Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 190 (2007) 3–19
www.elsevier.com/locate/entcs

Can a Model Checker Generate Tests for Non-Deterministic Systems?

Sergiy Boroday[1], Alexandre Petrenko[2]

CRIM, Centre de recherche informatique de Montréal
550 rue Sherbrooke Ouest, Bureau 100, Montréal, H3A 1B9, Canada

Roland Groz[3]

Institut National Polytechnique de Grenoble
F-38402 St Martin d'Hères, Cedex, France

Abstract

Modern software is increasingly concurrent, timed, distributed, and therefore, non-deterministic. While it is well known that tests can be generated as LTL or CTL model checker counterexamples, we argue that non-determinism creates difficulties that need to be resolved and propose test generation methods to overcome them. The proposed methods rely on fault modeling by mutation and use conventional (closed) and modular (open) model checkers.

Keywords: Testing, Software, Black Box, Test Generation, Verification, Model Checking, Module Checking.

1 Introduction

Test generation from deterministic specifications using verification techniques and tools is a well-known approach. It is suggested by [44] and refined by several authors, e.g., [13], [37], [8], [34], [36], [25], [20]. However, modern systems are non-deterministic due to asynchrony, concurrency, multithreading, timing issues, non-observable or non-controllable elements. Moreover, even if an implementation under test (IUT) itself can be regarded as deterministic, in a model or specification, non-determinism may occur due to incompleteness of knowledge of implementation choices, limitations of a modeling language, and abstractions. Conservative model

[1] Email: Sergiy.Boroday@crim.ca
[2] Email: Alexandre.Petrenko@crim.ca
[3] Email: Roland.Groz@imag.fr

1571-0661/$ – see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2007.08.002

abstraction is widely used to reduce complexity (state space) or to remove constructs, which are difficult for simulation and verification (e.g., time aspects [1]). In declarative object oriented conceptual modeling, non-determinism allows to better reflect inherent non-determinism of the domain, reduce complexity, and achieve a better separation of concerns [4]. The problem of coping with non-determinism is a long-standing one in protocol testing [10], [16], [17], [18], [19], [21], [35].

We present several approaches to test generation for non-deterministic specifications. We argue that a non-deterministic specification/implementation poses certain difficulties, especially, when one targets not only weaker tests that allow for inconclusive verdicts, but also tests that deliver definitive and conclusive results no matter which system's branch is executed. It is well known that derivation of tests for nondeterministic models is more computationally difficult or even impossible [1]. Moreover, we demonstrate in this paper that naïve approaches for coping with non-determinism could even simply fail, i.e., lead to inconsistent results. We define two different types of tests (weak and strong) which coincide in the case of deterministic specifications. A weak test is usually associated with some assumptions, e.g., if a test is executed sufficiently often, all possible reactions of a non-deterministic IUT are observed [29], [31]. Unfortunately, such assumptions do not apply to the case when non-determinism occurs in the specification due to conservative (or existential) abstraction. Thus we pay special attention to derivation of strong tests, also known as separating sequences in the context of FSM (Finite State Machine) testing [43].

Non-determinism could pose some problems even for white-box testing [24]. However, here we target the case of black-box (or functional) testing, when additional difficulties occur due to lack of information on implementation details and limited observation. Only input and output variables are observable. The state and hidden (internal) variables of the IUT are not observable, so additional efforts to propagate faults to observable errors are required.

Among various test generation approaches we favor in this paper mutant-based testing, which is one of the most complex, but promising strategies to detect faults and fight the state explosion [34], [9]. Unfortunately, this technique is often associated with a so-called mutant explosion. While, traditionally, mostly deterministic mutants are considered, in [6] mutant-based test generation is extended for non-deterministic mutants. Introducing non-deterministic mutants alleviates the problem of mutant explosion and leads to a more general testing framework. Previously [36], we applied model checking to generate a confirming sequence, kind of a partial Unique Input Output sequence (UIO) [11], often used in FSM based testing and related to state identification and checking problems. Mutant-based technique could be applied to generate UIO using a mutation operator that changes the set of initial states to states which are not initial in the specification [8] (at least for specification with no equivalent states) [40].

In the context of mutation-based testing, black-box testing is also sometimes referred to as propagating faults to output. While the white-box testing (specification coverage) is often seen as a totally different testing paradigm, there exists

a connection between the two types of testing. In fact, to kill (detect) a mutant obtained simply by mutating an output, a test that covers the affected transition (or, in the case of Moore machines or Kripke structures, a state) is necessary and at the same time sufficient. The problem of finding mutant-killing tests can be reduced to the problem of reaching states of a module composed of a specification and faulty sub-modules which satisfy a given property. This approach could be used to derive tests that combine specification coverage and fault propagation [36], [6]. However, those interested in coverage based techniques could find extensive literature elaborating usage of model checking tools for particular data or control flow coverage criteria, such as [13], [24], [39], [14].

For FSM, deterministic as well as non-deterministic, test generation is a well studied theoretical problem, e.g., [22], [38], [35], [1]. However, methods developed for classical FSM are rarely applied for real size specifications due to state explosion problem. The current trend is to use model checking [12] technology, which embraces various sophisticated optimization techniques to cope with state explosion problem, such as BDD, partial orders, and SAT.

