Specifying and Generating Test Cases Using Observer Automata

Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson

Department of Information Technology, Uppsala University,
P.O. Box 337, SE-751 05 Uppsala, Sweden
{johan, hessel, bengt, paupet}@it.uu.se

Abstract. We present a technique for specifying coverage criteria and a method for generating test suites for systems whose behaviours can be described as extended finite state machines (EFSM). To specify coverage criteria we use observer automata with *parameters*, which monitor and accept traces that cover a given test criterion of an EFSM. The flexibility of the technique is demonstrated by specifying a number of well-known coverage criteria based on control- and data-flow information using observer automata with parameters. We also develop a method for generating test cases from coverage criteria specified as observers. It is based on transforming a given observer automata into a bitvector analysis problem that can be efficiently implemented as an extension to an existing state-space exploration such as, e.g. SPIN or UPPAAL.

1   Introduction

Model based test case generation has in recent years been developed as a prominent technique in testing of reactive software systems. A model serves both the purpose of specifying how the system should respond to inputs from its environment, and of guiding the selection of test cases, e.g., using suitable coverage criteria. Typical notations for such models are state machines in some form, often extended with data variables. Test cases can be selected as individual "executions" of the model, checking that the outputs from the system under test (SUT) conform to those specified by the model.

There is a large literature and several tools (e.g., [4, 17, 24, 18, 3]) for generation of test cases from extended state machine models (EFSMs). In typical approaches, the selection of test cases follows some particular coverage criterion, such as coverage of control states, edges, etc., or using an explicitly given set of test purposes [5, 23]. When the model contains data variables, constraint solving techniques can be used to find actual values of input parameters that drive the execution in a desired direction [17, 21, 19].

Since different coverage criteria are suitable in different situations, and satisfy different constraints on fault detection capability, cost, information about where potential faults may be located, etc., it is highly desirable that a test generation tool is able to generate test suites in a flexible manner, for a wide variety of different coverage criteria. In other words, a test generation tool should accept

a simple specification of a coverage criterion, given in a language that can easily specify a large set of coverage criteria, and be able to generate test suites accordingly.

In this paper, we present a technique for specifying coverage criteria in a simple and flexible manner, and a method for generating test cases according to such coverage criteria. The technique fits well as an extension of a state-space exploration tool, such as, e.g., SPIN [11] or UPPAAL [16], which performs enumerative or symbolic state-space exploration. It can also be used to generate monitors that measure the coverage of a specific test suite by monitoring the test execution.

In our technique, a coverage criterion is given as a set of *coverage items*, each of which represents an interesting structural property of the EFSM which should be examined by a test suite. A coverage item can state that a particular state, edge, or similar, should be visited, it can be an explicit test purpose, etc. Each coverage item is specified by an *observer*, which observes the execution of a test case, and reports acceptance when the test case has *covered* the coverage item that it specifies. For instance, a coverage item stating that a control state l of an EFSM model should be visited simply observes how the EFSM executes and reports acceptance when it enters l.

A typical coverage criterion is given as a (often rather large) set of coverage items. An important mechanism to facilitate specification of many coverage criteria is to allow *parameterization* of observers. In this way, one can specify a set of coverage items parameterized over, e.g., control states, data variables, edges, etc. of the EFSM model. Using this simple and general mechanism, we can specify most of the coverage criteria that have been used in the literature, and also tailor coverage to specific features of a particular SUT. For instance, if a particular interface is very error prone, we can specify a coverage criterion which requires all possible interleavings of interactions on that interface to be exhibited in a test suite.

A specification of a coverage criterion can be used for test suite generation using a state-space exploration tool. First, we superpose the coverage observers onto the EFSM, then we search for a test sequence or set of test sequences in which as many observers as possible report acceptance. For parameterized observers, we can record the achieved coverage by a (typically small) set of bitvectors, indexed by parameter values, which concisely represent the states of a large set of parameterized observers, in analogy with bitvector analysis in data-flow analysis, e.g., [20]. The same machinery can also be used to monitor the achieved coverage of a certain test suite.

The remainder of the paper is structured as follows. We present EFSMs in the next section, and observers in Section 3. In Section 4, we show how our definitions of coverage can be used for test case generation, and report on a partial implementation of the technique. Section 5 concludes the paper.

Related Work. Most related work on test case generation from models of reactive systems employ some rather specific selection of coverage criteria. Explicitly given test purposes have been considered, both enumerative [5] and symbolic [23].

Test purposes in these works can in some sense be regarded as coverage observers, but are not used to specify more generic coverage criteria and do not make us of parameterization, as in our work. For finite-state machines and EFSMs, several approaches focus on particular coverage criteria, e.g., Bouquet and Legeard [1] synthesize test cases corresponding to combinations of choices of control flow and boundary values of state variables, Nielsen and Skou [21] generate test cases that cover reachable symbolic states. These coverage criteria can be specified as observers in our framework.

Some approaches present more flexible techniques for specifying a variety of coverage criteria. Hong et al [13, 12] describe how flow-based coverage criteria can be expressed in temporal logic. A particular coverage item is expressed in CTL, and a model checker generates a trace which covers the coverage item. In our approach, we use observers instead of temporal logic, which avoids some of the limitations of temporal logic [26]. Friedman et al [6] specifies coverage by giving a set of projections of the state space (e.g., on individual state variables, components of control flow) that should be covered, possibly under some restrictions. Our approach generalizes this one, by allowing to define observers. Also, we can let one pass of a state-space exploration tool generate a test suite that covers a large set of coverage items, whereas the above approaches invoke a run of a model checker for each coverage item.

