Multiple Mutation Testing from FSM

Alexandre Petrenko, Omer Nguena Timo, S. Ramesh

Multiple Mutation Testing from FSM

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

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

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

Abstract. Fault model based testing receives constantly growing interest of both, researchers and test practitioners. A fault model is typically a tuple of a specification, fault domain, and conformance relation. In the context of testing from finite state machines, the specification is an FSM of a certain type. Conformance relation is specific to the type of FSM and for complete deterministic machines it is equivalence relation. Fault domain is a set of implementation machines each of which models some faults, such as output, transfer or transition faults. In the traditional checking experiment theory the fault domain is the universe of all machines with a given number of states and input and output sets of the specification. Another way of defining fault domains similar to the one used in classical program mutation is to list a number of FSM mutants obtained by changing transitions of the specification. We follow in this paper the approach of defining fault domain as a set of all possible deterministic submachines of a given nondeterministic FSM, called a mutation machine, proposed in our previous work. The mutation machine contains a specification machine and extends it with a number of mutated transitions modelling potential faults. Thus, a single mutant represents multiple mutations and mutation machine represents numerous mutants. We propose a method for analyzing mutation coverage of tests which we cast as a constraint satisfaction problem. The approach is based on logical encoding and SMT-solving, it avoids enumeration of mutants while still offering a possibility to estimate the test adequacy (mutation score). The preliminary experiments performed on an industrial controller indicate that the approach scales sufficiently well.

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

1 Introduction

In the area of model based testing, one of the key questions concerns a termination rule for test generation procedures. It seems to us that there are two main schools of thought considering this rule. One of them follows a traditional approach of covering

a specification model. In terms of the Finite State Machine (FSM) model, one could consider for coverage various features of an FSM, such as transitions or sequences of them which model test purposes often used to guide and terminate test generation. Another school focuses on fault coverage and thus follows fault model based testing, see, e.g., [26, 21, 22, 15, 16, 20].

Fault model based testing receives constantly growing interests of both, researchers and test practitioners. Fault models are defined in the literature in a variety of ways [26]. In [11], we propose to define a fault model as a tuple of a specification, a fault domain, and a conformance relation. In the context of testing from finite state machines, the specification is a certain type of an FSM. A conformance relation is specific to the FSM type and for complete deterministic machines it is equivalence relation. A fault domain is a set of implementation machines, aka mutants, each of which models some faults, such as output, transfer and transition faults.

In the traditional checking experiment theory the fault domain is the universe of all machines with a given number of states and input and output alphabets of the specification, see, e.g., [23, 9, 12, 13, 8, 14]. While this theory offers clear understanding what does it mean to have sound and exhaustive, i.e., complete tests, it leads to tests whose number grows in the worst case exponentially with the FSM parameters. To us, this is a price to pay for considering the universe of all FSMs. Intuitively, choosing a reasonable subset of this fault domain might be the way to mitigate the test explosion effect. As an example, if one considers the fault domain of mutants that model output faults, a test complete for this fault model is simply a transition tour. The space between these two extreme fault models has received in our opinion insufficient attention. In what follows, we present a brief account of what has been done in this respect.

In the area of program mutation testing, mutants are generated by modifying programs. The number of tests is limited by the number of mutants, which usually need to be compared one by one with the original program to determine tests that kill them [3, 4]. Test minimization could then be achieved via explicit enumeration of all the mutants in the fault domain followed then by solving a set cover problem.

Mutation testing in hardware area seems to predate program mutation. An early work of Poage and McCluskey in 1964 [2] focuses on hardware faults in FSM implementations and builds a fault domain by extracting FSM mutants from modified circuits. The idea of this approach is to consolidate the comparisons of individual mutants aiming at reduction of the number of tests, however, mutants still need to be analyzed one by one. The approach in [1] focuses on detection of single FSM mutations with the same test, but provides no guarantees that mutants with multiple mutations (higher order mutants) can always be killed.

Explicit mutant enumeration can be avoided by defining a fault domain as a set of all possible submachines of a given nondeterministic FSM, called a mutation machine, proposed in our previous work [5, 10, 7]. The mutation machine contains a specification machine and extends it with a number of mutated transitions modelling potential faults. Mutated transitions might be viewed as faults injected in the specification machine, see, e.g., [25]. Thus, a single mutant represents multiple mutations and mutation machine represents numerous mutants. In our previous work, methods were developed for test generation using this fault model [5, 10, 7]. The main idea

was to adjust classical checking experiments for a fault domain smaller than the universe of all FSMs. A checking experiment once obtained is in fact a complete test suite, however, this approach does not offer a means of analyzing mutation coverage of an arbitrary test suite or individual tests.

Traditional program mutation testing uses explicit mutant enumeration to determine test adequacy or mutation score. It is a ratio of the number of dead mutants to the number of non-equivalent mutants. We are not aware of any attempt to characterize a fault detection power of tests considering multiple mutants that avoids their enumeration.

The paper aims at solving this problem. We propose a method for analyzing mutation coverage of tests which we cast as a constraint satisfaction problem. The approach is based on logical encoding and SMT-solving, it avoids enumeration of mutants while still offering a possibility to estimate the test adequacy (mutation score). The analysis procedure can be used for test prioritization and test minimization, and could eventually lead to an incremental test generation.

The remaining of this paper is organized as follows. Section 2 defines a specification model as well as a fault model. In Section 3, we develop a method for mutation coverage analysis. Section 4 reports on our preliminary experiments performed on an industrial controller. Section 5 summarizes our contributions and indicates future work.