While previously we studied the problem of test derivation for a communicating extended FSM (CEFSM) model [6], now we cast the problem in the framework of the model checking theory which is traditionally based on Kripke structures and modules. In this paper, we generalize and further elaborate our and previously known results for model checking driven test generation for the case of non-deterministic specifications and implementations (mutants). Two types of tests, called strong and weak, are distinguished. Some properties of these tests, such as their relation to fairness are established. For the most general and difficult case of non-determinism, a novel test generation approach, based on modular model checking (module checking) of a composition of the mutant and specification is proposed. An incremental test generation approach that involves observers (transformed specifications or mutants) and traditional model checking is also outlined. Several special cases of test generation which could be resolved with traditional model checking are discussed. Counterexamples are built to demonstrate where naïve approaches to test generation with model checkers fail. The definitions and results are cast in a formal setting, based on the definition of a module which is often used in recent studies on validation of component based and modular software.

The paper is organized as follows. The next section introduces necessary definitions of test, module, model checking, and module checking. Section 3 discusses test generation from a counterexample derived by model or module checking in the presence of non-determinism. Section 4 discusses how our results apply to the case of multiple mutants. In Section 5, we briefly discuss some related work and conclude in Section 6.

2 Preliminaries

Here we introduce the necessary notions and notations. Unlike most of FSM testing literature, which is usually based on Mealy machines [29], our work is based on the

notion of a module, which could be seen as a Kripke structure with a partition of atomic propositions onto input, output, and internal (hidden) [27]. In Mealy machines, differently from Kripke structures, the labels are assigned to transitions and not to states. In some cases, Mealy machines allow for more intuitive and compact specifications than Kripke structures, especially in the black-box testing context. However, our choice is motivated by the fact that temporal logics, used in model checking, are traditionally defined over Kripke structures. As we later show, the input-output behavior of a module could be modeled by a Mealy machine. In the presence of hidden variables, the model of extended finite state machine [36] could be used to obtain a compact representation of the module.

2.1 Model Checking

A *Kripke structure* is a tuple $Kr = (AP, W, R, W_0, L)$, where

- AP is the set of atomic propositions;
- W is the set of states;
- $R \subseteq W \times W$ is the transition relation;
- $W_0 \subseteq W$ is the set of initial states;
- $L: W \to 2^{AP}$ is the labeling function which maps each state into a set of atomic propositions that hold in this state.

For $(w, w') \in R$, we say that w' is a *successor* of w. We say that a Kripke structure is *deadlock-free* if every state w has at least one successor.

An infinite sequence of state successors is called a *path*. A path, starting from an initial state is called an *execution* path. A path is called *fair* [42] if for each state that occurs infinitely often in the path each outgoing transition is taken infinitely often. Usually, model checking deals only with infinite sequences. A work-around to deal with finite executions is to infinitely repeat the last state.

In this paper, atomic propositions are also seen as Boolean variables, which valuate to 1 (true) when corresponding propositions hold in the state and to 0 (false) otherwise.

Hereafter, we deal only with usual CTL syntax and semantics over deadlock-free Kripke structures [15], [27]. Temporal logics extend the usual propositional logic with temporal combinators Finally (eventually), Globally (universally), Until, and path quantifiers All and Exists [15]. Beside standard conjunction, disjunction, and negation we use a logic equality combinator, denoted $\varphi \leftrightarrow \psi$, or simply $\varphi = \psi$.

Formulas, where each combinator F, G, U is immediately preceded by either quantifier A or quantifier E, constitute a temporal logic CTL, often supported by model checkers.

A model checking problem consists in checking whether a Kripke structure Kr satisfies a formula φ in all the initial states, denoted $Kr \models \varphi$. A counterexample is a path of the Kripke structure, path prefix, or set of paths (in case of a complicated CTL property) that causes formula violation. Most model checkers report one or several minimum counterexamples.

2.2 Modular Specifications

A *composition* of two Kripke structures $Kr = (AP, W, R, W_0, L)$ and $Kr' = (AP', W', R', W'_0, L')$ is a Kripke structure $Kr \parallel Kr' = (AP'', W'', R'', W''_0, L'')$, where

- $AP'' = AP \cup AP'$;
- $W'' = \{(w, w') \colon L(w) \cap AP' = L'(w') \cap AP\}$;
- $R'' = \{((w, w')(s, s')) \colon (w, s) \in R, (w', s') \in R'\} \cap W''$;
- $W''_0 = (W_0 \times W'_0) \cap W''$;
- $L''(w, w') = L(w) \cup L(w')$ for $(w, w') \in W''$.

Thus, the composition synchronizes over state labels shared by the components. In other words, each state of the composition is composed of state of the first component and state of the second component, such that each atomic proposition that belongs to both Kripke structures is present or absent in both states simultaneously.

A module is a triple (Kr, I, O), where $Kr = (AP, W, R, W_0, L)$ is a Kripke structure, and $I, O \subseteq AP$ are disjoint sets of *input* and *output variables*. $H = AP \setminus (I \cup O)$ is called the set of *hidden* (internal) variables. While hidden variables may appear redundant, we need them here for technical reasons. A module is *closed*, if $I = \emptyset$, otherwise it is *open*. Intuitively, in every state w, the module reads $L(w) \cap I$, stores internally $L(w) \cap H$, and outputs $L(w) \cap O$. $Inp(w) = L(w) \cap I$ is called the *input* of the module in the state w. $Out(w) = L(w) \cap O$ is called the *output* of the module in the state w.

A module is called *deterministic* if for each state w and each $i \subseteq I$, w has at most one successor state s with the input $Inp(s) = i$, moreover, for all w, $s \in W_0$ $Inp(w) = Inp(s)$ implies $w = s$. A module is *non-deterministic*, otherwise.

