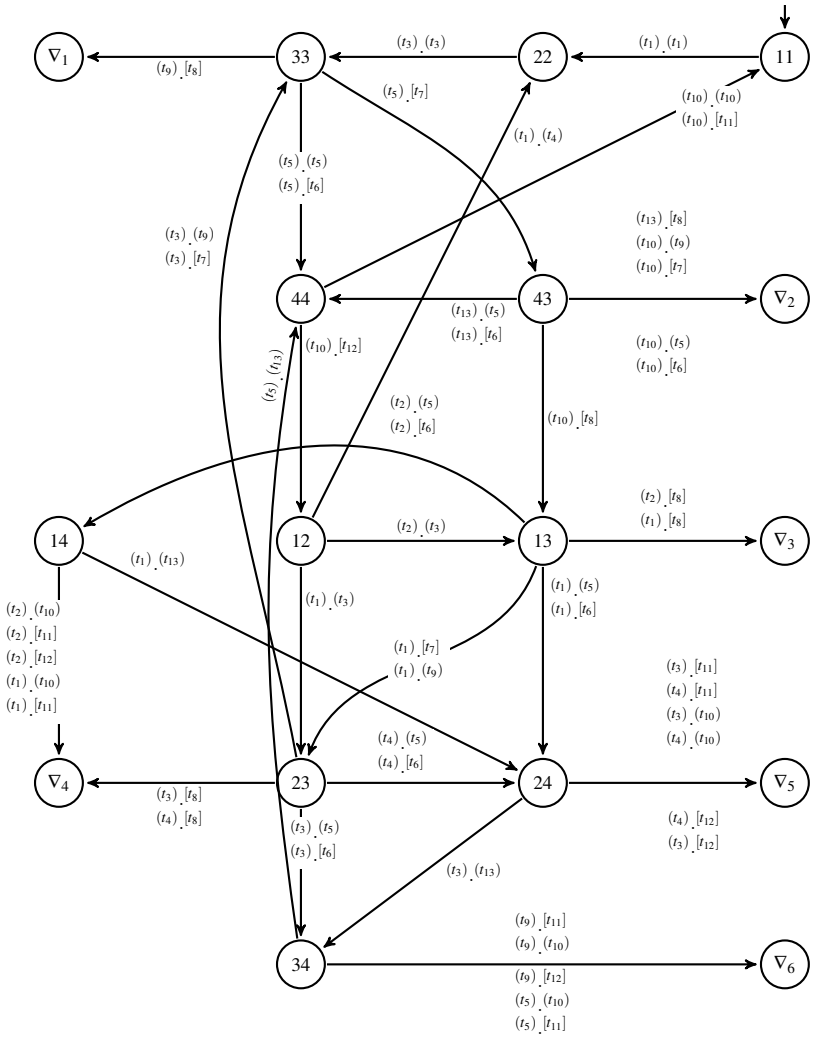

Fig. 4 The distinguishing automaton $\mathcal{D}_1$ for the specification machine $\mathcal{S}_1$ in Fig. 1 and the mutation machine $\mathcal{M}_1$ in Fig. 2, state 11 is initial. The input of each transition is represented with a pair of transitions. Multiple transitions are represented with a single arc and multiple pairs of transitions. Mutated transitions appear in brackets whereas unaltered transitions appear in parenthesis.

are represented with a single arc labeled with multiple pairs of transitions, e.g., there are five transitions from 43 to ∇_2.

A test α 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. The revealing executions of the mutation machine define the executions of the distinguishing automaton to the sink states. Let n represents the maximal number of transitions from any state of $\mathcal{D}$. A

test α of length $|\alpha|$ triggers at most $n^{|\alpha|}$ executions of the distinguishing automaton $\mathcal{D}$. Thus, the shorter is a test, the shorter is the time for determining the revealing executions triggered by the test.

Example 8 (Executions of the distinguishing automaton defined by executions of the mutation machine) The test α_1 from Example 6 triggers four executions in the distinguishing automaton in Fig. 2. Two of them go through the path $11 \to 22 \to 33 \to 43 \to \nabla_2$; they are determined by two revealing executions of the mutation machine; the first one $t_1t_3t_7t_6$ is deterministic and the second $t_1t_3t_7t_5$ is nondeterministic. The two others take the path $11 \to 22 \to 44 \to 43 \to 11$ and they are determined by non-revealing deterministic executions $t_1t_3t_5t_{10}$ and $t_1t_3t_5t_{11}$ of the mutation machine.

The revealing executions of $\mathcal{M}$ can be triggered with all the tests in $L_{\mathcal{D}}$.