2 Background

2.1 Finite State Machines

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

M is *completely specified* (complete FSM) if for each tuple $(s, x) \in S \times I$ there exists transition $(s, x, o, s') \in T$. It is *deterministic* (DFSM) if for each $(s, x) \in S \times I$ there exists at most one transition $(s, x, o, s') \in T$; if there are several transitions for some $(s, x) \in S \times I$ then it is *nondeterministic* (NFSM); M is *observable* if for each tuple $(s, x, o) \in S \times I \times O$ there exists at most one transition; if there are several transitions for some $(s, x, o) \in S \times I \times O$ then it is *non-observable.*

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

A *trace* of M in state s is a string of input-output pairs which label an execution from s. Let $Tr_M(s)$ denote the set of all traces of M in state s and Tr_M denote the set of

traces of M in the initial state. Given sequence $\beta \in (IO)^*$, the *input* (*output*) *projection* of β, denoted $\beta\downarrow_I$ ($\beta\downarrow_O$), is a sequence obtained from β by erasing symbols in O (I).

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

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

We define several relations between states in terms of traces of a complete FSM.

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

We also use relations between machines. Given FSMs $M = (S, s_0, I, O, T)$ and $N = (P, p_0, I, O, N)$, $N \leq M$ if $s_0 \leq p_0$; $N \simeq M$ if $s_0 \simeq p_0$; $N \not\simeq M$ if $s_0 \not\simeq p_0$. In this paper, we use equivalence relation between machines as a conformance relation between implementation and specification machines.

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

2.2 Fault Model

Let $A = (S, s_0, I, O, N)$ be a complete initially connected DFSM, called the *specification* machine.

Definition 1. A complete initially connected NFSM $M = (S, s_0, I, O, T)$ is a *mutation* machine of $A = (S, s_0, I, O, N)$, if $N \subseteq T$, i.e., if A is a submachine of M.

We assume that all possible implementation machines for the specification machine A constitute the fault domain $Sub(M)$, the set of all deterministic submachines of the mutation machine M of A. A submachine $B \in Sub(M)$, $B \neq A$ is called a *mutant*. Transitions of M that are also transitions of A are called *unaltered*, while others, in the set $T \setminus N$, are *mutated* transitions. Given $(s, x) \in S \times I$, we let $T(s, x)$ denote a set of transitions from state s and input x in M. If $T(s, x)$ is a singleton then its transition is called a *trusted* transition. The set $T(s, x)$ is called a *suspicious* set of transitions if it is not a singleton, transitions in a suspicious set are called *suspicious*. Trusted transitions are present in all mutants, but suspicious transitions in each set $T(s, x)$ are alternative and only one can be present in a deterministic mutant.

A mutant B is *nonconforming* if it is not equivalent to A, otherwise, it is called a *conforming* mutant. We say that input sequence $\alpha \in I^*$ such that $B \not\simeq_\alpha A$ *detects* or kills the mutant B.

The tuple $\langle A, \simeq, Sub(M)\rangle$ is a fault model following [11]. For a given specification machine A the equivalence partitions the set $Sub(M)$ into conforming implementations and faulty ones. In this paper, we do not require the FSM A to be reduced, this implies that a conforming mutant may have fewer states than the specification A; on the other hand, we assume that no fault creates new states in implementations, hence mutants with more states than the specification FSM are not in the fault domain $Sub(M)$.

Consider the following example.

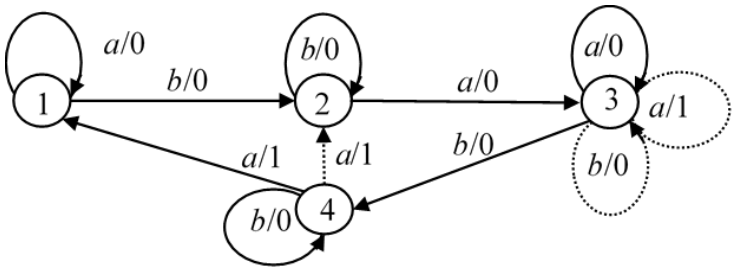

Fig. 1. A mutation machine with the specification machine as its submachine, where mutated transitions are depicted with dash lines, state 1 is the initial state.

The mutation machine M contains six suspicious transitions, one mutated transition represents output fault and the other two transfer faults. M contains eight deterministic submachines, the specification machine and seven mutants which share the same five trusted transitions.

As discussed in previous work [5, 10, 7], the mutation machine formally models test hypotheses about potential implementation faults. The mutation machine M allows compact representation of numerous mutants in the fault domain $Sub(M)$. More precisely, their number is given by the following formula:

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

In the extreme case, considered in classical checking experiments a fault domain is the universe of all machines with a given number of states and fixed alphabets. The corresponding mutation machine becomes in this case a chaos machine with all possible transitions between each pair of states. The number of FSMs it represents is the product of the numbers of states and outputs to the power of the product of the numbers of states and inputs.

3 Mutation testing

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

In the domain of program mutation testing, such a test suite is often called adequate for a program (in our case, a specification machine) relative to a finite collection of programs (in our case the set of mutants), see, e.g., [4].