Given a sequence $w_1, w_2, \ldots, w_k$ of successor states of a module (Kr, I, O), starting from an initial state, we say that (Kr, I, O) *produces* an output sequence $Out(w_1)Out(w_2)\ldots Out(w_k)$ in response to an input sequence $Inp(w_1)Inp(w_2)\ldots Inp(w_k)$, while $Inp(w_1)Out(w_1)Inp(w_2)Out(w_2)\ldots Inp(w_k)Out(w_k)$ is called an *input-output sequence*.

Similar to Mealy machines, a module (Kr, I, O) is called *observable* [43], if the module $(Kr, I \cup O, \emptyset)$, obtained from the initial module by moving all the output variables into the input variable set, is deterministic. Otherwise, the module (Kr, I, O) is called *non-observable*. Intuitively, observable non-determinism is a simple form of non-determinism, when a path, taken by the module, could be deduced from the observed input-output sequence. Non-observable non-determinism is harder to deal with. Fortunately, for each non-observable module, an observable module with the same set of input-output sequences exists. Such module can be constructed by a well known powerset construction procedure.

Input i is *enabled* in the state w if w has at least one successor state s with input $Inp(s) = i$. Otherwise, input is *disabled*. A module is *input enabled* (completely defined) if each input labels an initial state and is enabled in every state. In this

paper, we consider only input enabled specification and mutant modules, and, hence, deadlock-free.

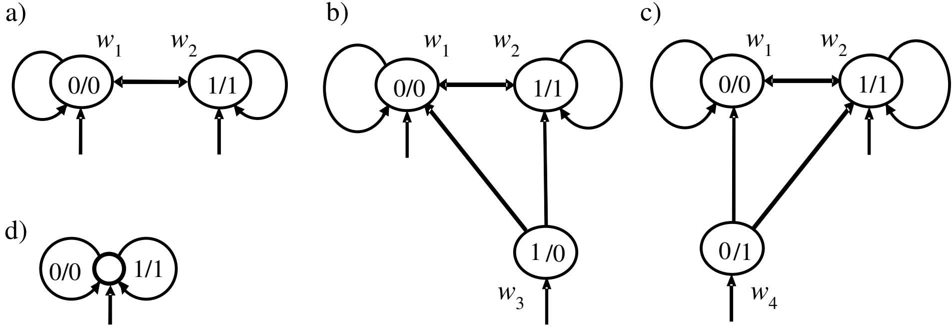

Fig. 1. a) The specification; b) and c) mutants; d) Mealy machine.

A module, which constitutes our running example of a specification is shown in Fig. 1a. For simplicity, values of the input variable i and the output variable o are depicted as pairs of Boolean values. Each state has a unique label. The specification has two states, w_1 labeled with 0/0, and w_2 labeled with 1/1, both states are initial. The specification can also be represented as a Mealy machine (Fig. 1d). The transition relations of all the modules are defined in such way that states, where $i = o = 0$ or $i = o = 1$, and only them, are successors of all the other states. All three modules in Fig. 1 are input enabled.

A *composition* of the modules $M = (Kr^M, I^M, O^M)$ and $S = (Kr^S, I^S, O^S)$, such that no hidden variable of one module is a variable of another ($AP^M \cap H^S = AP^S \cap H^M = \emptyset$), and sets of output variables are disjoint ($O^S \cap O^M = \emptyset$), is $M \parallel S = (Kr^M \parallel Kr^S, (I^M \cup I^S) \setminus (O^M \cup O^S), (O^M \cup O^S))$. If needed, output and hidden variables are renamed for the composition. Note that our definition allows common inputs in the module composition.

A union of the modules $M = (Kr^M, I^M, O^M)$, where $Kr^M = (AP^M, W^M, R^M, W_0^M, L^M)$ and $S = (Kr^S, I^S, O^S)$, where $Kr^S = (AP^S, W^S, R^S, W_0^S, L^S)$ with mutually exclusive state sets is the module $M \cup S = ((AP^M \cup AP^S, W^M \cup W^S, R^M \cup R^S, W_0^M \cup W_0^S, L), I^M \cup I^S, O^M \cup O^S)$, where $L(w) = L^M(w)$ if $w \in W^M$, and $L(w) = L^S(w)$ if $w \in W^S$. If the state sets intersect, states could be renamed. One can easily see that the set of input-output sequences of the union is the union of sets of input-output sequences of the original modules.

Model checking of specifications with infinite state space is limited to small to medium specifications and undecidable in the most general case. Thus, hereafter, we consider only modules with finite number of states, variables, and propositions.

2.3 Module Checking

A module satisfies φ if the Kripke structure of the module satisfies φ. However, a formula (property) that holds on a module in isolation does not always hold when the module is embedded into a system of communicating modules. Often it

is important that a property is satisfied when the module reactively communicates with other (possibly unknown) modules. To address this problem, a new technique, more general than conventional model checking, is introduced [27].

A module M *reactively satisfies* φ, denoted $M \models_r \varphi$, if $E \parallel M \models \varphi$ for all deadlock-free modules $E \parallel M$, where E is a module such that no hidden variables of one module is a variable of another module, and no output variable of one module is an output variable of another module. The module E represents a possible environment that does not block the module M. Checking reactive satisfaction of a formula constitutes the problem of modular model checking with incomplete information or *module checking*, for short [27]. For LTL, module checking coincides with model checking.

2.4 Testing

We consider here mutation-based test generation, where faults are modeled using the same formalism as a specification, and mutants are obtained from the specification by applying mutation operations. While mutation is traditionally performed on code, recently, a variety of specification mutation operators is suggested [9], [34].