Lemma 1 *Every $\alpha \in L_{\mathcal{D}}$ triggers an α-revealing execution in $\mathcal{M}$ and every revealing execution of $\mathcal{M}$ is triggered by some $\alpha \in L_{\mathcal{D}}$*

Proof Every test $\alpha \in L_{\mathcal{D}}$ triggers an execution of $\mathcal{D}$ to the sink state. Let e be an execution of $\mathcal{D}$ to the sink state. Let e' be an execution of $\mathcal{M}$ defining e and e'' be the execution of $\mathcal{S}$ defining e; such two executions always exist and output sequence of e' is different from the output sequence of e''. By definition, α is a reduction of the input sequences of e' and e'', i.e., $\alpha \subseteq inp(e') \cap inp(e'')$; this implies that α triggers both e' and e''. Combining the last two statements, we get that e' is α-revealing. □

We now state important properties of the distinguishing automaton. Mutants producing executions which belong to the set of α-revealing executions in the mutation machine are detected by α. Such mutants can be identified based on the suspicious transitions in revealing executions.

Let $Susp(e)$ denote the set of suspicious transitions in execution e in the mutation machine. For e to be an execution in a submachine, every suspicious transitions in e must be defined in the submachine.

Definition 16 (Submachine involved in an execution)
Let $\mathcal{P}$ be a submachine of a mutation machine. We say that $\mathcal{P}$ is *involved* in an execution e of the mutation machine if $Susp(e) \subseteq Susp(\mathcal{P})$.

Lemma 2 *Every $\alpha \in L_{\mathcal{D}}$ kills any mutant involved in a deterministic α-revealing execution of $\mathcal{M}$.*

Proof Let $\mathcal{P} = (P, p_0, V, O, E)$ be a mutant involved in a deterministic α-revealing execution e of $\mathcal{M}$. Then $\mathcal{P}$ can produce execution e. Since e is revealing, it produces a different output from that of the execution of $\mathcal{S}$ triggered by α. This means that $p_0 \simeq_\alpha s_0$ and α kills $\mathcal{P}$. □

The following theorem is a consequence of Lemma 1 and Lemma 2.

Theorem 1 *Given the distinguishing automaton $\mathcal{D}$ for specification $\mathcal{S}$ and mutation machine $\mathcal{M}$, and $\mathcal{P} = (P, p_0, V, O, E) \in Mut(\mathcal{M})$, $\mathcal{P}$ is nonconforming if and only if $p_0 \not\simeq_\alpha s_0$ for some $\alpha \in L_{\mathcal{D}}$.*

1 **Procedure** Generate_Det_Rev_Exec $(\alpha, \mathcal{D})$;
Input : α, a test
Input : $\mathcal{D}$, the distinguishing automaton for a mutation machine and its specification
Output : DR_α, the set of deterministic revealing executions of the mutation machine
2 Build the automaton A_α for α;
3 Build $\mathcal{D}_\alpha$ the product of A_α with $\mathcal{D}$;
4 Let next_config be a queue of configurations of the form (n, π) where n is a state of $\mathcal{D}_\alpha$ and π a path of the mutation machine;
5 Set $\pi_0 = \varepsilon$ the empty execution of the mutation machine;
6 Set next_config= $\{(n_0, \pi_0)\}$, where n_0 is the initial state of $\mathcal{D}_\alpha$;
7 **while** *next_config is not empty* **do**
8 Select the next configuration (n, π) in next_config;
9 **for** *every transition t outgoing from n* **do**
10 let t' be the transition of the mutation machine defining t;
11 **if** $\pi.t'$ *is deterministic* **then**
12 $\pi_{new} = \pi.t'$;
13 $n_{new} = tgt(t)$;
14 **if** n_{new} *is defined by the sink state of* $\mathcal{D}$ **then**
15 $DR_\alpha = DR_\alpha \cup \{\pi_{new}\}$;
16 **else**
17 Add (n_{new}, π_{new}) as the last configuration to next_config ;
18 **end**
19 **end**
20 **end**
21 **end**
22 **return** DR_α ;

Algorithm 1: Generation of deterministic revealing executions for a test α

For detecting nonconforming mutants, we may not only consider tests belonging to $L_\mathcal{D}$. The following corollary characterizes the set of tests which trigger revealing executions of the mutation machines; it is a consequence of Definition 3 and Lemma 1.

Corollary 1 *A test $\beta \in G^*$ triggers a revealing execution of $\mathcal{M}$ if and only if $\beta \subseteq \alpha$ for some $\alpha \in L_\mathcal{D}$.*

Checking whether a revealing execution is deterministic can be done by verifying that it does not use two different suspicious transitions from the same state. This verification can be performed on-the-fly on enumerating all the executions of $\mathcal{D}$ triggered by a given test. Based on Corollary 1 the enumeration can be done by building the synchronous product of an automaton representing a test and $\mathcal{D}$. The state transition graph of the automaton for a test of length n with $(n \geq 0)$ is a path having $n+1$ states and n transitions. There is one transition from state i to state $i+1$ labeled with the i^{th} input of the test, with $i = 1..n$. Algorithm 1 presents procedure *Generate_Det_Rev_Exec* for determining the deterministic revealing executions triggered by a test.

Example 9 The automaton for $\alpha_1 = (3v_1 + v_2 < -3 \wedge \overline{v_3})(v_1 \geq 8 \wedge v_2 > 3)(-7 \leq v_1 < 2 \wedge v_2 \leq -4 \wedge \overline{v_3})(6 \leq v_1 \leq 7 \wedge v_2 == 0 \wedge \overline{v_3})$ has 5 states and 4 transitions. The first input of α_1, $(3v_1 + v_2 < -3 \wedge \overline{v_3})$ labels the transition from state 1 to 2. Transitions between the other states are similarly defined.

