Connectivity Testing Through Model-Checking

Jens Chr. Godskesen[1,2], Brian Nielsen[1], and Arne Skou[1]

[1] Center of Embedded Software Systems, Aalborg University,
Fredrik Bajersvej 7B, DK-9220 Aalborg, Denmark
{jcg,bnielsen,ask}@cs.auc.dk
[2] IT-University of Copenhagen
Glentevej 67, DK-2400 Copenhagen NV., Denmark

Abstract. In this paper we show how to automatically generate test sequences that are aimed at testing the interconnections of embedded and communicating systems. Our proposal is based on the *connectivity fault model* proposed by [8], where faults may occur in the interface between the software and its environment rather than in the software implementation.
We show that the test generation task can be carried out by solving a reachability problem in a system consisting essentially of a specification of the communicating system and its fault model. Our technique can be applied using most off-the-shelf model-checking tools to synthesize minimal test sequences, and we demonstrate it using the UppAal real-time model-checker.
We present two algorithms for generating minimal tests: one for single faults and one for multiple faults. Moreover, we demonstrate how to exploit the unique time- and cost-planning- facilities of UppAal to derive cheapest possible test suites for restricted types of timed systems.

1 Introduction

Testing modern embedded and communicating systems is a very challenging and difficult task. In part, this is due to their complex communication patterns and by their reduced controllability and observability caused by the embedding and close integration with hardware. Although testing is the primary validation technique used by industry today, it remains quite ad hoc and error prone. Therefore there is a high demand for systematic and theoretically well founded techniques that work in practice and that can be supported by automated test tools.

A promising approach to improve the effectiveness of testing is to base test generation on an abstract formal model of the system under test (SUT) and use a test generation tool to (automatically or user guided) generate and execute test cases. A main problem is to automatically generate and *select* a reasonably small number of effective test cases that can be executed within the time allocated to testing.

This paper presents a technique for (formal) model-based (extended-finite state machines) black-box behavioral testing of embedded systems where a particular fault model, *connectivity faults*, is used to select test cases. Moreover, we

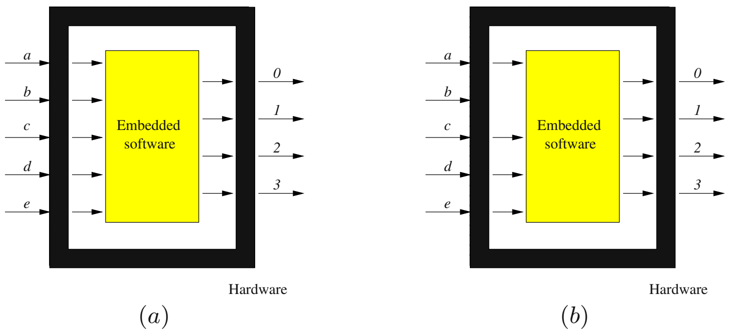

Fig. 1. An idealized view on embedded systems (a) and faulty embedded systems (b)

demonstrate how such test cases can be generated using the diagnostic trace facility of a standard, unmodified, model checking tool using standard reachability analysis.

1.1 Connectivity Testing

An embedded system may as presented in [8] idealistically be regarded as consisting of embedded software encapsulated by hardware, like depicted in Figure 1(a), where all communications to and from the software pass through the hardware. This is visualized by letting the inputs from the system environment to the software (a, b, c, d, e) pass through the hardware towards the software via *connections* (the unlabeled arrows). Likewise the outputs $(0, 1, 2, 3)$ generated by the software have to pass via connections through the hardware in order to emerge at the system environment.

A connection is by assumption related to exactly one input or output. This assumption implicitly implies that there is a one to one correspondence between external inputs to the system and the inputs to the embedded software, likewise there is a one to one correspondence between the outputs from the software and the outputs from the system.

Ideally it should be ascertained that the specification of the software component is correct. For instance, it may have been verified by some FSM verification technique. Then exploiting the ability to automatically generate executable code from specifications and assuming a careful construction of such compilers it would be reasonable to expect the generated code to be correct with respect to the specification, that is the two perform the same FSM behaviour.

In the composition of the two system components it then follows that the hardware (and probably drivers managing the interaction between the hardware and the software or malfunctioning sensors and actuators) may be the only error prone part. Therefore, in order to manage the multitude of potential errors we shall make an abstraction and regard the hardware (and the drivers, sensors, and actuators) as a black box interfacing the embedded software through the connections. As a consequence system errors may now only be referred to in terms of the connections.

In the system in Figure 1(a) a fault could for instance be that one of the input connections is missing as shown in Figure 1(b), where the b-input is *disconnected*. In the physical world, say for a mobile phone for instance, this may correspond to the situation where some button of the phone is not connected such that the software will never receive the input, and therefore the pressing of the button will cause no effect.

To ensure that the faults are testable they are assumed to be *permanent*. Testing in order to detect the kind of faults addressed in this paper is a matter of providing sequences of inputs that will reveal the missing connections. If say the b-input connection in Figure 1(a) is missing this may be revealed by an input sequence containing b and where eventually an expected output event is not produced or (in case the system is not input enabled) an expected input is not allowed.

We require that test generation is sound and complete, but from a practical perspective the generated suite should also be cost effective, e.g., in terms of test execution time. Thus, the suite should be minimized in the number of tests and length.

1.2 Contributions

We provide algorithms for generating tests for embedded systems with respect to fault models for input connectivity errors where for the system under test, it is assumed that the embedded software behaves as an FSM. By exploiting a real-time model-checker like UppAal [13] we are able to generate timed test sequences. However, to ease presentation we define our algorithms in the untimed setting of I/O deterministic EFSM (previous work [8] defined connectivity errors in term of Mealy machines). We prove that a *minimal* length sound and complete test (with respect to single connectivity faults) can be found via a reachability question of a composition of the system model, its fault model, and a simple environment model. We extend the basic algorithm to generate a *minimal* length test for *multiple* connectivity faults, and we prove its soundness and completeness.