Let S be a module that models a specification and M be a module that models a faulty implementation (mutant) that share the same sets of input and output variables.

A finite sequence of inputs $i_1 \dots i_k$, where $i_j \subseteq I$, is called a *weak test* for S and M if in response to it M produces an output sequence that S cannot.

A finite sequence of inputs $i_1 \dots i_k$ is called a *strong test* for S and M if any output sequence M produces in response to $i_1 \dots i_k$ cannot be produced by S.

While, in certain cases, it might be more efficient to use a test suite or adaptive (on-the-fly) tests instead of simple tests defined above, for simplicity, in this work, we target only single preset tests.

If S and M are deterministic, the notions of strong and weak tests coincide; in the non-deterministic case, a weak test may exist even when there is no strong test.

2.4.1 Weak Tests and Fairness

Here we show that, under fairness assumption, repeatedly executed (a sufficient number of times) weak test reveals a fault. In order to be able to repeat a test, a reliable reset to set the module into all its initial states is needed. Such a reliable reset could be modeled as follows. Let S_{reset} and M_{reset} be modules obtained from the original modules by adding a *reset* input variable to both S and M. Each original initial state is replaced by a reset state with the same variables along with the designated variable *reset* and the same set of successors. Each reset state is a successor of all states. Let $(i_1 \dots i_k)^\omega$ be a module, which infinitely often repeats $i_1 i_2 \dots i_k$ as its output (which, in our case, become inputs for other modules). The following proposition holds.

Proposition 2.1 *If $i_1 \dots i_k$ is a weak test for S and M, then each fair execution path of $M_{reset} \parallel (i_1 \dots i_k\{reset\})^\omega$ and each fair execution path of $S_{reset} \parallel$*

$(i_1 \dots i_k\{reset\})^\omega$ *contain different output sequences.*

Obviously, no weak test exists, if and only if each input-output sequence of the mutant is also an input-output sequence of the specification. In case of a deterministic specification, a weak test does not exist if and only if the specification and mutant have the same set of input-output sequences.

2.4.2 Strong Tests

If there exists a strong test for modules S and M, these two modules are called *separable* (similar to non-deterministic FSM, see, e.g., [43]). In case of a deterministic specification, a strong test does not exist if and only if each input-output sequence of the specification is also an input-output sequence of the faulty module.

A strong test is symmetric: each strong test for a pair (S, M) is also a strong test for (M, S), i.e., it is not important which of the two is the specification or the mutant.

2.4.3 Complexity Issues and Justification for Modular Model Checking

Derivation of strong tests and its complexity is discussed by Alur et al [1], where strong tests for two modules are referred as preset distinguishing strategies for two machines. A preset distinguishing strategy is seen as a winning strategy for a (blindfold) $\exists\forall$ game with incomplete information. Since such games are PSpace complete, the same is true for the preset distinguishing strategy existence problem, and the length of a preset distinguishing sequence is exponential (in the worst case). The proposed method for the preset distinguishing strategy generation is based on derivation of a special type of a power set module of exponential size that resembles Gill's successor tree [22]. Thus, such an automaton could be constructed and fed to a model checker. However, model checking technology still may face difficulties with large space state and specification size. Thus, we try to build a smaller specification, applying a game-theoretic model checking known as module checking.

At the same time, we discuss possible use of more traditional and widespread LTL or CTL model checking techniques for simpler cases of the most general problem. Moreover, we show that candidate test verification and, hence, incremental test derivation is possible for non-deterministic specifications and mutants. Yet, we doubt that one could use conventional LTL or (non-modular) CTL model checking technique to derive strong test or decide its existence without significant computational efforts on the model transformation. An exponential growth of the size of the system or property being model checked is expected. Indeed, the complexity of CTL model checking is linear in terms of state space, while finding strong tests is PSpace-complete.

3 Test Generation by Model and Module Checking

Test generation for a deterministic specification and deterministic mutant modules is well understood. However, we first cast this problem in our framework to generalize it later for non-deterministic cases.

3.1 Deterministic Case

For the deterministic case we define a system that consists of a specification and mutant modules. Then we model check a formula that states equality of the outputs of specification and mutant modules, so a counterexample will give a test. In our framework, this simple idea could be formalized as follows.

In order to compose modules with common hidden and output variables $O \cup H$, we introduce a renaming operator. Formally, the renaming operator $'$ is defined on hidden and output variables of a module: $(p)' = p'$, where p' is not in the set of the atomic propositions of these modules. We lift the operator to sets of variables and modules. Also, let $p = p'$ hold in a state when both atomic propositions p and p' are simultaneously hold or do not hold in this state.

Theorem 3.1 *Let $S = (Kr^S, I, O)$ and $M = (Kr^M, I, O)$ be two deterministic modules with the same input and output variables.*

$$S \parallel M' \models \mathrm{AG} \bigwedge_{p \in O} (p = p')$$

if and only if there is no (strong or weak) test.

$\bigwedge_{p \in O}(p = p')$ is a shorthand for $o_1 = o'_1 \wedge o_2 = o'_2 \wedge \ldots \wedge o_{|O|} = o'_{|O|}$, which formally denotes the equality of outputs of S and M (While a more elegant formula, $Out' = (Out)'$ could be used instead; some readers may find it confusing.)