4.2 Encoding SIFSMs Involved in Deterministic Revealing Executions

We use Boolean expressions for encoding SIFSMs involved in revealing executions. The Boolean expression specifying the submachines involved in a revealing execution is built over the variables representing the suspicious transitions of $\mathcal{M}$. The suspicious transitions are used to identify submachines of a mutation machine since each submachine has a distinct set of suspicious transitions. Given the set of suspicious transitions $Susp(\mathcal{M})$, we introduce $|Susp(\mathcal{M})|$ 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; such a variable is called a *transition variable*. A solution of a Boolean expression over the transition variables in $Susp(\mathcal{M})$ is an assignment to `True` or `False` of every transition variable which makes the expression `True`; such a solution can be obtained with solvers [9, 17] which return *null* in case the expression has no solution.

Definition 17 (Machine specified by a Boolean expression) A solution of a Boolean expression *selects* (resp. *excludes*) transitions to which it assigns the value `True` (resp `False`) to the corresponding transition variables. A solution of a Boolean expression *determines* a (possibly nondeterministic and partially specified) submachine $\mathcal{P}$ if it selects a subset of $Susp(\mathcal{M})$ which together with the trusted transitions of $\mathcal{M}$ constitutes the submachine. A Boolean expression *specifies* submachines determined by its solutions.

Let c be a Boolean expression over transition variables whose values indicate the presence or absence of corresponding suspicious transitions in submachines of a mutation machine. We denote by $Generate\_a\_submachine(c)$ a function which either returns a submachine specified by c or returns *null* if c has no solution. The function can call a solver to obtain a solution of c.

Example 10 Let $\{t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}\}$ be the Boolean transition variables for the eight suspicious transitions of $\mathcal{M}_1$ in Figure 2. Expression $c = t_6 \wedge t_7$ is defined over the eight transition variables; missing variable t can be added to c with expression $t \vee \bar{t}$. Consider a solution of c which assigns true to t_6, t_7 and t_{10} and false to the other variables. This solution selects three suspicious transitions t_6 and t_7 and t_{10}. The submachine in Figure 3 is specified by c because it is composed of the three transitions and the trusted transitions of $\mathcal{M}$, namely $t_1, t_2, t_3, t_4, t_9, t_{10}$ and t_{13}. By deleting t_{10} from the submachine in Figure 3, we obtain another submachine determined by a different solution of c; so it is also specified by c.

A mutation machine has both nondeterministic and deterministic revealing executions. Mutants are involved in deterministic revealing executions only. However partially specified and nondeterministic submachines can also be involved in deterministic revealing executions of the mutation machines. We consider deterministic revealing executions because they are the only involving nonconforming mutants we need to identify.

Let DR_α be the finite set of deterministic α-revealing executions of $\mathcal{M}$. We use Boolean expressions for encoding of suspicious transitions of executions in DR_α. Let

$e \in DR_\alpha$ be a deterministic α-revealing execution of $\mathcal{M}$. Let $c_e \stackrel{\text{def}}{=} \bigwedge_{t \in Susp(e)} t$ be the conjunction of all variables for the suspicious transitions in e. As usual, the disjunction over the empty set is *False* and the conjunction over the empty set is *True*. Note $Susp(e)$ is included in the set of variables $Susp(\mathcal{M})$ over which we build the Boolean expressions. A solution of c_e selects not only all the transitions in $Susp(e)$ but also some arbitrary suspicious transitions not in e; this is because we assumed that every Boolean expression is defined over the set of the variables for all the suspicious transitions. A deterministic α-revealing execution of the mutation machine may involve several submachines including the mutants. Those submachines are specified with c_e.

Lemma 3 *The conjunction $c_e \stackrel{\text{def}}{=} \bigwedge_{t \in Susp(e)} t$ of transition variables for transitions in $Susp(e)$ specifies the submachines involved in the revealing execution e.*

Corollary 2 *Let $\alpha \in G^*$ be a test. The Boolean expression $c_\alpha \stackrel{\text{def}}{=} \bigvee_{e \in DR_\alpha} c_e$ specifies all the submachines involved in all executions in DR_α and detected by the test α.*

Proof The expression c_α is the disjunction of c_e for every α-revealing execution $e \in DR_\alpha$. Since c_e specifies the submachines involved in e, c_α specifies all the submachines which are involved in all executions in DR_α. All such submachines are detected by α since they can produce a revealing execution in DR_α. □