Previous work [8] provided *dedicated*, *heuristic* polynomial time reduction algorithms; ours always produce the minimal (at the expense of increased complexity). Our algorithms can be implemented in most model-checking tools, but are additionally valid for a particular class of timed automatons using a real-time model-checker like UppAal. It symbolically solves clock constraints to perform reachability analysis on a network of timed automata, and produces a timed diagnostic trace (an alternating sequence of discrete transitions and time delays) to explain how the property is (is not) satisfied. We demonstrate the applicability of the algorithms on a medium size example (a cruise controller) – both in an untimed and a timed version, and indicate how the unique time- and cost-optimizing features of UppAal can be used to generate optimal tests.

The paper is organized as follows: Section 2 formally presents I/O EFSM's and tests. Section 3 presents the modelling of connectivity faults and illustrates how to test for such faults. Section 4 and Section 5 respectively present the algorithm for single and multiple faults. Section 6 presents the case study, and Section 7 elaborate on generation of time- and cost- optimal tests using UppAal's unique diagnostic trace features. Section 8 concludes and outlines future work.

1.3 Related Work

The use of diagnostic traces produced by model-checkers as test sequences has been proposed by many others [2, 3, 5, 6, 7, 10, 11, 15, 9]. A simple approach is based on manually stated test purposes, (i.e specific observation objectives to be made on the system under test) such as observing a given output, or bringing the SUT to a given state or mode, see e.g.,[6]. The test purpose is then formalized and translated to a logical (reachability) property to be analyzed by a model-checker. The resulting diagnostic trace is interpreted as a test case for that test purpose.

Another common approach is based on producing test suites that satisfy some coverage criteria of the specification, e.g. state- or transition- coverage, def-use pair coverage, MC/DC coverage etc. The simplest way of realizing e.g.,transition coverage is to formulate a property for each transition separately and use the model checker to produce a test case for each transition. More advanced techniques will naturally try to reduce the size of the test suite by removing redundant prefix-traces [15] or composing test cases by generating (minimal [9]) transition tours, [11, 9].

In [2] mutation testing is considered although in another setting than ours (they consider software testing). Mutations are used for generating tests to implementations of FSM's using model checking. Given is an FSM, M, and a constraining temporal logic formula, ϕ. A mutation may be either a change of a transition in M, or a change of ϕ. For each mutation a test is generated as a counter example as to why $M \not\models \phi$ (if $M \not\models \phi$). Duplicates and test being prefixes of other tests are removed, hence they do not as in our case generate a smallest possible test suite.

2 I/O EFSM

In this section we define input/output extended finite state automata (I/O EFSM) and their semantics.

Definition 1. *An* I/O EFSM *is a tuple*

$$M = (S, I, O, X, s_0, \longrightarrow)$$

where S is a finite set of states, $s_0 \in S$ is the initial state, I and O are finite disjoint sets of input and output labels respectively, X is a finite set of integer variables, and $\longrightarrow \subseteq S \times G_X \times (I \cup O) \times A_X \times S$ is a transition relation*, G_X is a set of* guards*, over the variables in X, and A_X is a set of finite sequences (possibly the empty sequence ϵ) of assignments to variables in X. Each guard is a boolean* *expression over integer constants and variables in X, and each assignment is on the form $x := e$ where $x \in X$ and $e \in E_X$ is a arithmetical expression over the variables in X and integer constants.*

Whenever $(s, g, \alpha, a, s') \in\longrightarrow$ we write $s \xrightarrow{g,\alpha,a} s'$. We write $s \xrightarrow{g,\alpha} s'$ ($s \xrightarrow{\alpha,a} s'$) instead of $s \xrightarrow{g,\alpha,\epsilon} s'$ ($s \xrightarrow{true,\alpha,a} s'$). We write $s \xrightarrow{\alpha} s'$ instead of $s \xrightarrow{\alpha,\epsilon} s'$. Often we write $\alpha!$ ($\alpha?$) whenever α is an output (input) symbol. Note that, for reasons of clarity and ease of presentation, we have omitted internal τ actions; our algorithms can easily be adapted to handle these as well.

2.1 Semantics

The semantics of an I/O EFSM M is a labelled transition system defined wrt. a *valuation function* assigning values to the variables of M and used to evaluate the guards on transitions.

Definition 2. *A valuation v for a set of integer variables X is a function $v : X \to \boldsymbol{N}$. We let V_X denote the set of all valuations for X. $\overline{0}_X \in V_X$ is the valuation where $\overline{0}_X(x) = 0$ for all $x \in X$. For $n \in \boldsymbol{N}$, $v[x \mapsto n]$ evaluates all variables y to $v(y)$ except x that is evaluated to n.*

Given a valuation $v \in V_X$, the value of a guard $g \in G_X$ with respect to v, denoted by $v(g)$, is the obvious evaluation of the boolean expression g relative to v. Moreover, for a sequence of assignments $a \in A_X$, $v(a) \in V_X$ is defined inductively by $v(\epsilon) = v$ and $v(x := e, a) = v[x \mapsto n](a)$ where n is the value obtained by evaluating expression e using the valuation v.

The semantics of an I/O EFSM is defined as a labelled transition system.

Definition 3. *Let $M = (S, I, O, X, s_0, \longrightarrow_M)$. The labelled transition system induced by M is*

$$T_M = (S \times V_X, I \cup O, (s_0, \overline{0}_X), \longrightarrow)$$

where $(s_0, \overline{0}_X) \in S \times V_X$ is the initial state. The labelled transition relation $\longrightarrow \subseteq (S \times V_X) \times (I \cup O) \times (S \times V_X)$ is the least relation satisfying:

$$\frac{s \xrightarrow{g,\alpha,a}_M s'}{(s, v) \xrightarrow{\alpha} (s', v(a))}\ v(g) \textit{ is true}$$

Whenever $(s, v) \xrightarrow{\alpha} (s', v')$ we write $(s, v) \xrightarrow{\alpha?} (s', v')$ if $\alpha \in I$, otherwise if $\alpha \in O$ we write $(s, v) \xrightarrow{\alpha!} (s', v')$.