The idea of the proof is as follows. The set of output variables of the composition $S \parallel M'$ is $O \cup O'$. The composition is synchronized only by input in the sense that for each pair of execution paths of modules S and M', that share the same sequence of inputs, pairs of states form an execution path of the composition. Each execution path of composition is a sequence of pair of states of execution paths of S and M', that correspond to the same input sequences. The logic formula simply states that each output variable $o \in O$ always valuates as the corresponding renamed variable o' on each path of the composition. That is the output of M coincides with the output of S for each input sequence, which is a necessary and sufficient condition for the absence of strong test for two modules.

Note that formula in Theorem 3.1 belongs both to CTL and LTL. Since LTL module checking coincides with model checking, $S \parallel M' \models \mathrm{AG} \bigwedge_{p \in O}(p = p')$ is equivalent to $S \parallel M' \models_r \mathrm{AG} \bigwedge_{p \in O}(p = p')$.

An input sequence defined by a (finite) counterexample to the expression in Theorem 3.1 constitutes a strong test case. This fact is known and widely exploited. Indeed, the set of such input sequences, defined by counterexamples, and the set of strong tests coincide.

Example. To illustrate Theorem 3.1, we consider the modules shown in Fig. 1a and Fig. 1b. While the specification always reacts to input $i = 1$ with output $o = 1$, the mutant starts with output $o = 0$. Thus, $S \parallel M' \models \mathrm{AG}(o = o')$ does not hold. $i = 1$ is both the shortest counterexample and the shortest test.

3.2 Non-Deterministic Mutant

3.2.1 Weak Test Derivation

Even if a specification is deterministic, a mutant may still exhibit some non-determinism, e.g., related to data races or abstraction. This is why it is interesting to consider the case of a deterministic specification and non-deterministic mutants.

Theorem 3.2 *Let $S = (Kr^S, I, O)$ and $M = (Kr^M, I, O)$ be two modules with the same input and output variables, where S is deterministic, while M is possibly non-deterministic.*

$$S \parallel M' \models \mathrm{AG} \bigwedge_{p \in O} (p = p')$$

if and only if there is no weak test for S and M'.

As we noted before, $S \parallel M' \models \mathrm{AG} \bigwedge_{p\in O}(p = p')$ is equivalent to $S \parallel M' \models_r \mathrm{AG} \bigwedge_{p\in O}(p = p')$.

An input sequence defined by a finite counterexample to the above formula could serve as a weak test.

Example 3.3 Consider the specification in Fig. 1a and the mutant as in Fig. 1b, but with all three states being initial. $i = 1$ is a weak test, since the mutant can produce the output $o = 0$, which the specification cannot produce in response to the input $i = 1$. At the same time, the mutant can produce output $o = 1$, as the specification.

Determinism of the specification is essential. In the case when both modules are non-deterministic, the formula $\mathrm{AG} \bigwedge_{p\in O}(p = p')$ could be violated even when no strong or weak test exists. For example, let both, specification and faulty, modules be as in Fig. 1b, but with all states being initial. Thus, both modules are non-deterministic and coincide. The formula of Theorem 3.2 does not hold, since there exists an execution path, such that $o \neq o'$ already in the first state (w_2, w_3), though no weak test exists.

3.2.2 Strong Test Generation

In order to determine a strong test, we first replace a non-observable mutant by an observable one with the same set of input-output sequences. The following proposition states that this transformation does not affect test existence.

Proposition 3.4 *If M_1 and M_2 have the same set of input-output sequences then each strong test for S and M_1 is also a strong test for S and M_2.*

The idea behind strong test derivation relies on transforming an observable mutant module into an observer [23]. We build a module Obs(M) for a given module M by the following sequence of transformations. Each output of the original module becomes an input of the new module. A hidden variable *found* is defined, thus, if present, original hidden variables are removed. If needed, determinization is performed by powerset construction. In all non-trivial cases, the obtained module is

not input enabled due to inputs, obtained from the output set O^M of the original module M. The module is completed to an input enabled module with a set of additional sink states. The variable *found* valuates true only in these states. More formally, for each state and each disabled input i, a successor sink state, labeled with $i \cup \{found\}$ is added. For each input i that does not label an initial state, an additional initial state labeled with $i \cup \{found\}$ is defined. Each of the added states is a successor of all the other added states.

Note that determinization is required only for non-observable module M; otherwise, the obtained module is deterministic due to performed inversion of outputs into inputs. Possibly, approximate conservative determinization [41] could be applied.

The following holds for the obtained modules.

Theorem 3.5 *Let $S = (Kr^S, I, O)$ and $M = (Kr^M, I, O)$ be two modules with the same input and output variables, where S is deterministic, while M is possibly non-deterministic.*

$$S \parallel \mathrm{Obs}(M) \models \mathrm{AG}\,\overline{found}$$

if and only if there is no strong test.

Note that, since LTL module checking coincides with model checking, $S \parallel \mathrm{Obs}(M) \models \mathrm{AG}\,\overline{found}$ is equivalent to $S \parallel \mathrm{Obs}(M) \models_r \mathrm{AG}\,\overline{found}$.

3.3 Non-Deterministic Modules

3.3.1 Weak Test

A weak test can also be generated using a composition of a module with an observer. However, the observer should be built from the specification, rather than from the mutant.

Theorem 3.6 *Let $S = (Kr^S, I, O)$ and $M = (Kr^M, I, O)$ be two modules with the same input and output variables.*

$$M \parallel \mathrm{Obs}(S) \models \mathrm{AG}\,\overline{found}$$

if and only if there is no weak test case for S and M.

Note that, since LTL module checking coincides with model checking, $M \parallel \mathrm{Obs}(S) \models \mathrm{AG}\,\overline{found}$ is equivalent to $M \parallel \mathrm{Obs}(S) \models_r \mathrm{AG}\,\overline{found}$.