Example 11 (Boolean expression specifying submachines involved in revealing executions) Let $\alpha_1 = (3v_1 + v_2 < -3 \wedge \overline{v_3})(v_1 \geq 8 \wedge v_2 > 3)(-7 \leq v_1 < 2 \wedge v_2 \leq -4 \wedge \overline{v_3})(6 \leq v_1 \leq 7 \wedge v_2 == 0 \wedge \overline{v_3})$ be the test introduced in Example 6. It triggers four executions in the distinguishing automaton in Fig. 4, only two amongst the four executions reach the sink state. The four executions are defined by four executions of mutation machine in Fig. 2 including $e_1 = t_1t_3t_7t_6$, $e_2 = t_1t_3t_7t_5$, $e_3 = t_1t_3t_5t_{11}$ and $e_4 = t_1t_3t_5t_{10}$. The executions e_1 and e_2 define the two executions to the sink state of the distinguishing automaton; they are are revealing. Execution e_2 is not involved in any mutant because it is nondeterministic. Only execution e_1 is deterministic and involved in mutants; it includes the two suspicious transitions in $Susp(e_1) = \{t_6, t_7\}$. Thus $c_{e_1} = t_6 \wedge t_7$ and $c_{\alpha_1} = c_{e_1}$. The solutions of c_{α_1} determine the submachines involved in e_1. The nonconforming mutant in Fig. 3 is one such submachine. Other submachines can be nondeterministic or partially specified, e.g., the machine obtained from the mutant by adding transition t_{12} of the mutation machine is a nondeterministic submachine involved in e_1.

Let $\overline{c}_\alpha$ denote the negation of c_α; it specifies the submachines not involved in any deterministic α-revealing execution.

Lemma 4 *A submachine is not involved in a deterministic α-revealing execution e if and only if it is specified with $\overline{c}_\alpha$.*

Thus the sets of suspicious transitions in all deterministic α-revealing executions represent all submachines detected by test α; on the other hand, α does not detect submachines which are not involved in these executions.

Lemma 5 *Test $\alpha \in G^*$ does not detect any submachine of $\mathcal{M}$ specified with $\overline{c}_\alpha$.*

The Boolean expressions $\overline{c}_\alpha$ specifies the submachines of a mutation machine not involved in deterministic revealing executions. These submachines exclude suspicious transitions in the revealing executions but they also include other transitions of the mutation machine, causing some of the specified submachines to be nondeterminism or partially specified.

Example 12 (Boolean expression specifying undetected submachines) Considering the running example, $\overline{c}_{\alpha_1} = \bar{t}_6 \vee \bar{t}_7$ does not specify the nonconforming mutant in Fig. 3. Nondeterministic submachines having the suspicious transitions t_6, t_8 and t_9 are determined by a solution of $\overline{c_\alpha}$ which assigns `True` to t_6, t_9 and t_8; such submachines are not mutants in $Mut(\mathcal{M})$. On the other hand the solution of $\overline{c}_{\alpha_1}$ which assigns `True` to t_5, t_9, t_{11} and t_{12}, and `False` to the variables for the others suspicious transitions of the mutation machine determines a mutant.

To determine the mutants undetected by a test, we must exclude the nondeterministic and partially specified submachines from the submachines specified by $\overline{c}_\alpha$, by adding a constraint that only mutants should be considered.

4.3 Encoding Mutants (un) Detected by a Test

Let $\langle \mathcal{S}, \simeq, Mut(\mathcal{M})\rangle$ be a fault model. As mutants are submachines of mutation machines, they can also be identified with the suspicious transitions as discussed in the previous subsection. So, we build Boolean expression over the transitions variables representing the suspicious transitions of $\mathcal{M}$ for specifying the mutants in $Mut(\mathcal{M})$.

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 transition variables of a cluster Z_i expresses the requirement that all the transitions corresponding to the variables 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 uniquely determined by Boolean expression $z_i \stackrel{\text{def}}{=} (\bigwedge_{t\in Z_i} t) \wedge (\overline{\bigvee_{t\in Susp(s)\setminus Z_i} t})$ which permits the selection of all the suspicious transitions in Z_i and excludes the remaining suspicious transitions leaving s.

Lemma 6 *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.*

Proof Given two clusters $Z_i, Z_j \in Z(s)$, there is a transition $t \in T(s)$ belonging only to one cluster. Assume that $t \in Z_i$ and $t \notin Z_j$. By definition, any solution of z_i selects t and any solution of z_j cannot select t, which means that a solution of z_i cannot be a solution of z_j and vice versa.

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 7 *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.*

Proof Let x be a valuation of the Boolean variables for the suspicious transitions of state s. By definition, the valuation x is a solution of c_s if and only if x is a solution of at least one z_i. According to Lemma 6, the sets of solutions of the Boolean expressions z_i and z_j for any two clusters $Z_i, Z_j \in Z(s)$ used in the definition of c_s are disjoint. Consequently x is a solution of c_s if and only if x is a solution of exactly one z_i. So a solution of c_s determines a cluster in $Z(s)$ and every cluster in $Z(s)$ is determined by a solution of c_s. □

Example 13 (Boolean expression specifying the clusters for suspicious state 3)
For the four clusters of state 3, namely, $Z_{3_1} = \{t_5, t_9\}$, $Z_{3_2} = \{t_5, t_8\}$, $Z_{3_3} = \{t_6, t_7, t_9\}$ and $Z_{3_4} = \{t_6, t_7, t_8\}$ we build expressions $z_{3_1} = t_5 t_9 (\overline{t_6 \vee t_7 \vee t_8})$, $z_{3_2} = t_5 t_8 (\overline{t_6 \vee t_7 \vee t_9})$, $z_{3_3} = t_6 t_7 t_9 (\overline{t_5 \vee t_8})$ and $z_{3_4} = t_6 t_7 t_8 (\overline{t_5 \vee t_9})$. A solution of any expressions cannot be a solution of any of the three others. Then $c_3 = (z_{3_1} \vee z_{3_2} \vee z_{3_3} \vee z_{3_4})$.