Differently from the classical program mutation testing, where the mutant killing tests are constructed mostly manually, in case of deterministic FSMs, tests that kill a given mutant FSM can be obtained from the product of the two machines, see, e.g., [2, 1, 27]. The problem can also be cast as model checking for a reachability property, considered in several work, see, e.g., [18]. This approach can also be used to check whether a given test kills mutants, but it requires mutant enumeration.

In this work, we develop an analysis approach that avoids mutant enumeration while still offering a possibility to estimate the test adequacy (mutation score).

3.1 Distinguishing automaton

Tests detecting mutants of the specification are presented in a product of the specification and mutation machines obtained by composing their transitions as follows.

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

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

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

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

Theorem 1. Given the distinguishing automaton D for A and M, if $B \not\simeq_\alpha A$, $B \in Sub(M)$, then $\alpha \in L_D$.

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

At the same time, not each and every word of the language detects a mutant. An input sequence $\alpha \in L_D$ triggers several executions in the distinguishing automaton D which are defined by a single execution in the specification machine A and some execution in the mutation machine M both triggered by α. The latter to represent a mutant must be deterministic. Such a deterministic execution of the mutation machine M defining (together with the execution of A) an execution of the distinguishing automaton D to the sink state is called α-*revealing*. Input sequences triggering revealing executions enjoy a nice property of being able to detect mutants.

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

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

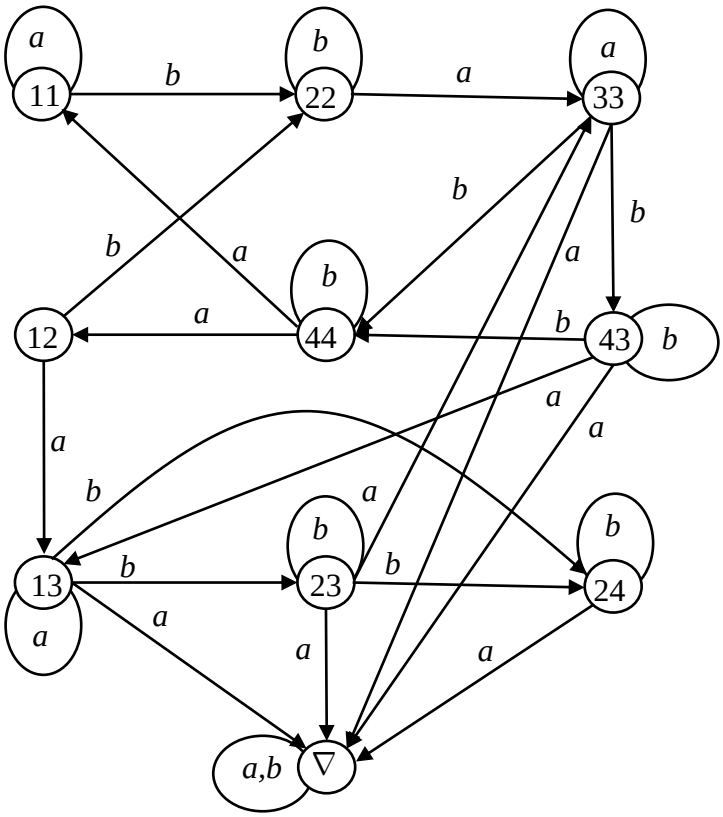

Fig. 2. The distinguishing automaton D for the specification A and mutation M machines in Fig.1, state 11 is the initial state.

3.2 Mutation coverage analysis

Consider an input sequence $\alpha \in I^*$ which detects a nonconforming mutant by triggering α-revealing executions. Analyzing these executions we can determine all mutated transitions involved in each of them. This analysis can performed by using a distinguishing automaton constrained to a given input sequence.

Let $\alpha \in I^*$ and $Pref(\alpha)$ be the set of all prefixes of α. We define a linear automaton $(Pref(\alpha), \varepsilon, I, D_\alpha)$, such that each prefix of α is a state, and $(\beta, x, \beta x) \in D_\alpha$ if $\beta x \in Pref(\alpha)$.

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

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

We illustrate the definition using the input sequence $\alpha = baaba$ for the specification and mutation machines in Fig. 1. Notice that the sequence hits all the mutated transitions in the mutation machine. The resulting α-distinguishing automaton for A and M is shown in Fig. 3.

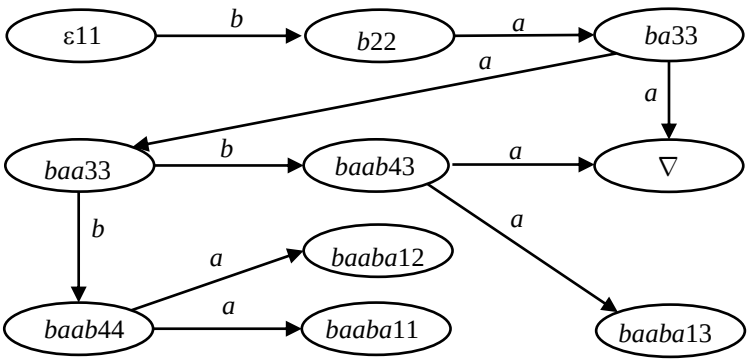

Fig. 3. The α-distinguishing automaton D_α for the specification A machine and mutation machine M in Fig. 1, where $\alpha = baaba$.

Notice that the input sequence *baaba* and its prefix *baa* trigger two executions which end up in the sink state ∇. These are

1. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *a*, 1, 3)**
2. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *a*, 0, 3)(3, *b*, 0, 3)(3, *a*, 0, 3)**.