We say that a transition system is *I/O deterministic* if for any state there are at most one output transition and at most one input transition for any input. M is I/O deterministic if its induced transition system T_M is I/O deterministic.

Two automatons M and M' are equivalent, $M \sim M'$, if the initial states in T_M and $T_{M'}$ are trace equivalent.

We only consider the parallel composition of I/O EFSM's at the semantic level. By convenience, and without loss of generality, we assume all machines have the same variables. It follows from the definition that output actions are broadcasted.

Definition 4. *Let $T_i = (S_i \times V_X, L_i, (s_0^i, \overline{0}_X), \longrightarrow_i)$, $i = 1, \ldots, k$ be I/O EFSM induced labelled transition systems. The parallel composition $\Pi_{i=1}^k T_i$ is defined by*

$$\Pi_{i=1}^k T_i = ((S_1 \times \ldots \times S_k) \times V_X, L, ((s_0^1, \ldots, s_0^k), \overline{0}_X), \longrightarrow)$$

where $L = \cup_{i=1}^k L_i$, and $\longrightarrow$ is the least relation satisfying

$$\frac{(s_i, v) \xrightarrow{\alpha!}_i (s_i', v_i') \quad \forall j \neq i.\ (s_j, v_j) \overset{\alpha?}{\longmapsto}_j (s_j', v_j')}{((s_1, \ldots, s_k), v) \xrightarrow{\alpha} ((s_1', \ldots, s_k'), v')}$$

where $(s, v) \overset{\alpha?}{\longmapsto}_n (s', v')$ if $(s, v) \xrightarrow{\alpha?}_n (s', v')$ and $(s, v) \overset{\alpha?}{\longmapsto}_n (s, v)$ if $(s, v) \not\xrightarrow{\alpha?}_n$ and v' is a valuation accumulating all the updates $v_1', \ldots, v_k'$.[1]

2.2 Tests

In our setting a *test* is an I/O EFSM except that each state is annotated by either the verdict *pass* or *fail*.

Definition 5. *Let I and O be finite disjoint sets of input and output symbols respectively. Let $\alpha_1 \ldots \alpha_n \in (I \cup O)^+$. Define the* test

$$M_{\alpha_1 \ldots \alpha_n}^{pass}(I, O) = (\{s_0, \ldots, s_n\}, I, O, s_0, \longrightarrow)$$

such that $\longrightarrow$ is the least relation where

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$$

and where s_n is annotated by pass and all $s_0, \ldots, s_{n-1}$ are annotated by fail. Define $M_{\alpha_1 \ldots \alpha_n}^{fail}(I, O)$ as $M_{\alpha_1 \ldots \alpha_n}^{pass}(I, O)$ except that the state s_{n-1} is annotated by pass and the remaining states by fail.

If in every complete run of $T_M \parallel T_{M_t^v(I,O)}$ the component $T_{M_t^v(I,O)}$ terminates in the state *pass* (*fail*) we say that M passes (fails) the test $M_t^v(I, O)$; otherwise I/O deterministic $T_M \parallel T_{M_t^v(I,O)}$ has precisely one complete run.

[1] We leave out the formal definition of v'. Our algorithms make use of shared variables between automatons but carefully ensure that simultaneous updates cause no problems.

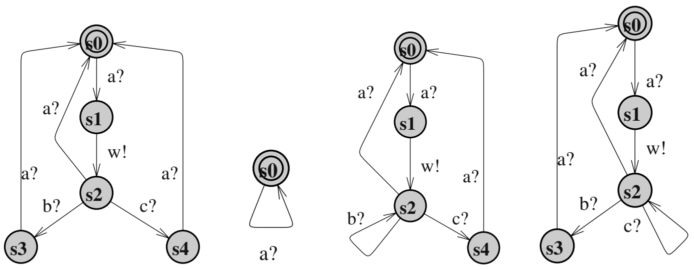

Fig. 2. A simple model M and its mutants $M[a]$, $M[b]$, $M[c]$. The initial location is doubly encircled

3 Modelling and Testing Connectivity Faults

As mentioned previously, a connection is assumed to be related to exactly one input[2]. That is, when the connection related to a given input (say α) is faulty, the software will not receive any α-input, i.e. the state of the software will remain unchanged, whenever the environment makes an input to the system via α. We can therefore model a connectivity fault as a so-called *mutation* $M[\alpha]$ of a correct model M by changing all α-transitions so that the state is not changed. This is made precise in the following definition:

Definition 6. *Let* $M = (S, I, O, X, s_0, \longrightarrow)$ *and* $\alpha \in I$. *Define*

$$M[\alpha] = (S, I, O, X, s_0, \longrightarrow_1)$$

where $\longrightarrow_1$ *is* $\longrightarrow$ *except that all transitions* $s \xrightarrow{g,\alpha,a} s'$ *are replaced by* $s \xrightarrow{g,\alpha}_1 s$.

Figure 2 shows a simple model with 3 inputs (a,b,c) and the mutants $M[a]$,$M[b]$,$M[c]$. An α-connectivity fault may be found by applying a test that distinguish M from $M[\alpha]$. For the mutants $M[a]$, $M[b]$, and $M[c]$ in Figure 2, we may construct the tests $M^{pass}_{aw}(I, O)$, $M^{fail}_{awbc}(I, O)$, and $M^{fail}_{awcb}(I, O)$ respectively where $I = \{w\}$ and $O = \{a, b, c\}$. Clearly, the tests are minimal (in terms of the number of synchronizations between the tester and the system) and $M^{fail}_{awbc}(I, O)$ and $M^{fail}_{awcb}(I, O)$ are sufficient to distinguish M from all three mutations. There exists no single test distinguishing M from all the mutations.