Each solution of $\bigwedge_{s \in S_{susp}} c_s$ 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 transitions present in the specification can be expressed with the negation of the conjunction of the variables of all the untrusted transitions $\overline{\bigwedge_{t \in Untr(\mathcal{S})} t}$. Any of its solutions excludes at least one cluster in the specification and therefore cannot determine the specification. The Boolean expression $c_{clstr} \stackrel{\text{def}}{=} \bigwedge_{s \in S_{susp}} c_s \wedge \overline{\bigwedge_{t \in Untr(\mathcal{S})} t}$ excludes nondeterministic and partially specified submachines and the specification, meaning that c_{clstr} specifies only all mutants in the fault domain $Mut(\mathcal{M})$

Lemma 8 *A submachine $\mathcal{P}$ of $\mathcal{M}$ is a mutant if and only if c_{clstr} specifies $\mathcal{P}$.*

Proof If $\mathcal{P}$ is a mutant then each state of $\mathcal{P}$ defines exactly one cluster. $\mathcal{P}$ is determined by the solution of c_{clstr} which selects the transitions of the clusters defined in each state of $\mathcal{P}$ connected to the initial state. Conversely, any solution of c_{clstr} selects exactly one cluster from states of the $\mathcal{M}$. The machine composed of the transitions in clusters and the trusted transitions of $\mathcal{M}$ is a mutant because it has exactly one cluster in each of its states. □

Example 14 (Boolean expression specifying the fault domain) The Boolean expression specifying the mutants contained in mutation machine $\mathcal{M}_1$ in Fig. 2 is defined over the variables in $Susp(\mathcal{M}_1) = \{t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}\}$, the set of suspicicous transitions of $\mathcal{M}_1$. The suspicious transitions are defined in suspicious states in $S_{susp} = \{3, 4\}$. In the example 13 we have computed $c_3 = (z_{3_1} \vee z_{3_2} \vee z_{3_3} \vee z_{3_4})$. Similarly for state 4, we build $c_4 = (z_{4_1} \vee z_{4_2})$ where $z_{4_1} = t_{10} t_{13} (\overline{t_{11} \vee t_{12}})$ and $z_{4_2} = t_{11} t_{12} t_{10} \overline{t_{13}}$. The seven mutants and the specification contained in $\mathcal{M}_1$ are specified by $c_3 \wedge c_4$. Then $c_{clstr} = c_3 \wedge c_4 \wedge (\overline{t_5 \wedge t_9 \wedge t_{10}})$ specifies only the seven mutants because any solution of c_{clstr} excludes at least one of the three untrusted transitions t_5, t_9 and t_{10} of the specification. c_{clstr} does not specify any nondeterministic or partially specified submachine of $\mathcal{M}_1$, e.g., c_{clstr} does not specify any nondeterministic submachine containing the transitions t_6, t_8, t_9.

To exclude nondeterministic and partially specified submachines as well as the specification from the submachines specified by $\overline{c}_\alpha$, the Boolean expression c_{clstr}

```
1  Procedure Build_expression (TS, 𝒟);
   Input  : TS, a test suite
   Input  : 𝒟, the distinguishing automaton of mutation machine ℳ and specification 𝒮
   Output : c_TS, a Boolean expression defining submachines of ℳ involved in revealing executions
            for tests in TS
2  c_TS := False;
3  for each α ∈ TS do
4  |  DR_α = Generate_Det_Rev_Exec(α, 𝒟);
5  |  c_α := False;
6  |  for each e ∈ DR_α do
7  |  |  c_e := ⋀_{t∈Susp(e)} t ;
8  |  |  c_α := c_α ∨ c_e;
9  |  end
10 |  c_TS := c_TS ∨ c_α;
11 end
12 Return c_TS;
```

Algorithm 2: Building c_{TS}

specifying the mutants in a fault model must be added to $\overline{c}_\alpha$. The following theorem is a consequence of Lemma 5 and Lemma 8.

Theorem 2 *Test $\alpha \in G^*$ does not detect a mutant specified with $\overline{c_\alpha} \wedge c_{clstr}$.*

5 Generation of Complete Test Suites

In this section we elaborate a method for generating tests to be added to a given initial test suite to make a complete test suite. Clearly, if the initial test suite is complete there is no need to generate new tests. In what follows, we propose first a method for checking the completeness of a given test suite; the method is based on solving Boolean expression specifying the test-surviving mutants and returns a test killing a surviving mutant in case the given test suite is not complete. Then the method is iteratively called for generating an additional test which together with given tests constitutes a complete test suite.