The suspicious transitions are in bold. The executions are deterministic and include two mutated transitions (3, *a*, 1, 3) and (3, *b*, 0, 3). The third mutated transition (4, *a*, 1, 2) is in the execution that does not lead to the sink state ∇. Hence, the input sequence *baaba* detects any mutant with two out of three mutated transitions.

The example indicates that α-distinguishing automata for the specification and mutation machines provide a suitable means for mutation analysis of a given test suite. Before we formulate a method for such an analysis, we consider yet another example of α-distinguishing automata with $\alpha = babaaba$.

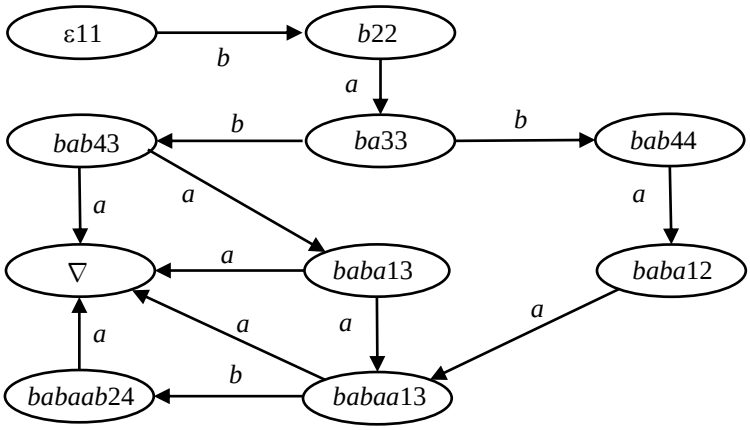

Fig. 4. A fragment of α-distinguishing automaton D_α for the specification A machine and mutation machine M in Fig. 1, where α = *babaaba*; executions missing the sink state are not shown.

The executions of the automaton in Fig. 4 leading to the sink state define the following executions of the mutation machine:

1. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 3)(3, *a*, 0, 3)**
2. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 3)(3, *a*, 1, 3)(3, *a*, 1, 3)**
3. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 3)(3, *a*, 1, 3)(3, *a*, 0, 3)(3, *b*, 0, 4)(4, *a*, 1, 1)**
4. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 2)**(2, *a*, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 2)**
5. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 2)**(2, *a*, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 1)**
6. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 3)(3, *a*, 1, 3)(3, *a*, 0, 3)(3, *a*, 1, 3)**
7. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 4)(4, *a*, 1, 2)**(2, *a*, 0, 3)**(3, *a*, 1, 3)**
8. (1, *b*, 0, 2)(2, *a*, 0, 3)**(3, *b*, 0, 3)(3, *a*, 1, 3)(3, *a*, 0, 3)(3, *b*, 0, 4)(4, *a*, 1, 2)**

The executions 3, 5, 6 and 8 are nondeterministic in the mutation machine, since each of them has both unaltered and mutated transitions for the same pair of state and input.

Consider the first execution, it involves two suspicious transitions, mutated transition (3, *b*, 0, 3) and unaltered transition (3, *a*, 0, 3). The prefix *baba* of the input sequence *babaaba* detects any mutant in which unaltered transition (3, *b*, 0, 4) is replaced by the mutated one (3, *b*, 0, 3) and the suspicious transition (3, *a*, 0, 3) is left unaltered. Let *B* be a set of transitions of a mutant $B \in Sub(M)$.

A mutant B is then detected by the input sequence *babaaba* or its prefix if its set of transitions *B* satisfies at least one of the following constraints on suspicious transitions:

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

7. (3, *b*, 0, 4), (4, *a*, 1, 2), (3, *a*, 1, 3) $\in B$
8. (3, *b*, 0, 3), (3, *a*, 1, 3), (3, *a*, 0, 3), (3, *b*, 0, 4), (4, *a*, 1, 2) $\in B$

Clearly nondeterministic executions 3, 5, 6 and 8 have unsatisfiable constraints since they require, e.g., that suspicious transition (3, *b*, 0, 4) is unaltered and replaced by the mutated transition (3, *b*, 0, 3) in the same mutant.

As stated above any mutant with the transition relation satisfying one of these constraints is detected by the input sequence *babaaba* or its prefix, since a wrong output sequence should be produced by such a mutant. On the other hand, a mutant that does not satisfy any of them escapes detection by this input sequence. To characterize these mutants, we formulate constraints which exclude all the executions of detected mutants by considering the negation of the disjunction of the constraints for all the triggered revealing executions. The resulting constraint becomes a conjunction of negated constraints of the executions.

For instance, the negated first constraint is (3, *b*, 0, 3) $\notin B$ or (3, *a*, 0, 3) $\notin B$. This reads that the unaltered transition (3, *b*, 0, 4) or mutated transition (3, *a*, 1, 3) must be present. The constraint (3, *b*, 0, 3) $\notin B$ is equivalent to (3, *b*, 0, 4) $\in B$; similarly, (3, *a*, 0, 3) $\notin B$ is equivalent to (3, *a*, 1, 3) $\in B$. We have that the negated constraint is (3, *b*, 0, 4) $\in B$ or (3, *a*, 1, 3) $\in B$.