4 The Test Generation Algorithm

In this section we present an algorithm for generating a test that distinguish an I/O EFSM from a single mutation (if they are distinguishable). In the algorithm

[2] We restrict ourselves to input faults. However, the extension to output faults is straightforward.

we use the following two operators: $M?$ is M where outputs become inputs and $M(x := e)$ is M where on any transition x is updated by e. Formally we have

Definition 7. *Let* $M = (S, I, O, X, s_0, \longrightarrow)$ *then* $M? = (S, I \cup O, \emptyset, X, s_0, \longrightarrow)$.

Definition 8. *Let* $M = (S, I, O, X, s_0, \longrightarrow)$. *Then for any* x *and* $e \in E_{X \cup \{x\}}$

$$M(x := e) = (S, I, O, X \cup \{x\}, s_0, \longrightarrow_1)$$

where $\longrightarrow_1$ *is* $\longrightarrow$ *except that for every* $\alpha \in I \cup O$, *any* $s \xrightarrow{g,\alpha,a} s'$ *is replaced by* $s \xrightarrow{g,\alpha,a'}_1 s'$ *where* a' *is* $a, x := e$.

We let $M(x_1, \ldots, x_k := e_1, \ldots, e_k)$ be $M(x_1 := e_1) \ldots (x_k := e_k)$ whenever $e_i \in E_{X \cup \{x_i\}}$, $i = 1, \ldots, n$.

4.1 The Algorithm

The problem we want the algorithm to solve is the following:

> Given an I/O deterministic EFSM M and one of its input symbols α, if $M \not\sim M[\alpha]$ then find a test M' with fewest possible states such that M and $M[\alpha]$ respectively passes and fails M'.

Intuitively, the idea behind the algorithm is to put M and its mutation together in parallel with a third machine, the environment E. Only E is allowed to submit actions, the other machines are modified to contain solely input actions. The role of E is to broadcast actions such that whenever the two other machines do not agree on receiving an action (recall they are I/O deterministic) a fault has been detected. The algorithm searches for a shortest possible trace of actions broadcast by E leading to a fault.

Pseudo Code

```
1. Let $x$, $y$ and $z$ be disjoint variables none of which belong to $X$.
2. Let $E = (\{s\}, \emptyset, I \cup O, X \cup \{x, y, z\}, s, \{(s, \beta, s) \,|\, \beta \in I \cup O\})$.
3. Let $c_1$ and $c_2$ be distinct constants and let $M_1 = M?(x, y, z := (x + 1)\%2, y, c_1)$ and $M_2 = M[\alpha]?(x, y, z := x, (y + 1)\%2, c_2)$.
4. Construct $T_E \parallel T_{M_1} \parallel T_{M_2}$ with initial state $s_T$.
5. Let $t$, if it exists, be a minimal trace such that $s_T \xrightarrow{t} (s_t, v_t)$ with $v_t$ satisfying $x \neq y$. If $t$ doesn't exists return false.
6. If $v_t(z) = c_1$ return $M_t^{pass}(O, I)$, otherwise return $M_t^{fail}(O, I)$.
```

The only technicality of the algorithm is the use of the variables x, y, and z. The role of x and y is to count (modulo 2) whenever $M?$ and $M[\alpha]?$ respectively have synchronized with E. Hence, whenever $x \neq y$ a fault have been detected. The role of z is to register which of $M?$ and $M[\alpha]?$ engaged in a synchronization with E, this is important as to wheter the returned test should be a test with verdict *pass* or *fail*.

The correctness of the construction of the test automaton follows from the theorem below.

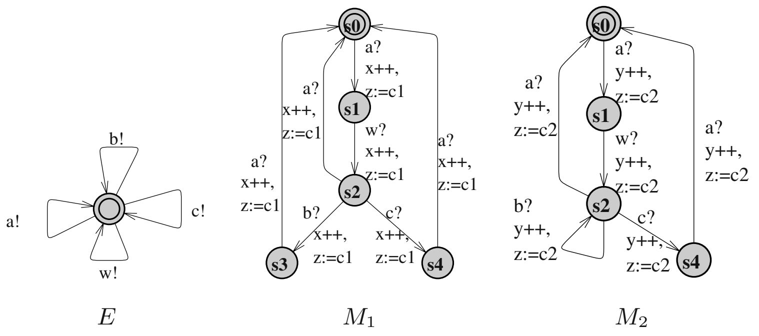

Fig. 3. Annotated automata

Theorem 1. *If the algorithm on input M and α returns false then $M \sim M[\alpha]$, otherwise it returns a test M' with fewest possible states such that M passes M' and $M[\alpha]$ fails M'.*

4.2 Example

If we apply the algorithm on input M (Figure 3) and action b then the three machines put in parallel, M_1, M_2, and E, are as devised in Figure 3. For illustrative clarity $x{+}{+}$ is taken to mean x incremented modolus 2.

The test $M^{fail}_{awbc}(\{w\}, \{a, b, c\})$ is a minimal length test that may be constructed by the algorithm. Clearly, $awbc$ is a shortest possible sequence leading to a state in $T_E \parallel T_{M_1} \parallel T_{M_2}$ where $x \neq y$, and since only M_2 can engage in the last event c the value of z is c_2.

5 The Generalized Algorithm

Next, we generalize the algorithm above such that a whole suite of test automatons are generated for a set of mutations (if all mutations are distinguishable from M).

5.1 The Algorithm

The problem the algorithm solves is

> Given $M = (S, I, O, X, s_0, \rightarrow)$, an I/O deterministic automaton, and a set of input symbols $\mathcal{A} \subseteq I$, if $M \not\sim M[\alpha]$ for all $\alpha \in \mathcal{A}$, find a minimal test suite $\mathcal{M}$ such that 1) M passes all automatons in $\mathcal{M}$, and 2) for all $\alpha \in \mathcal{A}$, $M[\alpha]$ fails M', for some $M' \in \mathcal{M}$.

Notice, that a minimal test suite $\mathcal{M}$ satisfies that for all $M' \in \mathcal{M}$ there exists $\alpha \in \mathcal{A}$ such that $M[\alpha]$ fails M' and $M[\alpha]$ passes all other test automatons in $\mathcal{M}$, i.e. all tests returned by the algorithm are indeed needed and cannot be removed from the suite if all connectivity faults are to be revealed.