5.1 Checking the Completeness of a Test Suite

Let us define $c_{TS} \stackrel{\text{def}}{=} \bigvee_{\alpha \in TS} c_\alpha$, a Boolean expression which specifies the submachines involved in revealing executions for the tests in *TS*. Procedure *Build_expression* for building c_{TS} is presented in Algorithm 2. 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 $Mut(\mathcal{M})$ to obtain *FD*. A fault subdomain can always be reduced with an expression specifying the mutants to be excluded. In particular, 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 the mutants it specifies is conforming.*

1 **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 revealing a nonconforming mutant which survived the test suite;
  $\alpha = \varepsilon$, if *TS* is complete
2 $c_{TS} := Build\_expression(TS, \mathcal{D})$;
3 $c_{fd} := \overline{c_{TS}} \wedge c_{fd}$;
4 $c_{\mathcal{P}} := False$;
5 $\alpha := \varepsilon$;
6 **repeat**
7 $c_{fd} := c_{fd} \wedge \overline{c_{\mathcal{P}}}$;
8 $\mathcal{P} := Generate\_a\_submachine(c_{fd})$;
9 **if** $\mathcal{P} \neq null$ **then**
10 Build $\mathcal{D}_{\mathcal{P}}$, the distinguishing automaton of $\mathcal{S}$ and $\mathcal{P}$;
11 **if** $\mathcal{D}_{\mathcal{P}}$ *has no sink state* **then**
12 $c_{\mathcal{P}} := \bigwedge_{t \in Susp(\mathcal{P})} t$;
13 **else**
14 Set α to an input sequence in $L_{\mathcal{D}_{\mathcal{P}}}$;
15 **end**
16 **end**
17 **until** $\alpha \neq \varepsilon$ *or* $\mathcal{P} = null$;
18 **return** (c_{fd}, α)

Algorithm 3: Checking the completeness of a test suite for a fault domain

The fault domain $Mut(\mathcal{M})$ is specified with c_{clstr}, which leads to Corollary 3.

Corollary 3 *Test suite TS is complete for* $Mut(\mathcal{M})$ *if and only if* $\overline{c_{TS}} \wedge c_{clstr}$ *has no solution or each of the mutants it specifies is conforming.*

Based on Theorem 3, checking the completeness of a test suite for a fault subdomain *FD* amounts to its iterative reduction 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 3 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 5.2. Procedure *Check_completeness* proceeds as follows. It calls *Build_expression* for building c_{TS}, the Boolean expression which specifies 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 specifies 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 Complete_test_gen $(TS_{init}, \langle \mathcal{S}, \simeq, Mut(\mathcal{M})\rangle)$;
Input : TS_{init}, an initial (possibly empty) test suite
Input : $\langle \mathcal{S}, \simeq, Mut(\mathcal{M})\rangle$, 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 $Mut(\mathcal{M})$;
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 4: Generation of a complete test suite from initial test suite TS_{init}

Procedure *Check_completeness* makes calls to *Generate_a_submachine* to select a mutant in a fault domain specified with 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 nonempty 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 represented with mutation machines.

Example 15 (Checking the completeness of a test suite) In checking the completeness of the initial test suite $\{\alpha_1\}$ for the example mutation machine and test α_1, *Check_completeness* takes as input $c_{fd} = c_{clstr}$, $TS = \{\alpha_1\}$ and the distinguishing automaton in Fig. 4. Then, it determines $c_{TS} = c_{\alpha_1} = t_6 \wedge t_7$, sets $c_{fd} = \overline{c_{TS}} \wedge c_{clstr}$ where $c_{TS} = c_{\alpha_1}$, $c_{\mathcal{P}} = False$ 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 four suspicious transitions t_5, t_9, t_{11} and t_{12}.

The mutant is nonconforming and killed by the test $\alpha_2 = (-2 > 3v_1 + v_2 \wedge \bar{v}_3)(v_1 \geq 10 \vee v_1 < 4v_2)(v_1 \geq 4v_2 + 6 \wedge \bar{v}_3)(2v_1 - v_2 \leq 14 \wedge v_3)((-2 \leq 3v_1 + v_2 \vee v_3) \wedge (v_1 \geq 10 \vee v_1 < 4v_2))(-2 \leq 3v_1 + v_2 \wedge 6 + 4v_2 \leq v_1 \wedge \bar{v}_3)(-2 \leq 3v_1 + v_2 \wedge 2v_1 - v_2 \leq 14 \wedge \bar{v}_3)$ labeling two paths to the sink state in the distinguishing automaton. The length of α_2 is 7. The execution of *Check_completeness* terminates with outputs c_{fd} and nonempty test $\alpha = \alpha_2$, which indicates that the test suite $\{\alpha_1\}$ is not complete.

5.2 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

Table 1 Execution of Procedure *Complete_test_gen* with the initial test α_1.