To formalize the above discussions we cast the execution analysis as a constraint satisfaction problem by using auxiliary variables to specify the choices between suspicious transitions. Let $T_1, T_2, \ldots, T_m$ be the sets of suspicious transitions, where unaltered transitions are the first elements and the remaining elements of each set are lexicographically ordered. We introduce auxiliary variables $z_1, z_2, \ldots, z_m$, such that variable z_i represents the suspicious set T_i. For the variable z_i the domain is $D_i = \{1, 2, \ldots, |T_i|\}$, such that $z_i = 1$ represents the unaltered transition in the set T_i and the other values correspond to mutated transitions. We use conditional operators $\{=, \neq\}$ and logical operators AND ($\wedge$) and OR ($\vee$) for constraint formulas.

Each execution of a mutation machine that involves suspicious transitions yields assignments on variables representing these transitions, which expresses a constraint formula as the conjunction of individual assignments ($z_i = c$), where $c \in D_i$. Then the negated constraint formula becomes the disjunction of individual constraints ($z_i \neq c$).

A set of revealing executions triggered by one or more input sequences is then the conjunction of disjunctions of individual constraints.

In our example, the sets of suspicious transitions are

T_1(3, *a*) = {(3, *a*, 0, 3), (3, *a*, 1, 3)},

T_2(3, *b*) = {(3, *b*, 0, 4), (3, *b*, 0, 3)} and

T_3(4, *a*) = {(4, *a*, 1, 1), (4, *a*, 1, 2)}.

Each of these sets define two values of variables z_1, z_2 and z_3, where the value 1 of each variable represents a corresponding unaltered transition.

The constraint formula becomes:

$((z_2 \neq 2) \vee (z_1 \neq 1)) \wedge ((z_2 \neq 2) \vee (z_1 \neq 2)) \wedge ((z_2 \neq 2) \vee (z_1 \neq 2) \vee (z_1 \neq 1) \vee (z_2 \neq 1) \vee (z_3 \neq 1)) \wedge ((z_2 \neq 1) \vee (z_3 \neq 2)) \wedge ((z_2 \neq 1) \vee (z_3 \neq 2) \vee (z_3 \neq 1)) \wedge ((z_2 \neq 2) \vee (z_1 \neq 2) \vee (z_1 \neq 1)) \wedge ((z_2 \neq 1) \vee (z_3 \neq 2) \vee (z_1 \neq 2)) \wedge ((z_2 \neq 2) \vee (z_1 \neq 2) \vee (z_1 \neq 1) \vee (z_2 \neq 1) \vee (z_3 \neq 2))$.

Clearly, the formula always has a solution where values of variables determine unaltered transitions representing a specification machine, but we need a solution if it exists which has at least one mutated transition. To this end, we add the constraint $(z_1 \neq 1) \vee (z_2 \neq 1) \vee (z_3 \neq 1)$ excluding the solution defining the specification machine.

The final constraint formula is

$((z_2 \neq 2) \vee (z_1 \neq 1)) \wedge ((z_2 \neq 2) \vee (z_1 \neq 2)) \wedge ((z_2 \neq 2) \vee (z_1 \neq 2) \vee (z_1 \neq 1) \vee (z_2 \neq 1) \vee (z_3 \neq 1)) \wedge ((z_2 \neq 1) \vee (z_3 \neq 2)) \wedge ((z_2 \neq 1) \vee (z_3 \neq 2) \vee (z_3 \neq 1)) \wedge ((z_2 \neq 2) \vee (z_1 \neq 2) \vee (z_1 \neq 1)) \wedge ((z_2 \neq 1) \vee (z_3 \neq 2) \vee (z_1 \neq 2)) \wedge ((z_2 \neq 2) \vee (z_1 \neq 2) \vee (z_1 \neq 1) \vee (z_2 \neq 1) \vee (z_3 \neq 2)) \wedge ((z_1 \neq 1) \vee (z_2 \neq 1) \vee (z_3 \neq 1))$.

To solve it, we use the SMT solver Yices [23] which finds the solution $(z_1 = 2)$, $(z_2 = 1)$, $(z_3 = 1)$. The solution defines a mutant with the single mutated transition (3, *a*, 1, 3). The mutant is nonconforming, which can be verified with the help of a distinguishing automaton obtained for the specification machine and the mutant. This means that the input sequence *babaaba* does not detect the mutant defined by the solution. To ensure its detection we have two options, to add a new input sequence or to try to extend the input sequence *babaaba* until it detects the remaining mutant. The latter option avoids using the reset operation in testing, required in the former option.

Following the first option we notice that the input sequence which detects the escaped mutant is *baa* already obtained in the example of the α-distinguishing automaton in Fig. 3, where α = *baaba*. Considering the revealing execution (1, *b*, 0, 2)(2, *a*, 0, 3)(3, *a*, 1, 3) triggered by its prefix *baa*, we generate an additional constraint $(z_1 \neq 2)$ which prevents the suspicious transition (3, *a*, 1, 3) to be chosen and add it to the final constraint formula which has no solution. The set {*babaaba*, *baa*} is therefore a complete test suite for the specification machine A and mutation machine M in Fig. 1.

Following the second option, we find that it is possible to extend the input sequence *babaaba* which leaves the specification machine in state 3 with the input *a* to detect the mutated transition (3, *a*, 1, 3). As before, we add constraint $(z_1 \neq 2)$ and the final constraint has no solution. The set {*babaabaa*} is also a complete test suite.