The main idea is to extend the previous algorithm by running *all* mutants concurrently, but tightly synchronized, with the unmutated automaton M. Whenever the unmutated machine M cannot match a transition by one of its mutations a connectivity error has been detected, and M needs to be reset (and only then) to extend the sequence to kill more mutants.

Pseudo Code

```
1. Let $\{x, x_\alpha, y_\alpha, z \mid \alpha \in A\}$ be fresh variables disjoint from $X$. Extend $M$ to contain the variables $Y = X \cup \{x, x_\alpha, y_\alpha, z \mid \alpha \in \mathcal{A}\}$.
2. Let $M' = M?(x := (x+1)\%2)$ and let for all $\alpha \in \mathcal{A}$, $M_\alpha = M[\alpha]?(x_\alpha := (x_\alpha + 1)\%2)$
3. Let $go$ and $reset$ be two new fresh actions not in $I \cup O$. Add $\{go, reset\}$ to the set of output labels for $M'$. Let $M''$ be $M'$ with any transition $s \xrightarrow{g,\alpha,a} s'$ where $\alpha \in I \cup O$, replaced by $s \xrightarrow{g,\alpha,a} s'' \xrightarrow{go} s' \xrightarrow{reset} s_0$ where for each replacement $s''$ is a new fresh state.
4. For each $\alpha \in \mathcal{A}$, add $\{reset\}$ to the set of input labels for $M_\alpha$ and add a reset transition, $s \xrightarrow{reset} s_0$, for any state $s$ in $M_\alpha$ to its initial state $s_0$.
5. Let $E = (\{s_0, s_1\}, \{go, reset\}, I \cup O, Y, s_0, \longrightarrow)$ where
       $\longrightarrow = \{(s_0, (\alpha, z := 0), s_1) \mid \alpha \in I \cup O\} \cup \{s_1 \xrightarrow{go, z:=1} s_0, s_1 \xrightarrow{reset, z:=1} s_0\}$
6. Construct $M'_\alpha = (\{s_0, s_1\}, \emptyset, \emptyset, Y, s_0, \longrightarrow)$ for all $\alpha \in \mathcal{A}$ where $s_0 \xrightarrow{g,\tau,a} s_1$ with $a = (y_\alpha := 1)$ and $g = (x \neq x_\alpha)$.
7. Construct $T_E \parallel T_{M''} \parallel \Pi_{\alpha \in \mathcal{A}} T_{M_\alpha} \parallel \Pi_{\alpha \in \mathcal{A}} T_{M'_\alpha}$ with initial state $s_T$.
8. Let $P$ be $z \neq 0 \wedge \forall \alpha \in \mathcal{A}.\ y_\alpha \neq 0$.
9. Let $t = t'\beta$, if it exists, be a minimal trace such that $s_T \xrightarrow{t} (s_t, v_t)$ with $v_t$ satisfying $P$. If $t$ doesn't exists return false.
10. Let $t_1, \ldots, t_k$ be such that $u = t_1 reset\ t_2 \ldots reset\ t_k$ where $u$ is $t'$ with all $go$ and $\tau$'s removed.
11. Return $\mathcal{M} = \{M^{fail}_{t_1}(O,I), \ldots, M^{fail}_{t_{k-1}}(O,I), M^{v}_{t_k}(O,I)\}$ where $v$ is $fail$ if $\beta = reset$, otherwise if $\beta = go$ then $v$ is $pass$.
```

To control when M has to be reset every α transition by M is now followed by a new output action called *go* which intuitively acknowledge to the environment E that M could match α. After having ouput an action, E waits for this acknowledgement before it sends a new action. If the acknowledge does not arrive E knows that M could not perform the action, implying that a test has been found for at least one of the mutations. In that case the only possible synchronization is a *reset* between M and the environment automaton.

In order to detect when a connectivity error has been identified we introduce an observation automaton M'_α for each mutation α. It consists of two states and

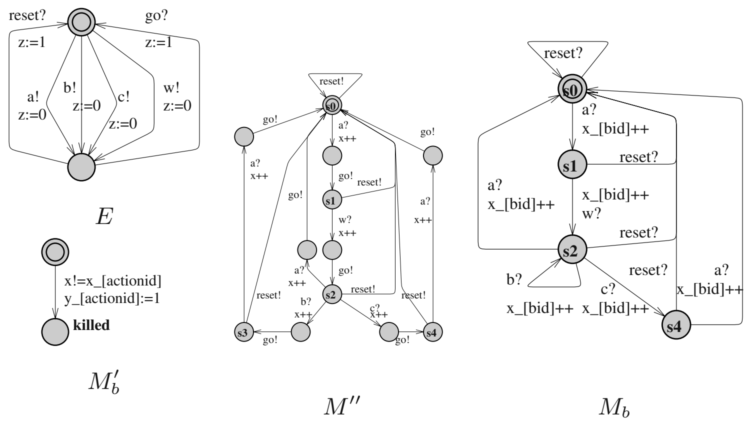

Fig. 4. The annotated automata (M_a, M_c, M'_a, M'_c, omitted). The notation $v[i]$ is UppAal notation for indexing array v at position i. *bid* is the position for action b

one transition that fires when M and $M[\alpha]$ does not agree on some input or output transition, i.e. when $x \neq x_\alpha$. All mutations have been revealed when all observation automata has fired, i.e, when all $y_\alpha = 1$.

Based upon the trace $t = t'\beta$ found (if a trace is found) a set of test automatons are constructed. First all *go*'s and τ's are removed from t'. Then t' is split in the parts $t_1, \ldots, t_k$ separated by *reset* labels. For all t_i, but t_k, *fail* test automations, $M^{fail}_{t_i}(O, I)$ are created, since M clearly cannot perform those traces—that was the sole reason why M was reset. To be able to tell whether the final part, t_k, should give rise to a fail or pass test automaton we force the trace to always end with either *reset* or *go*. This is done by introducing a variable z in the environment automaton that is set to 0 on transitions with labels in $I \cup O$ and to 1 when a *go* or *reset* is performed. Then searching for t we require z is not zero. Clearly, if the last event in t, i.e. β, is a *go* then $M^{pass}_{t_k}(O, I)$ is created, otherwise if it is *reset* then $M^{fail}_{t_k}(O, I)$ is created.