	In *Check_completeness*				End of the step	
step	Revealing Executions for TS_{curr}	$c_{TS_{curr}}$	c_{fd}	Suspicious transitions a mutant specified with c_{fd}	TS	TS_{curr}
init	$\emptyset$	`True`	c_{clstr}	$\{t_9,t_5,t_{11},t_{12}\}$	$\emptyset$	α_1
1	$t_1t_3t_7t_6$	$t_6 \wedge t_7$	$c_{fd} \wedge (\bar{t}_6 \vee \bar{t}_7)$	$\{t_9,t_5,t_{11},t_{12}\}$	α_1	α_2
2	$t_1t_3t_5t_{12}t_3t_5t_{11}$	$t_5 \wedge t_{11} \wedge t_{12}$	$c_{fd} \wedge (\bar{t}_5 \vee \bar{t}_{11} \vee \bar{t}_{12})$	$\{t_8,t_5,t_{10}\}$	α_1,α_2	α_3
3	$t_1t_3t_8$	t_8	$c_{fd} \wedge \bar{t}_8$	no surviving mutant	α_1, α_2, α_3	ε
α_1	$(3v_1+v_2<-3\wedge\overline{v_3})(v_1\geq 8\wedge v_2>3)(-7\leq v_1<2\wedge v_2\leq -4\wedge\overline{v_3})(6\leq v_1\leq 7\wedge v_2==0\wedge\overline{v_3})$					
α_2	$(-2>3v_1+v_2\wedge\bar{v}_3)(v_1\geq 10\vee v_1<4v_2)(v_1\geq 4v_2+6\wedge\bar{v}_3)(2v_1-v_2\leq 14\wedge v_3)((-2\leq 3v_1+v_2\vee v_3)\wedge(v_1\geq 10\vee v_1<4v_2))(-2\leq 3v_1+v_2\wedge 6+4v_2\leq v_1\wedge\bar{v}_3)(-2\leq 3v_1+v_2\wedge 2v_1-v_2\leq 14\wedge\bar{v}_3)$					
α_3	$(-2>3v_1+v_2\wedge\bar{v}_3)(v_1\geq 10\vee v_1<4v_2)(v_1<4v_2+6\vee v_3)$					

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 4 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 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 specification* $\mathcal{S}$ *w.r.t. a fault domain* $Mut(\mathcal{M})$.

Proof Procedure *Complete_test_gen* always terminates since 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. Moreover, at every iteration step *Check_completeness* returns a new test detecting mutants which survived the tests generated in previous steps. On termination of *Complete_test_gen*, the initial and generated tests detect all nonconforming mutants, meaning that they constitute a complete test suite w.r.t. the fault domain $Mut(\mathcal{M})$. □

Example 16 (Complete test suite generation) Considering the running example, Table 1 summarizes data computed in executing *Complete_test_gen* to generate a complete test suite from the initial test suite $TS_{init} = \{\alpha_1\}$. The iteration steps appear 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 $\{\alpha_1, \alpha_2, \alpha_3\}$ 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.

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 [20]. 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 [17] for solving of 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 [9] 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 [26] 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 [31]. In our previous work [18], the input of each transition in the hierarchical model was represented with a Boolean variable. The flattened and determinized SIFSM with 13 states, 62 transitions and 22 Boolean input variables. In this experiments we replace every Boolean variable of the SIFSM used in [18] with the original predicate of the transition. The predicates are defined over Boolean and integer variables. The resulting SIFM has 5 Boolean input variables and 6 integer input variables. Every state has at most 7 outgoing transitions.

Then we have manually introduced faults (transition faults, output faults, swapping of variables, replacing variables with constants, changing comparison operators), obtaining a mutation machine with $2^{13} - 1 = 8191$ mutants, excluding the specification. As we report in first column of the first part of Table 2, our tool generates, within 47 seconds, a complete test suite with 13 tests detecting the mutants. The tool takes 8 seconds in determining the clusters, 12 seconds in building the distinguishing

Table 2 Experimental results with the prototype tool: case of output faults

Part 1: faults on arithmetic and comparison operations, swapping variables, replacing variables with constant							
#Mutants	8191	1.8E+5	1.2E+6	1.8E+7	3.7E+8	2.5E+9	3.2E+11
#Tests	13	12	18	18	29	32	30
Total (s.)	47	133	143	175	548	531	519
Cluster (s.)	8	26	29	44	42	63	149
Dist. aut. (s.)	12	36	53	47	58	68	68
Rev. ex. (s.)	9	23	36	43	141	109	124
Test Sel. (s.)	15	49	19	38	300	285	170
B. exp. Solv. (s.)	0.04	0.09	0.09	0.13	0.45	0.48	0.68
Part 2: detecting output faults							
#Mutants	12288	7.3E+4	1.7E+6	3.5E+7	7.6 E+8	1.2E+10	1.2E+15
#Tests	13	14	16	19	22	23	36
Total (s.)	20	24	22	29	59	51	74
Cluster (s.)	6	5	9	10	9	20	38
Dist. aut. (s.)	3	2	2	2	5	3	3
Rev. ex. (s.)	3	3	4	6	12	9	12
Test Sel. (s.)	6	3	8	10	19	17	17
B. exp. Solv. (s.)	0.03	0.02	0.02	0.03	0.05	0.06	0.1
Part 3: detecting transfer faults							
#Mutants	8.1E+3	7.3E+4	1.1E+6	1.9E+8	5.5E+10	8.3E+12	2.8E+14
#Tests	14	22	20	28	68	53	48
Total (s.)	59	120	114	111	594	3897	427
Cluster (s.)	7	11	8	10	25	49	43
Dist. aut. (s.)	34	44	41	50	80	110	138
Rev. ex. (s.)	6	17	11	18	132	2670	125
Test Sel. (s.)	11	45	51	31	355	267	115
B. exp. Solv. (s.)	0.02	0.03	0.04	0.04	0.2	0.3	0.3