This example indicates that various test generation strategies could be investigated, complementing checking experiments and checking sequences approaches. The latter allows one to avoid using multiple resets in testing. Notice that a classical checking experiment for this example derived by using, e.g., the W-method [12, 13], contains many more input sequences, moreover, the specification machine in Fig. 1 has no distinguishing sequence, which is usually required to generate a checking sequence. By this reason the existing methods cannot construct a single test, however, the example indicates that the mutation analysis allows us to do so. We leave the detailed elaboration of a test generation method for future work and formulate in this paper a procedure for mutant coverage analysis.

The procedure uses as inputs a test suite *TS* for a specification machine A and mutation machine M and consists of the following steps:

1. For each input sequence $\alpha \in TS$
   (a) Determine the α-distinguishing automaton
   (b) Find all executions leading to the sink state
   (c) Determine α-revealing executions of the mutation machine

(d) Build the disjunction of constraints excluding the α-revealing executions
2. Build the conjunction of the obtained disjunctions and add the constraint that excludes the solution defining the specification machine
3. Solve the constraint formula by calling a solver
4. If it finds no solution terminate with the message "*TS* is complete", otherwise check whether the mutant defined by a solution is conforming
5. If it is nonconforming terminate with the message "*TS* is incomplete", otherwise add the constraint that excludes the solution defining the conforming mutant and go to Step 3.

The main steps of the procedure have already been discussed and illustrated on the examples, except of the last two steps which deserve more explanation. Constraint solvers normally provide a single solution if it exists. An extra constraint prevents the solution to point to just the specification machine, but the found solution may correspond to a conforming mutant. In the domain of general mutation testing the problem of dealing with mutants equivalent, i.e., conforming, to the specification is well understood. In testing from an FSM, most approaches assume that the specification machine is reduced, so conforming mutants are isomorphic machines. Checking FSM equivalence is based on an FSM product. Notice that the proposed approach does not require the specification machine be reduced.

The complexity of the proposed method is defined by the number of constraints. We expect that the method scales well, since the recent advances in solving techniques drastically improve their scalability [23, 24]. The number of constraints for a single execution is limited by the number of states of a mutation machine, but the number of executions increases with the number of mutated transitions. On the other hand, the number of executions of the distinguishing automaton which do not end up in the sink state grows with the number of mutated transitions, as faults may compensate each other. These executions are not revealing and do not contribute to the number of constraints. In Section 4 we present the results of our preliminary experiments performed on an industrial controller to assess the scalability of the approach.

3.3 Applications

The proposed mutation coverage analysis approach allows one to check if a given test suite is a complete test suite. A logical formula constructed by the proposed method represents the coverage of the test suite for a given fault model. If the test suite is found to be incomplete the question arises on how its quality in terms of fault coverage can be characterized. In the traditional software mutation testing, the fault detection power of tests is characterized by mutation score. It is a ratio of the number of killed mutants to the number of non-equivalent mutants. Note that the number of all possible mutants remains unknown and the mutation score is determined based on a limited set of generated mutants. As opposed to this approach, in our approach the total number of mutants can always be determined using the formula given in Section 2.2. Moreover, while the mutation analysis method avoids complete mutant enumeration, it does generate conforming mutants while searching for nonconforming ones.

The enumeration of conforming mutants is achieved by adding constraints to a logical formula excluding repeated generation of already found mutants.

In the same vain, our method can be enhanced to generate and enumerate (at least partially) undetected nonconforming mutants. Once a nonconforming mutant is given by a solution found by a SMT solver and the method terminates declaring the test suite to be incomplete, we may continue this process by adding a constraint excluding its repeated generation. As a result a list of nonconforming mutants can be obtained. Two extreme cases of incomplete tests are worth to be discussed here.

First, a given test suite may have no detection capability at all. This property is in fact detected very early by the method; in this case all the α-distinguishing automata have no sink state reachable from the initial states, tests generate no constraints, the method can terminate at this step since there is no need to call a solver. No mutant in $Sub(M)$ is killed, the score is zero.

Second, a given test suite is "almost" complete and kills most of the mutants in $Sub(M)$. In this case, the process of nonconforming mutant generation does not take much time and once terminated yields the number of conforming mutants c as well as the number of survived nonconforming ones n. Then the mutation score is computed as follows:

$$(|Sub(M)| - c - n) / (|Sub(M)| - c).$$

It is worth to note that the way the mutation score is determined is completely different from that in software mutation testing, as our method generates mutants based on a given test suite and not the other way around.

When a given test suite is "far" from being complete the number of survived nonconforming mutants can explode especially when a mutation machine is close to a complete chaos machine which represents the complete universe of FSMs. In this situation one possible solution to cope with the mutant explosion problem is to terminate generating nonconforming mutants once their number reaches a predefined maximum, e.g., a percentage of $|Sub(M)|$ or the time period allocated for mutation analysis ends. The obtained score is an (optimistic) estimation of an upper bound of the actual mutation score.

The proposed procedure could also be used for test minimization by defining a subsume relation between tests based on comparison of the logical formulas generated from them. Tests subsumed by other tests can always be removed from the original test suite. Similarly the generated formulas can be used to prioritize tests when needed, see, e.g., [28].

4 Experimental results

In this section we report on a prototype tool implementing the proposed approach and its use on a case study of an FSM model of an automotive controller of industrial size.

4.1 Prototype tool

The prototype tool takes as inputs a mutation machine and a test suite, both described in text format. The inputs are parsed with an ANTLR-based module [30] to build an internal representation of the two objects. The mutation analysis algorithm manipulates these representations to build α-distinguishing automata, determine revealing executions of the mutation machine and generate constraints for the Yices SMT solver [23]. The solver is used as a backend to decide the satisfiability of the constraints. The tool parses the outputs from Yices to extract a solution if it is found to build a mutant. The prototype can also be used with other SMT solvers compatible with the SMT-LIB 2.0.