Constraint Logic Programming for model based test case generation has been used, e.g., by Marre and Arnould [18], by Meudec [19], by Pretschner et al. [22]. These approaches typically compile the specification into a constraint logic programming language, in which test cases can be extracted using symbolic execution.

2   Extended Finite State Machines

We assume that the specification of a module to be tested is given as an extended finite state machine in some syntax. In this section, we present a generic way to describe EFSMs, but our work can be adapted to more specific EFSM notations such as, e.g., UML Statecharts [7] or SDL [14].

We assume that a System Under Test (SUT) interacts with its environment through events. Whenever the SUT receives an input event, it responds by performing some local computation and emitting an output event. To a given SUT, we associate a set A of *event types*, each with a fixed *arity*. An *event* is a term of form $a(d_1, \ldots, d_k)$ where a is an event type of arity k and $d_1, \ldots, d_k$ are the parameters of the event. The set A of event types is partitioned into *input event types* and *output event types*. A *trace* is a finite sequence

$$a_1(\overline{d}_1)/b_1(\overline{d}'_1) \quad a_2(\overline{d}_2)/b_2(\overline{d}'_2) \quad \cdots \quad a_n(\overline{d}_n)/b_n(\overline{d}'_n)$$

of input/output event pairs. Intuitively, the trace represents a behavior where the SUT, starting from its initial state, receives the input event, $a_1(\overline{d}_1)$ and responds with the output event $b_1(\overline{d}'_1)$. Thereafter, it receives the input event $a_2(\overline{d}_2)$ and so on. An *input sequence* is a finite sequence of input events.

Assume a set A_I of input event types, and a set A_O of output event types. An *Extended Finite State Machine* (EFSM) over (A_I, A_O) is a tuple $\langle L, l_0, \overline{v}, E \rangle$ where

- L is a finite set of *locations* (aka control states).
- $l_0 \in L$ is the *initial location.*
- $\overline{v}$ is a finite set of *state variables.*
- E is a finite set of *edges*, each of which is of form

$$e: \quad \overbrace{l} \xrightarrow{a(\overline{w}), g \rightarrow \overline{u} := \overline{expr}/b(\overline{expr'})} \overbrace{l'}$$
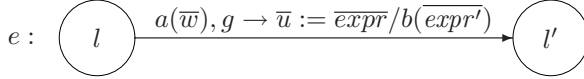
where
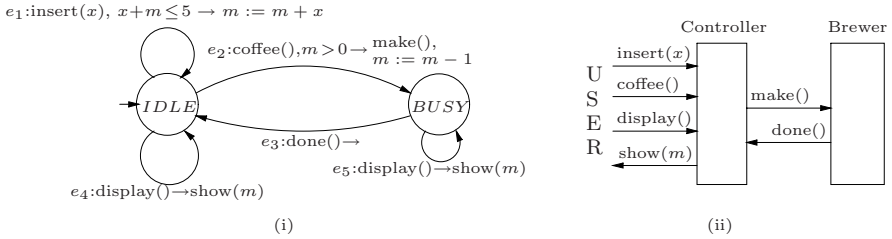- e is the name of the edge,
- l is the source location, and l' is the target location,
- $a \in A_I$ is an input event type, and $\overline{w}$ is a tuple of formal parameters of a,
- g is a guard,
- $\overline{u} := \overline{expr}$ is an assignment of new values to a subset $\overline{u} \subseteq \overline{v}$ of the state variables, and
- $b(\overline{expr'})$ is an expression which evaluates to an output event.

g, $\overline{expr}$, and $\overline{expr'}$ may depend on the formal parameters $\overline{w}$ of the input event and the state variables $\overline{v}$.

Intuitively, an edge of the above form denotes that whenever the EFSM is in location l and receives an event of form $a(\overline{w})$, then, provided that the guard g is satisfied, it can perform a computation step in which it updates its state variables by $\overline{u} := \overline{expr}$, emits the output event $b(\overline{expr'})$ and moves to location l'. We require the EFSM to be *deterministic*, i.e., that for any two edges with the same source location l and parameterized input event $a(\overline{w})$, the corresponding guards are inconsistent.

A *system state* is a tuple $\langle l, \sigma \rangle$ where l is a location, and σ is a mapping from $\overline{v}$ to values. We can extend σ to a partial mapping from expressions over $\overline{v}$ in the standard way. The *initial system state* is the tuple $\langle l_0, \sigma_0 \rangle$ where l_0 is the initial location, and σ_0 gives a default value to each state variable. A *computation step* is of the form $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})/b(\overline{d'})} \langle l', \sigma' \rangle$ consisting of system states $\langle l, \sigma \rangle$ and $\langle l', \sigma' \rangle$, an input event $a(\overline{d})$, and an output event $b(\overline{d'})$, such that there is an edge of the (above) form $l \xrightarrow{a(\overline{w}), g \rightarrow \overline{u} := \overline{expr}/b(\overline{expr'})} l'$, for which $\sigma(g[\overline{d}/\overline{w}])$ is true, $\sigma' = \sigma[\overline{u} \mapsto \sigma(\overline{expr}[\overline{d}/\overline{w}])]$, and $\overline{d'} = \sigma(\overline{expr'}[\overline{d}/\overline{w}])$. A *run* of the EFSM over a trace $a_1(\overline{d}_1)/b_1(\overline{d'}_1) \cdots a_n(\overline{d}_n)/b_n(\overline{d'}_n)$ is a sequence of computation steps

$$\langle l_0, \sigma_0 \rangle \xrightarrow{a_1(\overline{d}_1)/b_1(\overline{d'}_1)} \langle l_1, \sigma_1 \rangle \xrightarrow{a_2(\overline{d}_2)/b_2(\overline{d'}_2)} \cdots \xrightarrow{a_n(\overline{d}_n)/b_n(\overline{d'}_n)} \langle l_n, \sigma_n \rangle$$

labelled by the input-output event pairs of the trace.