Notice, that a test automaton $M^{v}_{t_i}(O, I)$ may detect several mutations.

Theorem 2. *If the algorithm on input M and $\mathcal{A}$ returns false then $M \sim M[\alpha]$ for some $\alpha \in \mathcal{A}$, otherwise it returns some $\mathcal{M}$ satisfying the properties in Section 5.1.*

5.2 Example

Given the I/O EFSM M in Figure 2 the algorithm produces the sequence *awbb.reset.awcb.reset*, resulting in the two tests $M^{fail}_{awbb}(\{w\}, \{a, b, c\})$ and $M^{fail}_{awcb}(\{w\}, \{a, b, c\})$. Both tests for connectivity of a. The used annotated models are depicted in Figure 4.

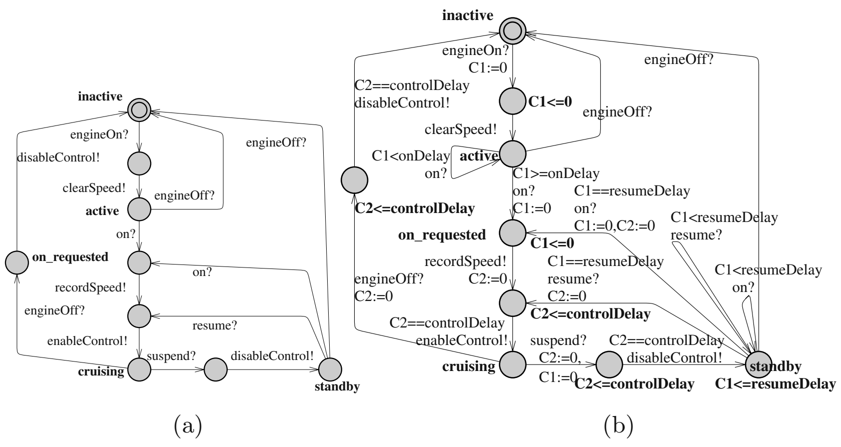

Fig. 5. User Interface Automaton (a) Timed user interface (b)

6 Cruise Controller Example

In this section we exemplify and benchmark our technique on a medium sized cruise controller example. The cruise controller is commonly studied and found in many variations in the literature, and thus serves as an illustrative example, see e.g., [14, 2].

6.1 The Cruise Controller

The model consists of two automatons. The *user interface* controls the different modes of operation according to the various user inputs, whereas the *speed control* keeps the actual speed close to a given desired speed by affecting the throttle of the engine.

The *user interface* (Figure 5(a)) basically has four modes, i.e. *inactive* when the engine is turned off, *active* when the engine is turned on, *cruising* when the speed control is enabled, and *standby* when the speed control is temporarily suspended. When the engine is turned on, the desired speed is cleared, and when cruise mode is entered, the actual speed is recorded and set as the desired speed. The cruise mode may be reentered from standby mode.

The *speed control* (Figure 6) switches between its two operational modes *disabled* and *enabled* according to enable and disable control signals from the user interface. In disabled mode, it sets the desired speed to zero or to the sampled actual speed when commanded by the user interface. In enabled mode, it samples the actual speed and based on the difference between actual and desired speed (represented by variables cSpeed, dSpeed), it stops acceleration of the engine (output *inc0*), or commands the engine to do medium (output *inc1*) or high (output *inc2*) acceleration. Further, in enabled mode, the user can manually increase or decrease the desired speed.

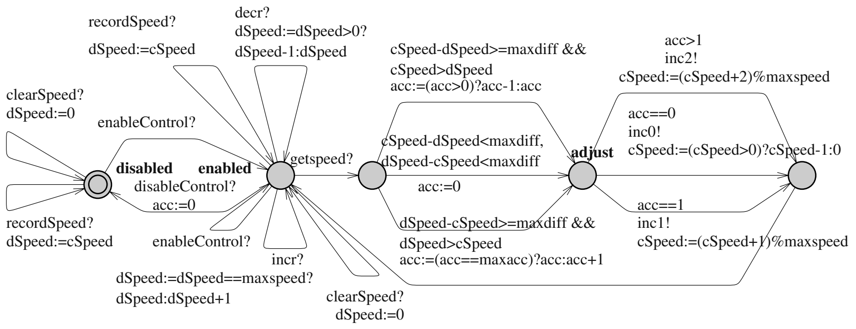

Fig. 6. Speed Control Automaton

The actions of the user interface are $I_u = \{\mathit{engineOn}, \mathit{engineOff}, \mathit{on}, \mathit{suspend}\}$, $O_u = \{\mathit{clearSpeed}, \mathit{recordSpeed}, \mathit{enableControl}, \mathit{disableControl}\}$. The actions of the speed ontroller are $I_s = O_u \cup \{\mathit{incr}, \mathit{decr}, \mathit{getspeed}, \}$, $O_s = \{\mathit{inc0}, \mathit{inc1}, \mathit{inc2}\}$. For the system composed of the user interface and speed controller synchronizing internally[3] on actions $O_u \cap I_s$, the actions are $I_c = I_u \cup I_s \setminus O_u$, $O_c = O_s$.

6.2 Generated Test Sequences

Unless the length of the test suite is important, the normal and computationally most efficient method is to generate a separate test sequence for each mutant. Our experimental results show that a sequence could be successfully generated for each mutant; also the sequences are quite short. The test suites generated for the cruise interface, the speed controller and the composed system contain respectively 5 (16), 7 (31), 8 (34) test cases (total steps). All were generated on a standard PC in less than one second. Table 1 lists some examples.