4.2 Case study

In our experiments, we use as a case study an automotive controller of the air quality system, which we also used in our previous work [29]. The functionality of the controller is to set an air source position depending on its current state and a current input from the environment.

The controller is initially specified as a hierarchical Simulink Stateflow model. Fig. 5 gives an overview of the model which is composed of three super-states *s*1, *s*2 and *s*23 and 13 simple states. Each super-state is composed of states and transitions. The initial state is the simple state *s*3. To obtain an FSM we introduced an input alphabet replacing transitions guards and flattened the hierarchical machine. We have identified 24 abstract inputs and two outputs. The resulting FSM has 14 states, since we added (for modeling of a branching behavior implemented with C code in the original state) one extra state to the given 13 simple states. It has 24 x 14 = 336 transitions.

The mutation machine was constructed from the following assumption about potential implementation faults. These faults may occur in outgoing transitions from any of the simple states in two super-states, namely *s*2 and s23 and four inputs, as Table 1 shows. The obtained mutation machine has 46 mutated transitions. The formula in Section 2 gives the number of mutants being equal to 3^{12} x 2^{17} = 69,657,034,752 including the specification machine.

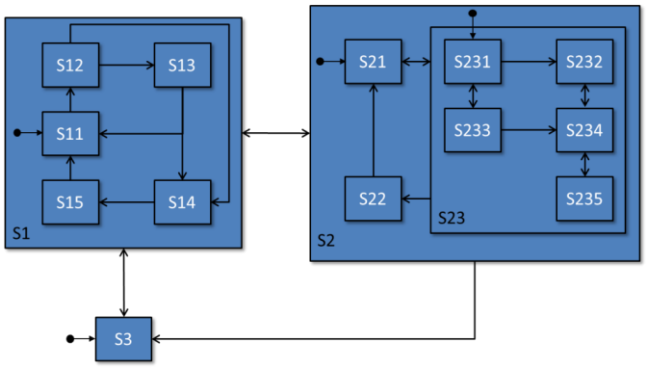

Fig. 5. An overview of the Simulink/Stateflow model in the controller

	s21	s22	s231	s232	s233	s234	s235
*a*2	3	3	3	3	3	3	3
*a*4	2	2	2	2	2	2	2
*a*14	1	1	3	3	3	3	3
*a*16	1	1	4	4	4	4	4

Tab. 1. The numbers of transitions for some pairs of states and inputs in the mutation machine (for the remaining pairs no mutated transitions were added).

4.3 Mutation analysis

To perform the mutation analysis, we needed a test suite, which could be generated randomly, however, we find it difficult to obtain tests that hit suspicious transitions in this case study, since 26 out of 336 transitions of the specification machine become suspicious in the mutation machine. We decided to use an early prototype of a test generation tool (which is work in progress) as an input for the mutation analysis tool. The tool generates test cases one by one, so that the mutation analysis tool processes a test suite of an increasing size. The process terminates once a current test suite is found to be complete. In this experiment, the test suite completeness was determined when it had 31 test cases. The length of the test cases varies from 4 to 25 and the number of revealing executions triggered by each of them varies from 1 to 13. In the last, 31st execution of Yices, it was given the formula of 69 clauses, for which it found no solution, meaning that the test suite is complete for the given mutation machine. The mutation analysis process took less than one minute on a desktop computer with the following settings: 3.4 Ghz Intel Core i7-3770 CPU, 16.0 GB of RAM, Yices 2.4.1, and ANTLR 4.5.1.

The fact that the tool was able to determine that the given test suite kills each non-conforming mutant out of 69,657,034,752 possible mutants indicates that the approach scales sufficiently well on a typical automotive controller even when the number of mutants is big. In this experiment, we varied only the number of tests (from 1 to 31), hence more experiments by varying the specification as well as mutation machines are needed to assess the tool scalability.

5 Conclusions

In this paper we focused on fault model based testing, assuming that a fault model is given as a tuple of a specification FSM, equivalence as a conformance relation and a fault domain. A fault domain is a set of implementation machines, aka mutants, each of which models some faults, such as output, transfer or transition faults. Avoiding their enumeration we define the fault domain as a set of all possible submachines of a given nondeterministic FSM, called a mutation machine, as we did in our previous work. The mutation machine contains a specification machine and extends it with a number of mutated transitions, modelling potential faults. Thus a single mutant represents multiple mutations and mutation machine represents numerous mutants. In the

area of mutation testing we could not find any attempt to analyze fault detection power of tests considering multiple mutants that avoids their enumeration.

We proposed a method for analyzing mutation coverage of tests which we cast as a constraint satisfaction problem. The method relies on the notion of a distinguishing automaton that is a product of the specification and mutation machines. To analyze mutation coverage of a single input sequence we define a distinguishing automaton constrained by this sequence. This allows us to determine all mutants revealing executions that are triggered by the input sequence. The executions are then used to build constraint formulas to be solved by an existing solver, Yices, in our experiments. The approach avoids enumeration of mutants while still offering a possibility to estimate the test adequacy (mutation score).

The preliminary experiments performed on an industrial controller indicate that the approach scales sufficiently well. We are planning to further enhance the approach to Extended FSMs [17] using mutation operators already defined for this type of FSMs.