e_1:insert(x), $x+m \leq 5 \rightarrow m := m + x$

Controller   Brewer

e_2:coffee$()$,$m>0 \rightarrow$ make$()$,
$m := m-1$

IDLE   BUSY

e_3:done$() \rightarrow$

e_5:display$() \rightarrow$show(m)

e_4:display$() \rightarrow$show(m)

U
S
E
R

insert(x)
coffee$()$
display$()$
show(m)

make$()$
done$()$

(i)   (ii)

Fig. 1. An EFSM specifying the controller of a simple coffee machine

Example 1. In Fig. 1 an EFSM (from [13]) specifying the behavior of the controller of a simple coffee machine which interacts with a user and a brewer unit is shown. The controller has $L = \{IDLE, BUSY\}$, $l_0 = IDLE$, $\overline{v} = \{m\}$, $A_I = \{insert, coffee, display, done\}$, $A_O = \{show, make\}$, and $E = \{e_1, e_2, e_3, e_4, e_5\}$. The parameter x and the variable m take values that are integers in the range $[0 \ldots 5]$.

An EFSM can be used to check that a trace of a SUT conforms to its specification, by checking that each output event produced by the SUT conforms to the corresponding output event prescribed in the EFSM. For test generation, the output events will not be significant, and we will therefore omit them in the rest of the paper, thus writing an edge of an EFSM as $l \xrightarrow{a(\overline{w}),g \rightarrow \overline{u}:=\overline{expr}} l'$. We can also consider specifications that are parallel compositions of EFSMs, but omit such a treatment in this version of the paper.

3 Observers

In this section, we present how to use observers to specify coverage criteria for test generation or test monitoring. A coverage criterion typically consists of a (long) list of items that should be "covered" or "visited". For instance, the criterion of "full location coverage" stipulates that a test suite should visit all locations of a given EFSM. We will use the term *coverage item* for an item that should be "covered" or "visited". Letting a test sequence be represented as a trace, we can use standard techniques from model-checking and run-time verification [25, 8] to represent a coverage item by an *observer*, which monitors a trace and "accepts" whenever the coverage item has been covered. An observer observes how an EFSM executes a run over a trace, and "remembers" some chosen aspects of the EFSM execution. The observer can observe the events of the trace, as well as syntactical components of edges that the EFSM traverses in response to observed events, but should not interfere with the execution of the system.

Typical coverage criteria consist not only of a single coverage item, but of a large set of coverage items. We therefore extend the notion of observers by a

parameterization mechanism so that they can specify a *set of* coverage items. Parameterized observers are simply observers, in which locations and edges may be parameterized by parameters that range over given domains. Each choice of parameter values gives a certain observer location or edge. For each specified coverage item, the observer has an *accepting* (possibly parameterized) location which (for convenience) we give the name of the corresponding coverage item. When the accepting location is entered, the trace has covered the corresponding coverage item.

As a very simple example, the coverage item *"visit location l of the EFSM"* can be represented by an observer with one initial state, and one accepting location, named $loc(l)$, which is entered when the EFSM enters location l. The coverage criterion *"visit all locations of the EFSM"* can be represented by a parameterized observer with one initial state, and one parameterized accepting location, named $loc(\mathtt{L})$, where $\mathtt{L}$ is a parameter that ranges over locations in the EFSM. For each value l of $\mathtt{L}$, the location $loc(l)$ is entered when the EFSM enters location l.

Formally, an *observer* is a tuple (Q, q_0, Q_f, B) where

- Q is a finite set of *observer locations*
- q_0 is the *initial observer location*.
- $Q_f \subseteq Q$ is a set of *accepting observer locations*, whose names are the corresponding coverage items.
- B is a set of edges, each of form
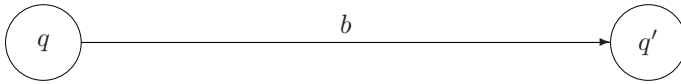
where b is a predicate that can depend on the input event received by the SUT, the mapping from state variables of EFSM to their values after performing the current computation step, and the edge in the EFSM that is executed in response to the current input event.

Intuitively, at any specific instant during test execution the observer is in one of its locations, q say. At each occurrence of an event, the observer traverses an outgoing edge from q, whose predicate is satisfied for this event, and the corresponding transition performed by the EFSM. Note that, in contrast to EFSMs, observers may be non-deterministic, since a coverage item in general can be covered in several ways.