These results indicate that our technique may be feasible for much larger systems, both in terms of test suite size and model size (number of inputs and state space). Since the algorithm generates the minimal sequences, some of them are quite surprising and would not likely be found by hand, e.g., the test for *engineOn*. Observe especially that it is not obvious how the desired and current speed should be set to distinguish the mutants of the speed controller. For instance, it would be incorrect to use the intuitive test $M^{pass}_{enableControl.incr.getspeed.inc0}(O_s, I_s)$ to check for connectivity of *incr* because *inc0* would also be output if *incr* was disconnected (given that $\mathit{maxDiff} = 2$, *dSpeed* and *cSpeed* initially equals 0, *acc* becomes 0 in both cases). Hence at least two increments are needed.

Also note that—because our algorithms does not require the specification or implementation to be input enabled—not all sequences end with an output,

[3] Recall that our technique can be adapted to handle these. The semantics of the input fault mutations in a composed system is as if they were made to their (synchronous) product I/O EFSM, hiding internal communication channels.

Table 1. Selection of Generated Tests (if v =P then $M_t^{pass}(O,I)$; if v =F then $M_t^{fail}(O,I)$)

Mutant	v	Generated Event Sequence t
		User Interface
engineOff	P	engineOn.clearSpeed.engineOff.engineOn
suspend	P	engineOn.clearSpeed.on.recordSpeed.enableControl.suspend.disableControl
resume	F	engineOn.clearSpeed.on.recordSpeed.enableControl.suspend.-disableControl.resume.engineOff
		Speed Controller
incr	P	enableControl.incr.incr.getspeed.inc1
decr	P	enableControl.incr.incr.decr.getspeed.inc0
clearSpeed	P	enableControl.incr.incr.clearSpeed.getspeed.inc0
		Cruise Controller System
engineOn	F	engineOn.engineOn
resume	F	engineOn.on.suspend.resume.resume
incr	P	engineOn.on.incr.incr.getSpeed.inc1

meaning that if the last input can be performed by the tester, the test will pass (or fail, depending on the verdict). If this is felt to be unnatural for some applications, it is very easy to force our algorithms to produce tests that ends with an output. The generated test for *engineOff* is then

$M^{pass}_{engineOn.clearSpeed.engineOff.engineOn.clearspeed}(O_u, I_u)$.

6.3 Multi-fault Test Sequences

In some cases it is important to produce a smallest test suite with as few and short tests as possible. A simple reduction technique like prefix elimination does not work well for connectivity testing (see sequences presented in Section 6.2). Our generalized algorithm from Section 5 is therefore more involved and guarantees that the minimal length test suite is computed, although at the expense of computational complexity (the problem is NP-hard [8]). It involves analyzing a system consisting of all mutants running concurrently in a synchronized step-lock fashion. Thus, state space explosion theoretically limits how many mutants can be composed, and it should be examined where this limit occur in practice. The following experiments are run on a 8x900 MHZ Sun Sparc Fire v880R workstation with 32 GB memory running Sun Solaris 9 (SunOS 5.9). However, UppAal only exploits one CPU and addresses at most 4 GB of memory. The results are tabulated in Table 2.

For the user interface, it turns out that it is possible to compute (using only a few seconds and megabytes of memory) a single test of 11 steps that detects all input faults $M_t^{pass}(O_u, O_u)$ where $t =$ *engineOn.clearSpeed.on.recordSpeed.enableControl.suspend.disableControl.-resume.enableControl.engineOff.disableControl*, giving a reduction of 31% (11 versus 16 steps).

The speed controller and the composed system have much larger state spaces and are more challenging. Still, all mutants but two could be composed in both cases. The multiple-fault test suite (up to and including the *incr* mu-

Table 2. Performance of multi-fault algorithm

Speed Control		
Mutant(s)	CPU-time(s)	Memory (KB)
enableControl	0.21	3704
+*disableControl*	0.37	5672
+*clearSpeed*	10.98	29008
+*recordSpeed*	152.77	281072
+*incr*	1917.02	2128824
+*decr*	-	-
+*getspeed*	-	-

Cruise System		
Mutant(s)	CPU-time(s)	Memory (KB)
engineOn	0.16	5232
+*engineOff*	0.29	5584
+*resume*	27.51	97608
+*on*	39.01	100208
+*suspend*	50.60	131192
+*incr*	874.00	1516800
+*decr*	-	-
+*getspeed*	-	-

tant) for the speed control consists of two tests: $M_{t_1}^{fail}(O_s, I_s)$ and $M_{t_2}^{fail}(O_s, I_s)$ where t_1= *enableControl.incr.incr.getspeed.inc1.recordSpeed.incr.getspeed.inc0.-clearspeed.getspeed.inc1*, t_2= *enableControl.disableControl.incr*, giving a reduction of 25% compared to detecting the same faults using seperate sequences. In addition many system resets are avoided. The cruise system (up to *incr*) requires only one test: $M_t^{fail}(O_c, I_c)$ where t= *engineOn.engineOff.engineOn.-on.suspend.resume.incr.incr.getspeed.inc0*, giving a reduction of 40%. The order of addition of mutants was arbitrary. Even if the test suite is generated by more rounds composing only some of the mutants each time, the reduction in test suite size is significant.

7 Timed Test Generation

We next demonstrate how connectivity tests for a class of timed systems can be generated. The tester now needs to be time aware to reveal them. This result requires no change to the basic algorithm if a real-time model-checker like UppAal is used.

Informally, a timed automaton [1] is an I/O EFSM equipped with a set of non-negative real-valued variables called *clocks* that may be used in guards, and may be set to zero on transition assignments. In addition, *location invariants* forces the automaton to take a transition before it becomes false. The semantics of a timed automaton is defined in terms of an infinite timed transition system consisting of both discrete transitions and time delay transitions. To ensure testability we impose similar semantic restrictions as in [16]: Our model, called DOUTA, are deterministic, output urgent (an output or τ occurs as soon as it is enbled) timed automata. DOUTA is formally defined in [9].