3.3.2 Strong Test Verification

The case of strong test derivation when both specification and mutants are non-deterministic is the most complex among those considered here, one may use approximation or ad-hoc methods to solve it. In this case, a verification procedure may be needed. Thus, we discuss how to check whether a candidate input sequence constitutes a strong test.

In order to model check whether a given candidate input sequence is a test, we define a tester module, called Tester, which simply feeds a given sequence to the specification and mutants. Tester(α) for a given input sequence $\alpha = i_1 \dots i_k$, is a module with the empty set of input variables, a hidden variable (flag) h, the set of states $\{w_0, w_1, \dots, w_k\}$, and transition relation $\{(w_j, w_{j+1})\colon 0 \leq j \leq k-1\} \cup \{(w_k, w_k)\}$, the labeling function L such that $L(w_j) = i_{j+1}$ for $0 \leq j \leq k-1$, and $L(w_k) = \{h\}$. The loop (w_k, w_k) and a flag h are needed, because in our framework, inspired from [27] model checking (as well as module checking) is defined only on deadlock-free modules.

Theorem 3.7 *Let $S = (Kr^S, I, O)$ and $M = (Kr^M, I, O)$ be two modules with the same input and output variables and α be an input sequence.*

$$S \parallel M' \parallel \text{Tester}(\alpha) \models \text{AF}\, \overline{h \vee \bigwedge_{p \in O} (p = p')}$$

if and only if the input sequence α is a strong test.

The idea of the proof is as follows. The formula formalizes the intuition that a test is strong if it guarantees that in the course of test execution, a mutant, sooner or later (eventually), produces an output, different from any output that the specification is capable of.

Note that $S \parallel M' \parallel \text{Tester}(\alpha) \models_r \text{AF}\, \overline{h \vee \bigwedge_{p \in O}(p = p')}$ is equivalent to $S \parallel M' \parallel \text{Tester}(\alpha) \models \text{AF}\, \overline{h \vee \bigwedge_{p \in O}(p = p')}$.

For most of complete model checking algorithms and tools, which compose modules prior to property analysis, replacing M' by Obs(M) may somewhat reduce space state, but the gain is relatively small for on-the-fly model checking, when only a fragment of the composition, which is relevant to the property, is usually constructed. An incremental generation of a strong test can be performed by consecutive verification of all candidates of a given length. If the state number of modules is finite, an upper bound of the test is known [1]. Technically, it is possible to define a module that consecutively tries all possible test candidates of the given length. Such approach could be faster than model checking multiple systems, but we do not see how it could be organized efficiently in the terms of the memory consumption.

3.3.3 Strong Test Derivation by Module Checking

To derive a strong test for the most general case, when both specification and mutant are non-deterministic, we introduce two auxiliary operators. One operator HideOut$(Kr, I, O) = (Kr, I, \emptyset)$ is a blindfold operator. Intuitively, the blindfold operator preserves the structure and inputs of the original module, but hides output variables from the environment, by placing them into the set of hidden variables of the resulting module. Another additional operator [1] AddIn, which adds a new, single initial state w_0, such that $L(w_0) = \emptyset$, and transitions from w_0 lead to all the former initial states (and only to them).

[1] This operator is needed only for non-deterministic specifications or mutants, which have several different initial states that share same input.

Theorem 3.8 *Let $S = (Kr^S, I, O)$ and $M = (Kr^M, I, O)$ be two modules with the same input and output variables.*

$$\mathrm{HideOut}(\mathrm{AddIn}(S \parallel M')) \models_r \mathrm{EG} \bigwedge_{p \in O} (p = p')$$

if and only if there is no strong test for S and M.

The intuition behind this statement is that two modules produce at least one common output sequence for any input sequence if and only if there is no strong test for these two modules. Input sequences are produced by a module checking environments. The condition of a blind environment is needed since here we are not interested in adaptive testers and testers that could prevent (block) certain outputs. Blindness of environment ensures that the formula is reactively satisfied if and only if it is satisfied for all the possible environments each state of which has a single successor. Thus, multiple tests are not addressed. Therefore, the reactive satisfaction of the formula of Theorem 3.8 on the blindfolded composition of the specification with the mutant formalizes the above necessary and sufficient condition of strong test non-existence. Since module checking allows for both, output sensitive (adaptive) and blind, environments, we hide the outputs of the composition with a designated operator HideOut. Note that O in the formula does not refer to the set of output variables of $\mathrm{HideOut}(\mathrm{AddIn}(S \parallel M'))$. O denotes the set of the output variables of the original module $S = (Kr^S, I, O)$. In $\mathrm{HideOut}(\mathrm{AddIn}(S \parallel M'))$ these variables are hidden.

While, unlike the previous cases, a minimum counterexample may involve several execution paths, it will provide a unique finite input sequence. To obtain a strong test, though, one should disregard the first (empty) input of this sequence.

Note, since the formula uses path existence quantification, it does not belong to any universal logic. In this case, replacing reactive satisfaction by usual would change the meaning of the formula. In the context of the conventional model checking, formula of Theorem 3.8 would state that each sufficiently long sequence of inputs constitutes a strong test for given S and M. Possibly, in (conventional) model checking setting, a specific for M (but still independent from S) non-separability condition could be expressed in the form of the so-called tree automaton [26], but we doubt that this is always possible in CTL.

4 Multiple Mutants

Here we discuss derivation of a test which targets several mutants. The problem could be reduced to the case of a single mutant by merging a set of mutants $\{M_1, \ldots, M_l\}$ into a single module $M_1 \cup \ldots \cup M_l$.