In many cases, the initial location q_0 has an edge to itself with the predicate *true*. We use the symbol $\bullet$ to represent q_0 together with such a self-loop. Similarly, we assume that each $q_f \in Q_f$ has an edge to itself with the predicate *true*. We use the symbol $\odot$ to represent accepting locations. In section 3.2, we discuss the effect of these self-loops in more detail. Intuitively, the one in q_0 is often used to allow the observer to non-deterministically start monitoring at any point of an EFSM run. The loop in each q_f is used to allow an observer to stay in an accepting location.

In order for observers to specify coverage criteria consisting of several coverage items, we allow locations and edges to be parameterized. Each parameter has a finite domain, which could be the set of EFSM locations, edges, state variables, or similar. We use uppercase letters in typewriter font for parameters. A parameterized location represents the collection of locations obtained by instantiating its parameters, and similarly for edges.

3.1   Observer Predicates

In the following we introduce a more specific syntax for the predicates b occurring on observer edges. The predicates will use a set of predefined *match variables* that are given values at the occurrence of

- an event $a(\overline{d})$,
- an edge $e : l \xrightarrow{a(\overline{w}),g \rightarrow \overline{u}:=\overline{expr}} l'$ of the EFSM, traversed in response to $a(\overline{d})$,
- the computation step $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})} \langle l', \sigma' \rangle$ generated in response to $a(\overline{d})$.

For a traversed EFSM edge we use the following match variables (with associated meaning):

event_type	is the event type a of the occurring event
event-pars	is the list $\overline{d}$ of parameters of the event
edge	is the name e
target_loc	is the target location l'
guard	is the guard expression g
assignments	is the set $\overline{u} := \overline{expr}$ of assignments
target_val	is the function from EFSM state variables to values, s.t. $val(u)$ is the value $\sigma'(u)$ of variable u just after the computation step.

Similarly, we also define *source_loc* for the source location and *source_val* for the value $\sigma(u)$ of variable u just before the computation step. To be able to express more interesting properties we also introduce a set of operations that can be used together with the match variables:

- *lhs* is a function to get the left hand side expression of an assignment. A left hand side expressions is always assumed to be a variable.
- *rhs* is a function to get the right hand side expressions of an assignment. The right hand side expression, *expr*, uses the vocabulary defined for the EFSM specification.
- *vars* is a function such that $vars(Exp)$ returns a set with all variables found in Exp. Exp is a set that contains the result of applying *rhs* to each assignment in *assignments*, or a *guard* expression.
- *affect* is a function such that $affect(A, Var_1, Var_2)$ returns the assignment it is being applied to, A, if $Var_1 \in vars(rhs(A)) \wedge Var_2 = lhs(A)$ otherwise the empty set is returned.
- *map* is a function such that $map(Fun, Set)$ applies the function, Fun on each element in the set Set and returns the set of the results.
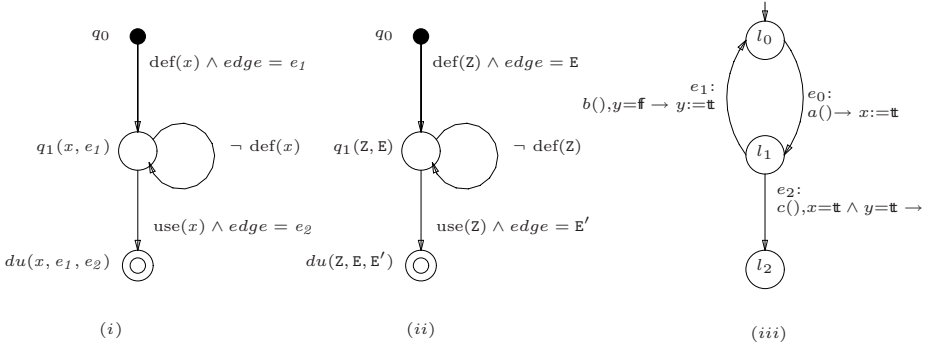
Fig. 2. Examples of (i) observer monitoring definition (on edge e_1) and use (on edge e_2) of variable x, (ii) a parameterized observer, and (iii) a simple EFSM

With the match variables and operations above we define new functions that can be used as tests in the observer. In this paper, we shall make use of:

- $def(v)$, which is true iff the variable v is defined by the transition in the EFSM. This can be expressed as:

$$v \in map(lhs, assignments)$$

- $use(v)$, which is true iff the variable v is used (in a guard or assignment) by the transition in the EFSM. This can be expressed as:

$$v \in vars(map(rhs, assignments)) \lor v \in vars(guard)$$

- $da(v_1, v_2)$, which is true iff the variable v_1 is on the right hand side and variable v_2 is on the left hand side of the same assignment in the EFSM specification. The function can intuitively be understood to be true if v_1 directly affects v_2. This can be expressed as:

$$map(affect(v_1, v_2), assignments) \neq \emptyset$$

Example 2. The (non-parameterized) observer in Fig. 2(i) specifies definition-use pair coverage for a specific variable m, and specific edges e_1 and e_2. Fig. 2(ii) shows a corresponding (parameterized) observer that specifies definition-use pair coverage for *any* EFSM variable Z, and EFSM edges E and E'. This is done by parameterizing the location q_1 with any variable and any edge, and the accepting location du with any variable and any two edges. The edges are parameterized in a similar way. For example, there is one observer edge from location $q_1(z, e)$ to location $du(z, e, e')$ for each EFSM variable z, and each pair e, e' of EFSM edges.