Consider the following real-time requirements for the user-interface automaton in Figure 5(a). 1) For safety reasons, the engine must be on for at least *onDelay* before cruise control may be switched on. Earlier requests must be ignored. 2) When cruise mode is suspended, at least *resumeDelay* must elapse before reengagement to avoid too rapid enabling and disabling of the speed controller. 3) It takes *controlDelay* to enable or disable the speed controller (involves external communication), whereas the speed can be set or cleared with a zero delay (assumed internal communication). These requirements are satisfied by the

DOUTA in Figure 5(*b*). Boldfaced clock constraints below locations are location invariants.

Given this specification (*onDelay*=5000, *controlDelay*=3000, *controlDelay*=200) UppAal produces the following timed test $M_t^{fail}(O_u, I_u)$ to reveal disconnection of the *resume* action, where *t*= *0.engineOn.0.clearSpeed.5000.-on.0.recordSpeed.200.enableControl.0.suspend.200.disableControl.2800.resume.-0.engineOff* . Note that the delays are not a trivial insertion of the delay constants occurring in the model (e.g the 2800 ms between *disableControl* and *resume*). It is usually infeasible to compute these by hand because it involves solving a large set of inequations on clock variables. The zero delays in the above sequence can be avoided by replacing the universal environment E by a more accurate (and slower) environment model timed automaton E' which restricts the choices of the tester.

UppAal also has efficient facilities for generation of time- and cost- optimal diagnostic traces [4, 12]. In fact, the above test is not only of minimal length, but also the fastest (minimal accumulated time delay). To avoid expensive operations, e.g., resets, UppAal can be used to generate suites with the fewest such operations. As a simple example, the generated multi-fault test presented in Section 6.3 for the speed controller required two tests, and thus one reset. Searching for a test with fewer resets UppAal found one (only two communication events longer that the minimal length test suite): $M_t^{pass}(O_u, I_u)$ where *t*= *enableControl.incr.clearSpeed.incr.getspeed.inc0.incr.getspeed.inc1.-disableControl.enableControl.clearSpeed.recordSpeed.incr.incr.getspeed.inc1.* It is also possible to take the actual time/cost for a reset into account.

8 Conclusions and Future Work

This paper describes two sound and complete algorithms that generate minimal test cases and test suites respectively for input connectivity faults. The algorithms are based on reachability analysis and may thus be implemented in most model-checkers. Based on experiments with a concrete model-checker, UppAal, and a medium sized example, we conclude that our techniques are feasible, and for the simple algorithm appear to scale to larger systems. For the generalized algorithm the number of simultaneous mutants that can be handled is limited due to state space explosion (recall that the problem is NP hard). Finally, we show how timed connectivity and examples of cost optimized test suites can be generated by the same algorithms.

We only looked at input connectivity faults, however it is trivial to generate test sequences for output connectivity faults, since this amounts to finding a sequence that visits a transition where the output is produced, hence making it observable.

As future work we plan to examine other more involved fault models, e.g. models where connections may be whole protocols. Since our algorithms are based on finding a trace that can be performed by the original automaton and not its mutant, or vice versa, our algorithms appear to be so general that many other fault models can be supported. In particular we plan to investigate how to test wrongly interconnected communicating (distributed) components that have been tested or verified in isolation. Also, we plan to investigate a timed connectivity fault model where disconnects are not permanent and we intend to do practical application and further experiments with time-and cost-optimal test suite generation.

References

[1] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, April 1994. 181

[2] P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *ICFEM*, page 46, 1998. 170, 178

[3] P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. 170

[4] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. Efficient Guiding Towards Cost-Optimality in Uppaal. In T. Margaria and W. Yi, editors, *TACAS 2001*, number 2031 in LNCS, pages 174–188. Springer–Verlag, 2001. 182

[5] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using modelchecking. In *1996 SPIN Workshop*, August 1996. Also WVU Report NASA-IVV-96-022. 170

[6] A. Engels, L. Feijs, and S. Mauw:. Test generation for intelligent networks using model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems. TACAS'97*, number 1217 in LNCS, 1997. 170

[7] A. Gargantini and C. L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC / SIGSOFT FSE*, pages 146–162, 1999. 170

[8] Jens Chr. Godskesen. Complexity issues in connectivity testing. In Ed Brinksma and Jan Tretmans, editors, *Proceedings of the Workshop on Formal Approaches to Testing of Software, FATES '01, (Aalborg, Denmark, August 25, 2001)*, 2001. 167, 168, 169, 180

[9] A. Hessel, K. G. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-Optimal Test Cases for Real-Time Systems. In *3rd Intl. Workshop on Formal Approaches to Testing of Software (FATES 2003)*, Montréal, Québec, Canada, October 2003. 170, 181

[10] H. Hong, I. Lee, O. Sokolsky, and S. Cha. Automatic test generation from statecharts using model checking. In Ed Brinksma and Jan Tretmans, editors, *Workshop on Formal Approaches to Testing of Software, FATES '01, (Aalborg, Denmark, August 25, 2001)*, 2001. 170

[11] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In J.-P. Katoen and P. Stevens, editors, *TACAS 2002*, pages 327–341. Kluwer Academic Publishers, April 2002. 170

[12] K. G. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automat. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of CAV 2001*, number 2102 in LNSC, pages 493–505. Springer–Verlag, 2001. 182

[13] K. G. Larsen, P. Pettersson, and W. Yi. UppAal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997. 169

[14] Magee and Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2002. 178

[15] M. P. E. Heimdahl and S. Rayadurgam and W. Visser and G. Devaraj and J. Gao. Auto-generating Test Sequences Using Model Checkers: A Case Study. In A. Petrenko and A. Ulrich, editors, *3rd Intl. Workshop on Formal Approaches to Testing of Software, FATES 2003*, volume 2931 of *LNCS*, pages 42–59, Montréal, Québec, CA, 2004. 170

[16] J. G. Springintveld, F. W. Vaandrager, and P. R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2), March 2001. 181