Proposition 4.1 *An input sequence is a strong test for S and the module $M_1 \cup \ldots \cup M_l$ if and only if it is a strong test for S and each of the modules $M_1, \ldots, M_l$.*

The proposition does not hold for weak tests.

Such transformation of several mutants into meta-mutant should be considered with care, since a test for the specification and the meta-mutant may not exist even if there exists a single test for each mutant. Consider the following example.

Example. Consider the specification in Fig. 1a, and mutants in Fig. 1b and Fig. 1c. The first mutant has an additional state w_3 labeled with 1/0, the initial states are w_1 and w_3. The second mutant has an additional state w_4 labeled with 0/1, the initial states are w_2 and w_4. Any strong test for the specification and the first mutant should start with $i = 1$. Similarly, any strong test for the specification and the second mutant should start with $i = 0$. Thus, there is no single test for the specification and both mutants at the same time.

A finer approach is to build a composition of the specification with all the mutants. However, outputs of each of them should be renamed. Unlike the case of the merged modules, for such a multi-component composition, the testing remains tractable even if there is no input sequence that is a strong (weak) test with respect to all the mutants. For example, the problem of deriving a test which is strong for a maximal number of mutants could be considered, similar to a confirming sequence, a kind of partial UIO, generation [37]. Such an optimization problem may look alien to classical model checking, but many model checking tools, such as ObjectGeode [33], provide means to collect statistics. Possibly, certain metrics, which assign a weight for each mutant, could be implemented.

5 Related Work

As far as we know, reuse of verification techniques for test generation is first suggested in [44], though, in our opinion, treatment on non-determinism is rather rudimentary in this work. In order to cope with non-determinism in a specification, [34] suggests synchronizing non-deterministic choices, i.e., in value of variables that are updated in a non-deterministic fashion (for the case when each state is uniquely defined by the values of variables). In terms of SMV language, it consists in declaring a variable global and removing update operations from a mutant. The approach, as suggested, works in simple cases. However, it is not clear how it applies to the most general case. For example, we consider the case when one hidden variable is updated in a non-deterministic fashion, but its value is revealed via an output variable only in a next step. The approach of [34] could result in the false tests for mutations that negate the output value in that step. Our observer based test generation could be seen as a more general version of the method sketched in [34].

Some obstacles in test generation for tricky coverage criteria from CTL model checking counterexample are resolved by adding an additional logic operator [24]. In this paper, instead of applying a rare temporal logic, we cope with non-determinism using module checking.

6 Conclusion

We discussed difficulties in test generation for non-deterministic systems not only using complexity-theoretical arguments, but also by demonstrating how naïve approaches may fail and proposed several solutions, which allow one to use a modular or conventional CTL (and, sometimes, LTL) model checker to generate tests automatically. As usually, counterexamples, generated by a model checker, could be used to build tests. We demonstrated that in the most general case of non-deterministic specifications and implementations, the existence of a test could be verified using the module checking approach. Alternatively, an incremental approach, where each candidate test is verified using a conventional model checking is proposed. While the incremental approach may appear inefficient, our preliminary experiments [7] give encouraging results. Moreover, incremental approach could rely on a larger range of tools and make possible the use of efficient techniques of bounded model checking [5], when the behavior of the specification is analyzed only on executions of a given length.

Our future plans include test derivation experiments with module checking and contingent AI planning. In view of a currently rather limited choice of module checking tools, we hope that our ideas of using module checking for test derivation could motivate development of new algorithms and tools.

References

[1] Alur, R., C. Courcoubetis, M. Yannakakis, *Distinguishing Tests for Nondeterministic and Probabilistic Machines*, in: 27th ACM Symp. on Theory of Computing (1995), 363–372.

[2] Alur, R., and D. Dill, *Automata for Modeling Real-Time Systems*, in: ICALP (1990), 322–335.

[3] Alur, R., and T. Henzinger, *Reactive Modules*, in: 11th LICS (1996), 207–218.

[4] Bekaert, P., and E. Steegmans, *Non-determinism in Conceptual Models*, in: Tenth OOPSLA Workshop on Behavioral Semantics (2001), 24–34.

[5] Biere, A., A. Cimatti, Clarke, E. M., O. Strichman, Y. Zhu., *Bounded Model Checking*, Advances in Computers **58** (2003), Academic Press, 118–149.

[6] Boroday, S., R. Groz, A. Petrenko, Y.-M. Quemener, *Techniques for Abstracting SDL Specifications*, in: 3rd SAM (2003), 141-157.

[7] Boroday, S., A. Petrenko, R. Groz, *Test Generation for Non-Deterministic Systems with Model and Module Checkers*, Technical Report CRIM-07/0202, Montreal, 2007.

[8] Boroday, S., A. Petrenko, R. Groz, Y.-M. Quemener, *Test Generation for CEFSM Combining Specification and Fault Coverage*, in: XIV TestCom (2002), 355–372.

[9] Camargo, S., P. Fabbri, J. Maldonado, P. Masiero, M. Delamaro, *Proteum/FSM: A Tool to Support Finite State Machine Validation Based on Mutation Testing*, in: SCCC (1999), 96–104.

[10] Cheung, T. Y., Y. Wu, X. Ye, *Generating Test Sequences and their Degrees of Nondeterminism for Protocols*, in: 11th Int. Symp. Protocol Specification, Testing, and Verification (1999), 278–293.