3.2   How Observers Monitor Coverage Criteria

In test case generation or when monitoring test execution of a SUT, an observer observes the events of the SUT, and the computation steps of the EFSM. Reached accepting locations correspond to covered coverage items. We formally define the execution of an observer in terms of a composition between an EFSM and an observer, which has the form of a *superposition* of the observer onto the EFSM. Each state of this superposition consists of a state of the EFSM, together with a *set* of currently occupied observer locations.

Say that a predicate b on an observer edge is satisfied by a computation step $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})} \langle l', \sigma' \rangle$ of an EFSM, denoted $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})} \langle l', \sigma' \rangle \models b$ if b holds for the event $a(\overline{d})$, the computation step $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})} \langle l', \sigma' \rangle$, and the edge $e :$ $l \xrightarrow{a(\overline{w}), g \rightarrow \overline{u} := \overline{expr}} l'$ from which the computation step is derived.

Formally, the superposition of an observer (Q, q_0, Q_f, B) onto an EFSM $\langle L, l_0, \overline{v}, E \rangle$ is defined as follows.

- *States* are of the form $\langle \langle l, \sigma \rangle \parallel \mathcal{Q} \rangle$, where $\langle l, \sigma \rangle$ is a state of the EFSM, and $\mathcal{Q}$ is a set of locations of the observer.
- The *initial state* is the tuple $\langle \langle l_0, \sigma_0 \rangle \parallel \{q_0\} \rangle$, where $\langle l_0, \sigma_0 \rangle$ is the initial state of the EFSM, and q_0 is the initial location of the observer.
- A *computation step* is a triple $\langle \langle l, \sigma \rangle \parallel \mathcal{Q} \rangle \overset{a(\overline{d})}{\rightsquigarrow} \langle \langle l', \sigma' \rangle \parallel \mathcal{Q}' \rangle$ such that $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})} \langle l', \sigma' \rangle$ and

$$\mathcal{Q}' = \left\{ q' \mid q \xrightarrow{b} q' \text{ and } q \in \mathcal{Q} \text{ and } \langle l, \sigma \rangle \xrightarrow{a(\overline{d})} \langle l', \sigma' \rangle \models b \right\}$$

- A state $\langle \langle l, \sigma \rangle \parallel \mathcal{Q} \rangle$ of the superposition *covers* the coverage item represented by the location $q_f \in Q_f$ if $q_f \in \mathcal{Q}$.

Note that the way the set $\mathcal{Q}$ is updated essentially results in an (on-the-fly) subset construction of the parameterised observer. Initially, $\mathcal{Q}$ contains only the initial observer location q_0. In the subsequent computation steps, $\mathcal{Q}$ contains the set of all occupied observer locations, representing already covered and partially covered coverage items. In each computation step, the set of occupied observer locations $\mathcal{Q}'$ is obtained by generating all possible successors to the locations in $\mathcal{Q}$, i.e. all q' such that there exists a $q \in \mathcal{Q}$ and an edge $q \xrightarrow{b} q' \in B$ with b satisfied by the computation step of the EFSM.

Recall that both the initial and all accepting observer locations have implicit self-loops with predicate *true*. This means that in the superposition of the observer onto an EFSM, the initial observer location q_0 is always occupied and all reached accepting observer locations (representing covered coverage items) are guaranteed to remain in $\mathcal{Q}$. The fact that q_0 is always occupied can be intuitively understood as allowing for the observer to non-deterministically start monitoring an EFSM (or a SUT) at *any* computation step of an run (or at any point during test execution).
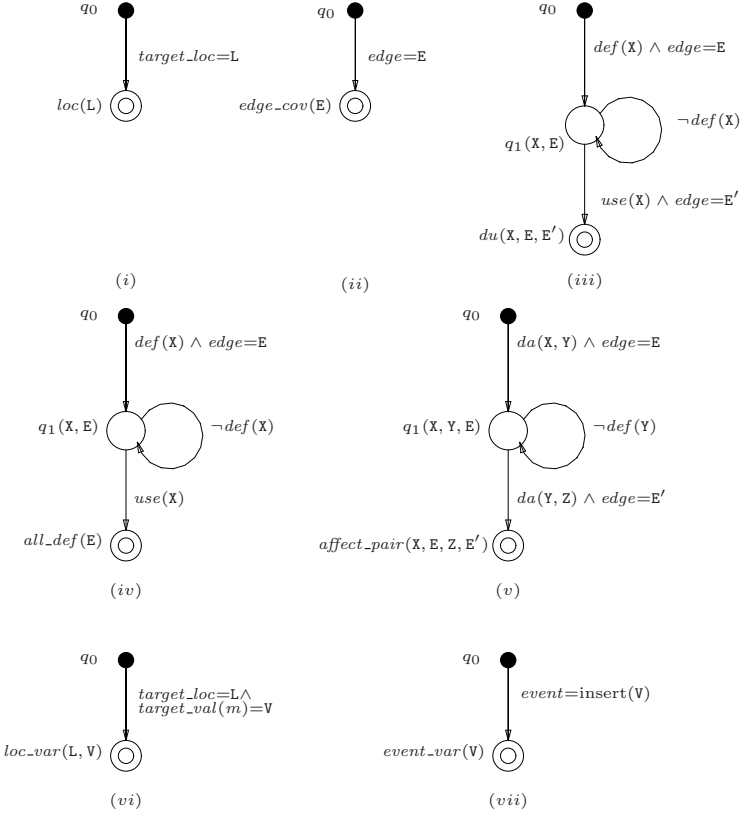
Fig. 3. Seven examples of coverage criteria expressed as observers