automaton, 9 seconds in extracting the revealing executions, 15 seconds in selecting tests from nonconforming mutants and 0.04 second in solving the Boolean expression specifying the undetected mutants.

We also generated complete tests from automatically generated mutation machines. We considered transfer faults, output faults, faults on arithmetics and comparison operations, and randomly selected faults. The others columns of the first part of Table 2 show the results obtained from mutation machines consisting of a mix of different types of faults including faults on arithmetic and comparison operations, swapping variables, replacing variables with constant. Note that these faults implicitly introduce splitting and merging of the inputs.

The second part of Table 2 presents the results for the mutation machines with output faults only. We observed that the maximal length of the tests is eight. Most of the computational time is spent on building the distinguishing automaton and determining the clusters. The third part of the table is for mutation machines with transfer faults only.

Analyzing the experimental results, we observed that the smaller the number of mutated transitions the faster the test generation process; In this case, a test triggers a relatively small number of executions of the mutation machine and especially when in

addition, mutation operations do not introduce numerous conforming mutants, which is often considered as a realistic situation [15].

7 Conclusion

We proposed a fault model and constraint solving-based approach to generate complete tests for symbolic input finite state machine (SIFSM), lifting the approach we developed earlier for classical Mealy FSM.

We have developed a prototype tool and obtained promising test generation results for an industrial-like case study with a rather big number of mutants. The experiment indicate that the proposed approach is particularly efficient when a fault domain does not have numerous confirming mutants.

Our current work focuses on extending the approach to FSMs with symbolic outputs determined using arithmetic operations over input variables [24] and adapting it to the implementations which cannot be reset.

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. Araujo H. L. S, Carvalho G., Sampaio A., Mousavi M. R., Taromirad M.: A process for sound conformance testing of cyber-physical systems. In: Prooceedings of IEEE International Conference on Software Testing, Verification and Validation Workshops. pp. 46–50 (2017)
2. 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)
3. 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)
4. Cavalcanti A., Simão A.: Fault-based testing for refinement in CSP. In: Proceedings 29th Ifip WG 6.1 International Conference on Testing Software and Systems. pp. 21–37. Springer International Publishing (2017)
5. 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)
6. 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)
7. 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)
8. 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)
9. 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)
10. 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)
11. 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)

12. Huang W.-L., Peleska J.: Complete model-based equivalence class testing for nondeterministic systems. Formal Aspects of Computing 29(2), 335-364. (2017)
13. 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)
14. Hübner F., Huang W.-L., Peleska J.: Experimental evaluation of a novel equivalence class partition testing strategy. In: Proceedings of the 9th International Conference on Tests and Proofs. pp. 155–172. Springer International Publishing (2017)
15. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37(5), 649–678 (sep 2011)
16. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293–303 (2009)
17. 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)
18. Nguena Timo, O., Petrenko, A., Ramesh, S.: Multiple Mutation Testing from Finite State Machines with Symbolic Inputs. In: Proceedings 29th IFIP WG 6.1 International Conference on Testing Software and Systems. pp. 36–51. Springer International Publishing (2017)
19. 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)
20. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2nd edn. (2013)
21. 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)
22. Peleska J., Huang W.-L. : Complete model-based equivalence class testing. Int J Softw Tools Technol Transfer 18, No. 3, pp. 265-283. Springer-Verlag (2016)
23. Petrenko, A.: Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography. In: Cassez F., Jard C., Rozoy B., Ryan M.D. (eds) Modeling and Verification of Parallel Processes. MOVEP 2000. Lecture Notes in Computer Science, vol 2067. Springer, Berlin, Heidelberg (2001)
24. 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)
25. Petrenko A., Boroday S., Groz R. : Confirming configurations in EFSM. In: Wu J., Chanson S.T., Gao Q. (eds) Formal Methods for Protocol Engineering and Distributed Systems. IFIP Advances in Information and Communication Technology, vol 28. Springer, Boston, MA (1999)
26. 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)
27. 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)
28. 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)
29. 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)
30. Pretschner A.: Defect-Based Testing. Dependable Software Systems Engineering, 141–163, (2017)
31. 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)
32. Taromirad M., Mousavi M. R.: Gray-Box Conformance Testing for Symbolic Reactive State Machines. In: Proceedings of the 7th International Conference on Fundamentals of Software Engineering. pp. 228-243. Springer International Publishing (2017)
33. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability 22(5), 297–312 (2012)