Acknowledgements. This work is supported in part by GM R&D and the MEIE of Gouvernement du Québec. The authors would like to thank the reviewers for their useful comments.

References

1. Pomeranz, I., Sudhakar M. R.: Test generation for multiple state-table faults in finite-state machines. IEEE Transactions on Computers 46 (7), pp. 783-794 (1997)
2. Poage, J.F., McCluskey, Jr., E. J.: Derivation of optimal test sequences for sequential machines. In: Proceedings of the IEEE 5th Symposium on Switching Circuits Theory and Logical Design, pp. 121-132 (1964)
3. DeMillo, R. A., Lipton, R. J., Sayward, F. G. Hints on test data selection: help for the practicing programmer. IEEE Computer 11(4), pp. 34-41 (1978)
4. DeMilli, R. A., Offutt, J.A.: Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17(9), pp. 900-910 (1991)
5. Grunsky, I.S., Petrenko, A.: Design of checking experiments with automata describing protocols. Automatic Control and Computer Sciences. Allerton Press Inc. USA. No. 4 (1988)
6. Hennie, F. C.: Fault detecting experiments for sequential circuits. In: Proceedings of the IEEE 5th Annual Symposium on Switching Circuits Theory and Logical Design. Princeton, pp. 95-110 (1964)
7. Koufareva, I., Petrenko, A., Yevtushenko, N.: Test generation driven by user-defined fault models. In: Proceedings of the 12th International Workshop on Testing of Communicating Systems, pp. 215-233 (1999)
8. Lee, D., Yannakakis, M.: Principles and methods of testing finite-state machines - a survey. Proceedings of the IEEE, vol. 84, No. 8, pp. 1090-1123 (1996)
9. Moore, E.F.: Gedanken - Experiments on sequential machines. In: Automata Studies. Princeton University Press, pp. 129-153 (1956)
10. Petrenko, A., Yevtushenko, N.: Test suite generation for a FSM with a given type of implementation errors. In: Proceedings of IFIP 12th International Symposium on Protocol Specification, Testing, and Verification, pp. 229-243 (1992)
11. Petrenko, A., Yevtushenko, N., Bochmann, G. v.: Fault models for testing in context. In Formal Description Techniques IX, Springer, pp. 163-178 (1996)

12. Vasilevskii, M.P.: Failure diagnosis of automata. Cybernetics. Plenum Publishing Corporation. New York. 4, pp. 653-665 (1973)
13. Chow T.S.: Testing software design modeled by finite-state machines. Transactions on Software Engineering, 4(3), IEEE, pp. 178–187 (1978)
14. Vuong, S.T., Ko, K.C.: A novel approach to protocol test sequence generation. Global Telecommunications Conference 3, IEEE, pp. 2 -5 (1990)
15. Godskesen, J.C.: Fault models for embedded systems. In: Proceedings of 10th IFIP Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Springer Berlin Heidelberg, pp. 356-359 (1999)
16. Cheng, K.-T., Jou, J.-Y.: A functional fault model for sequential machines. Transactions on Computer-Aided Design of Integrated Circuits and Systems 11(9), pp. 1065-1073 IEEE (1992)
17. Petrenko, A., Boroday, S., Groz, R.: Confirming configurations in EFSM testing. Transactions on Software Engineering 30(1), IEEE, pp. 29-42 (2004).
18. Gordon, F., Wotawa, F. Ammann, P.: Testing with model checkers: a survey. Software Testing Verification and Reliability 19.3, pp. 215-261 (2009)
19. Anand, S. et al.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software 86.8, pp. 1978-2001 (2013)
20. Petrenko, A. Fault model-driven test derivation from finite state models: Annotated bibliography. Modeling and verification of parallel processes, Springer Berlin Heidelberg, pp. 196-205 (2001)
21. Petrenko, A., Bochmann, Gv, Yao, M.: On fault coverage of tests for finite state specifications. Computer Networks and ISDN Systems 29.1, pp. 81-106 (1996)
22. Simao, A., Petrenko, A. Maldonado, J. C.: Comparing finite state machine test coverage criteria. IET Software 3 (2), pp. 91-105 (2009)
23. De Moura, L., Dutertre B.: Yices 1.0: An efficient SMT solver. The Satisfiability Modulo Theories Competition (SMT-COMP) (2006)
24. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg. pp. 337-340 (2008)
25. Rösch, S., Ulewicz, S., Provost, J. Vogel-Heuser, B.: Review of model-based testing approaches in production automation and adjacent domains - current challenges and research gaps. Journal of Software Engineering and Applications 8, pp. 499-519 (2015)
26. Bochmann, G. V., et al.: Fault models in testing. In Proceedings of the IFIP TC6/WG6. 1 Fourth International Workshop on Protocol Test Systems. North-Holland Publishing Co, pp. 17-30 (1991)
27. Petrenko, A., Yevtushenko, N.: Testing from partial deterministic FSM specifications. Transactions on Computers 54(9), pp. 1154-1165. IEEE (2005)
28. Korel, B., Tahat, L.H., Harman M.: Test prioritization using system models. In Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 559-568 (2005)
29. Petrenko, A., Dury, A., Ramesh, S., Mohalik, S.: A method and tool for test optimization for automotive controllers. In ICST Workshops, pp. 198–207 (2013)
30. Parr, T.: The definitive ANTLR 4 reference (2nd edition.). Pragmatic Bookshelf (2013)