Example 3. If the observer in Fig. 2(ii) is superposed onto the EFSM in Fig. 2(iii), the following computation steps can be taken $\langle\langle l_0, \{x = \mathbf{f}, y = \mathbf{f}\}\rangle \parallel \{q_0\}\rangle \overset{a()}{\rightsquigarrow}$ $\langle\langle l_1, \{x{=}\mathbf{t}, y{=}\mathbf{f}\}\rangle \parallel \{q_0, q_1(x, e_0)\}\rangle \overset{b()}{\rightsquigarrow} \langle\langle l_0, \{x{=}\mathbf{t}, y{=}\mathbf{t}\}\rangle \parallel \{q_0, q_1(x, e_0), q_1(y, e_1)\}\rangle$ $\overset{a()}{\rightsquigarrow} \langle\langle l_1, \{x = \mathbf{t}, y = \mathbf{t}\}\rangle \parallel \{q_0, q_1(x, e_0), q_1(y, e_1)\}\rangle \overset{c()}{\rightsquigarrow} \langle\langle l_2, \{x = \mathbf{t}, y = \mathbf{t}\}\rangle \parallel$ $\{q_0, q_1(x, e_0), q_1(y, e_1), du(x, e_0, e_2), du(y, e_1, e_2)\}\rangle$. Thus, the two possible definition-use pairs are covered.

3.3   Examples of Observers

Fig. 3 shows observers specifying a number of coverage criteria described in the literature [2].

The *all-locations* coverage criteria is specified by the observer shown in Fig. 3(i), where the parameter L is any location in an EFSM. If the observer is superposed onto the EFSM of Fig. 1, we have that L = $\{IDLE, BUSY\}$ and the edge of the parameterized observer represents two edges, one guarded by

$target\_loc = IDLE$ with target location $loc(IDLE)$ $target\_loc(BUSY)$, and the other guarded by $target\_loc = BUSY$ with target location $loc(BUSY)$. The set of possible coverage items is thus $\{loc(IDLE), loc(BUSY)\}$.

The *all-edges* coverage observer in Fig. 3(ii) is similar to the all-location coverage observer. The edges of the EFSM in Fig. 1 is $\mathrm{E}=\{e_1, \ldots, e_5\}$, and thus the set of possible coverage items when the observer is superposed onto the EFSM is $\{edge\_cov(e_i) \mid e_i \in \mathrm{E}\}$.

The *all-definition use-pairs* (all-uses [2]) coverage observer in Fig. 3(iii) has an accepting location $du(\mathrm{X}, \mathrm{E}, \mathrm{E}')$, where X is a variable name, E is an edge on which X is defined, and E' an edge on which X is used. Variable X may not be redefined in the trace between E and E'. If the observer is superposed onto the EFSM the complete set of coverage items is $\{du(m, e_1, e_1),\ du(m, e_1, e_2),\ du(m, e_1, e_4),\ du(m, e_2, e_1),\ du(m, e_2, e_2),\ du(m, e_2, e_4),\ du(m, e_2, e_5)\}$. The definition-use pair $du(m, e_1, e_5)$ can not be covered since m is always redefined on edge e_2 in between e_1 and e_5.

The *all-definitions* coverage observer of Fig. 3(iv) is similar to the all-definition use-pairs coverage except that only the defining edges are required to be covered. When the observer is superposed with the EFSM in Fig. 1 the set of accepting locations is $\{all\_def(e_1), all\_def(e_2)\}$.

The *all affect-pairs* (Nafos' required k-Tuples [2]) coverage observer shown in Fig. 3(v) accepts whenever a variable x affects a variable z via another variable y. In this case we require that x directly affects y which, without redefinition, directly affects z. No such affect pairs are possible in the EFSM of Fig. 1.

The *context coverage* criteria observer in Fig. 3(vi) covers all values of a given variable m. We use $target\_val(m)$, to denote the value of m at the target *EFSM-state*. The observer has an accepting location $loc\_var(\mathrm{L}, \mathrm{V})$, where V is the value domain of variable m. E.g. $loc\_var(IDLE, 0)$ and $loc\_var(BUSY, 1)$ are accepting locations. The observer in Fig. 3(vii) is similar, but covers the possible values the event parameter at transitions labelled with the event insert(x).

4 Test Case Generation

4.1 Algorithms

At test case generation, we use the superposition of an observer onto an EFSM, and views the test case generation problem as a search exploration problem. To cover a coverage item q_f is then the problem of finding a trace

$$tr = \langle\langle l_0, \sigma_0\rangle \parallel \{q_0\}\rangle \overset{a(\overline{d})}{\leadsto} \ldots \overset{a'(\overline{d}')}{\leadsto} \langle\langle l, \sigma\rangle \parallel \mathcal{Q}\rangle \text{ such that } q_f \in \mathcal{Q}$$

We will use $\omega(tr) = a(\overline{d}) \ldots a'(\overline{d}')$ to denote the *word* of the trace tr, or just ω whenever tr is clear from the context. In general, a single trace tr may cover several accepting locations of the observer. We say that the trace tr covers n accepting observer states if there are n accepting states in $\mathcal{Q}$, and we use $|Q_f \cap \mathcal{Q}|$ to denote the number of accepting states in $\mathcal{Q}$.