[11] Chun, W., and P. D. Amer, *Improvements on UIO Sequence Generation and Partial UIO Sequences*, in: Int. Symp. Protocol Specification, Testing and Verification XII (1992), 245–260.

[12] Clarke, E. M., O. Grumberg, D. Peled, "Model checking," MIT Press, 1999.

[13] Clatin, M., Groz, R. M. Phalippou, R. Thummel, *Two Approaches Linking a Test Generation Tool with Verification Techniques*, in: 8th IWPTS (1995), 159–174.

[14] Devaraj, G., M. Heimdahl, D. Liang., *Coverage-Directed Test Generation with Model Checkers: Challenges and Opportunities*, in: 29th COMPSAC (2005), 455–462.

[15] Emerson, E., *Temporal and Modal Logic*, in: Handbook of Theoretical Computer Science. Elsevier (1990), 995–1072.

[16] Evtushenko, N., and A. Petrenko, *Synthesis of Test Experiments in Some Classes of Automata*, Automatic Control and Computer Sciences **24** (1990), 50–55.

[17] Fantechi A., Gnesi S., A. Maggiore, *Enhancing Test Coverage by Back-tracing Model checker Counterexamples*, Electronic Notes in Computer Sciences **116** (2005), 1999–2011.

[18] Forghani, B., B. Sarikaya, B. N. Res, O. Ottawa, *Semi-automatic Test Suite Generation from Estelle*, Software Engineering J. **7** (1992), 295–307.

[19] Fujiwara, S., and G. v. Bochmann, *Testing Non-Deterministic State Machines with Fault Coverage*, in: Int. Workshop on Protocol Test Systems IV (1991), 267–280.

[20] Gargantini, A., and C. L. Heitmeyer, *Using Model Checking to Generate Tests from Requirements Specifications*, in: ESEC / SIGSOFT FSE (1999), 146–162.

[21] Ghriga, M., and P. G. Frankl, *Adaptive Testing of Non-Deterministic Communication Protocols*, in: Int. Workshop on Protocol Test Systems VI. vol. C-19 (1993), 347–362.

[22] Gill, A., *State-identification Experiments in Finite Automata*, Information and Control **4** (1961), 132–154.

[23] Groz, R., *Unrestricted Verification of Protocol Properties on a Simulation Using an Observer Approach*, in: PSTV (1986), 255–266.

[24] Hong, H. S., S. D. Cha, I. Lee, O. Sokolsky, H. Ural, *Data Flow Testing as Model Checking*, in: ICSE (2003), 232–243.

[25] Huang, S., D. Lee, M. Staskauskas, *Validation-Based Test Sequence Generation for Networks of Extended Finite State Machines*, in: Forte IX (1996), 403–418.

[26] Kupferman, O., and M. Vardi, *An Automata-theoretic Approach to Modular Model Checking*, TOPLAS **22(1)** (2000), 87–128.

[27] Kupferman, O., and M. Vardi, *Module Checking Revisited*, in: 9th CAV (1997), 36–47.

[28] Lee, D.,, and M. Yannakakis, *Testing Finite-State Machines: State Identification and Verification*, IEEE Trans. Computers **43 (3)** (1994), 306–320.

[29] Luo, G., G. v. Bochmann, A. Petrenko, *Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method*, IEEE TSE **20** (1994), 149–162.

[30] Mealy, G. H., *A Method for Synthesizing Sequential Circuits*, Bell System Tech. J. **34** (1955), 1045–1079.

[31] Milner. R., "A Calculus of Communicating Systems," LNCS 92, 1980.

[32] Moore, E. F., *Gedanken-experiments on Sequential Machines*, in: Automata Studies, Princeton, N.J (1956), 129–153.

[33] ObjectGeode Simulator. Telelogic, 2005.

[34] Okun, V., P. E. Black, Y. Yesha, *Testing with Model Checker: Insuring Fault Visibility*, in: WSEAS Int. Conf. on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems (2002), 1351–1356.

[35] Petrenko, A., N. Yevtushenko, A. Levedev, A. Das, *Nondeterministic State Machines in Protocol Conformance Testing*, in: IWPTS VI. C-19 (1993), 363–378.

[36] Petrenko, A., S. Boroday, R. Groz, *Confirming Configurations in EFSM Testing*, IEEE TSE **30 (1)** (2004), 29–42.

[37] Petrenko, A., S. Boroday, R. Groz, *Confirming Configurations in EFSM*, in: FORTE XII and PSTV XIX (1999), 5–24.

[38] Petrenko. A.: *Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography*, Modeling and Verification of Parallel Processes (2000), 196–205.

[39] Rayadurgam, S., and M. Heimdahl, *Coverage Based Test-Case Generation Using Model Checkers*, in: Eighth IEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems (2001), 83–93.

[40] Robinson-Mallett, C., and P. Liggesmeyer, *State Identification and Verification using a Model Checker*, in: IASTED SE, LNI P-79 (2006), 131–142.

[41] Rusu, V., L. du Bousquet, Jron, T., *An Approach to Symbolic Test Generation*, in: 2nd Int. Conf. on Integrated Formal Methods (2000), 338–357.

[42] Safra, S., *Exponential Determinization for ω-Automata with Strong-Fairness Acceptance Condition (Extended Abstract)*, in: Stanford University Annual ACM Symp. on Theory of Computing (1992), 275–282.

[43] Starke, P., "Abstract Automata," North-Holland Pub. Co. 1972.

[44] Wang, C., L. Koh, M. Liu, *Protocol Validation Tools as Test Case Generators*, in: 7th IWPTS (1994), 155–170.