We are now ready to present the test case generation algorithm. We shall limit the presentation to an algorithm generating a single trace. The same technique can be used to produce sets of traces to cover many coverage items. Alternatively, the EFSM model can be annotated with edges that reset the EFSM to its initial state. A generated trace can then be interpreted as a set of test cases separated by the reset edges [9].

An abstract algorithm to compute test case is shown in Fig. 4. To improve the presentation, we use s to denote a system of the form $\langle l, \sigma \rangle$, s_0 to denote the initial system state $\langle l_0, \sigma_0 \rangle$, and a to denote an input action $a(\bar{d})$. The algorithm computes the maximum number of coverage items that can be visited (MAX), and returns a trace with maximum coverage (ω_{max}). The two main data structures WAIT and PASS are used to keep track of the states waiting to be explored, and the states already explored, respectively.

PASS:= $\emptyset$, MAX := 0, ω_{max} := ω_0
WAIT:= $\{\langle\langle s_0 \parallel \{q_0\}\rangle, \omega_0\rangle\}$
while WAIT$\neq \emptyset$ **do**
    select $\langle\langle s \parallel \mathcal{Q}\rangle, \omega\rangle$ from WAIT
    if $|Q_f \cap \mathcal{Q}| >$ MAX **then**
        $\omega_{max} := \omega$, MAX := $|Q_f \cap \mathcal{Q}|$
    if for all $\langle s \parallel \mathcal{Q}'\rangle$ in PASS: $\mathcal{Q} \not\subseteq \mathcal{Q}'$ **then**
        add $\langle s \parallel \mathcal{Q}\rangle$ to PASS
        for all $\langle s'' \parallel \mathcal{Q}''\rangle$
            such that $\langle s \parallel \mathcal{Q}\rangle \overset{a}{\rightsquigarrow} \langle s'' \parallel \mathcal{Q}''\rangle$:
            add $\langle\langle s'' \parallel \mathcal{Q}''\rangle, \omega a\rangle$ to WAIT
return ω_{max} and MAX

Fig. 4. An abstract breadth-first search exploration algorithm for test case generation

Initially, the set of already explored states is empty and the only state waiting to be explored is the extended state $\langle\langle s_0 \parallel \{q_0\}\rangle, \omega_0\rangle$, where ω_0 is the empty trace. The algorithm then repeatedly examines extended states from WAIT. If a state $\langle s \parallel \mathcal{Q}\rangle$ found in WAIT is included in a state $\langle s \parallel \mathcal{Q}'\rangle$ in PASS, then obviously $\langle s \parallel \mathcal{Q}\rangle$ does not need to be further examined. If not, all successor states reachable from $\langle s \parallel \mathcal{Q}\rangle$ in one computation step are put on WAIT, with their traces extended with the input action of the computation step from which they are generated. The state $\langle s \parallel \mathcal{Q}\rangle$ is saved in PASS. The algorithm terminates when WAIT is empty

The variables ω_{max} and MAX are initially set to the empty trace and 0, respectively. They are updated whenever an extended state is found in WAIT which covers a higher number of coverage items than the current value of MAX. Throughout the execution of the algorithm, the value of MAX is the maximum number of coverage items that have been covered by a single trace, and ω_{max} is one such trace. When the algorithm terminates, the two values MAX and ω_{max} are returned.

4.2   Bitvector Implementation

In order to efficiently represent and manipulate the set $\mathcal{Q}$ of observer locations we shall use bitvector analysis [15]. Let the set $\mathcal{Q}$ be represented by a bitvector where each bit represents an observer location q'. Then each bit is updated by the following function

$$f_{q'}(q') = \bigvee_{\langle b,q \rangle \in \; in(q')} q \wedge b$$

where $in(q') = \{ \langle b, q \rangle \mid q \xrightarrow{b} q' \in B \}$ is the set of pairs of predicates b and source locations q of the edges ingoing to the location q'. That is, given a state of the superposition $\langle \langle l, \sigma \rangle \parallel \mathcal{Q} \rangle$ and an EFSM-transition $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})} \langle l', \sigma' \rangle$ the bit representing q' is set to 1 if there is an observer edge $q \xrightarrow{b} q' \in B$, such that $q \in \mathcal{Q}$ and $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})} \langle l', \sigma' \rangle \models b$. Otherwise the bit representing q' is set to 0. It should be obvious that this corresponds precisely to the semantics of an observer superposed onto an EFSM, described in Section 3.2.

Example 4. When the observer in Fig. 2(*ii*) is superposed onto the EFSM in Fig. 2(*iii*), we have $\mathtt{E} = \mathtt{E}' = E = \{e_0, e_1, e_2\}$ and $\mathtt{Z} = \overline{v} = \{x, y\}$. Thus, we have that

$$Q = \{ q_0 \} \cup \{ q_1(z, e_a) \mid z \in \overline{v} \wedge e_a \in E \} \cup \{ du(z, e_a, e_b) \mid z \in \overline{v} \wedge e_a, e_b \in E \}$$

Any enumeration of the set can be used as index in the bitvector. As the observer has three locations with parameters we get three types of bitvector functions:

$$f_{q_0}(q_0) = q_0 \wedge \mathtt{tt} \tag{1}$$
$$f_{q_1(v_i, e_j)}(q_1(v_i, e_j)) = (\; q_0 \wedge def(v_i) \wedge (edge = e_j)\;) \vee$$
$$(\; q_1(v_i, e_j) \wedge \neg def(v_i)\;) \tag{2}$$
$$f_{du(v_i, e_j, e_k)}(du(v_i, e_j, e_k)) = (\; q_1(v_i, e_j) \wedge use(v_i) \wedge (edge = e_k)\;) \vee$$
$$(\; du(v_i, e_j, e_k) \wedge \mathtt{tt}\;) \tag{3}$$

There is one function of type (1), six of type (2), and 18 of type (3). Note that (1) is always true and that (3) will remain true once it becomes true, due to implicit self-loops in these locations.

4.3   Implementation Efforts

Some of the techniques presented in this paper have been implemented in a prototype version of the model-checking tool UPPAAL [16], extended for test case generation [10]. The current implementation uses the bitvector implementation described above, but is limited to a number of predefined coverage criteria. For a given coverage criteria (a set of) test cases can be generated from system specifications described as DIEOU-timed automata [9]. We are currently in progress with a larger case-study in collaboration with Ericsson where this tool will be applied.

We are also developing a tool operating on a subset of the functional language Erlang, also using the techniques presented in this paper. The tool will be applied in a case-study in collaboration with Mobile Arts.

5   Conclusions

We have presented a technique for specifying coverage criteria in a simple and flexible manner using observer automata with parameters. Observers have shown to be a flexible tool in model checking and run-time monitoring, and by this paper we have shown that they are a versatile tool for specifying coverage criteria for test case generation and test monitoring. In particular the parameterization mechanism, as used in this paper, allows a succinct specification of several standard generic coverage criteria. In this way, test case generation can be transformed into a reachability problem, which can be attacked by a standard state-space reachability tool.

In previous works, we have implemented special cases of this test case generation technique, using UPPAAL, indicating that the approach is practical. We are currently working on a general implementation of the observer concept, and plan to apply it in a larger case study.

References

1. F. Bouquet and B. Legeard. Reification of executable test scripts in formal specification-based test generation: The java card transaction mechanism case study. In *FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 778–795. Springer Verlag, 2003.
2. Lori A. Clarke, Andy Podgurski, Debra J. Richardsson, and Steven J. Zeil. A formal evaluation of data flow path delection criteria. *IEEE Trans. on Software Engineering*, SE-15(11):1318–1332, Nov 1989.
3. L. du Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. Belinfante, and R.G. de Vries. Formal test automation: The conference protocol with tgv/torx. In H. Ural, R.L. Probert, and G. von Bochmann, editors, *IFIP 13th Int. Conference on Testing of Communicating Systems(TestCom 2000)*. Kluwer Academic Publishers, 2000.
4. L. du Bousquet and N. Zuanon. An overview of Lutess, a specification-based tool for testing synchronous software. In *Proc. 14th IEEE Intl. Conf. on Automated SW Engineering*, October 1999.
5. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29, 1997.
6. G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 134–143, 2002.
7. Stefania Gnesi, Diego Latella, and Mieke Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *Journal of Logic and Algebraic Programming*, 51(1):43–75, 2002.
8. K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02, 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 324–356. Springer Verlag, 2002.

9. Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-Optimal Real-Time Test Case Generation using UPPAAL. In Alexandre Petrenko and Andreas Ulrich, editors, *Proc. of 3*rd *International Workshop on Formal Approaches to Testing of Software*, number 2931 in Lecture Notes in Computer Science, pages 136–151. Springer Verlag, 2003.

10. Anders Hessel and Paul Pettersson. A test generation algorithm for real-time systems. To appear in Proc. of 4th Int. Conf. on Quality Software, Sept. 2004.

11. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.

12. H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE'03: 25th Int. Conf. on Software Enginering*, pages 232–242, May 2003.

13. H.S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage. In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02, 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer Verlag, 2002.

14. ITU, Geneva. *ITU-T, Z.100, Specification and Description Language (SDL)*, Nov 1999.

15. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, 1996.

16. K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1-2), 1997.

17. D. Lugato, C. Bigot, and Y. Valot. Validation and automatic test generation on UML models: the AGATHA approach. In *Proc. 7th Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS 02), Electronic Notes in Theoretical Computer Science*, volume 66, 2002.

18. B. Marre and A. Arnould. Test Sequence Generation from Lustre Descriptions: GATEL. In *Proc. 15th IEEE Intl. Conf. on Automated Software Engineering (ASE'00)*, Grenoble, 2000.

19. C. Meudec. ATGen: Automatic test data generation using constraint logic programming and symbolic execution. In *Proc. 1st Intl. Workshop on Automated Program Analysis, Testing, and Verification*, Limerick, 2000.

20. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

21. Brian Nielsen and Arne Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer*, 5:59–77, 2003.

22. A. Pretschner. Classical search strategies for test case generation with constraint logic programming. In *Proc. Formal Approaches to Testing of Software, FATES '01*, pages 47–60, Aalborg, Denmark, August 2001.

23. V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *Int. Conf. on Integrating Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer Verlag, 2000.

24. M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Koch. Autolink - putting sdl-based test generation into practice. In *11th Int. Workshop on Testing of Communicating Systems (IWTCS'98)*, Tomsk, Russia, Sept. 1998.

25. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86, 1st IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

26. P. Wolper. Temporal logic can be more expressive. In *Proc. 22nd Annual Symp. Foundations of Computer Science*, pages 340–348, 1981.