Checking States and Transitions of a set of Communicating Finite State Machines

Rob M. Hierons Brunel University

June 19, 2000

Abstract

Given a model M, consisting of communicating finite state machines (CFSMs), that represents the required behaviour of an implementation I, it is important to test I against M. This paper considers part of the testing process: checking the transition structure of I against that of M. One possible approach, to checking the transition structure of I, is to generate the product machine from M and then test the global transitions using standard finite state machine test techniques. This approach may, however, suffer from a combinatorial explosion. Instead, this paper introduces approaches that may allow local states and transitions of I to be checked without the generation of the product machine. The paper then considers the extension of these approaches to the checking of global states.

Keywords: communicating finite state machines, state checking, testing, constrained identification sequences

1 Introduction

Testing is an important, but expensive, part of the software development process. However, the presence of a formal model or specification, that defines the required behaviour of a system, introduces the possibility of automating or semi-automating much of the testing process. This can lead to more effective and efficient testing.

There are a number of approaches to formally modelling, or specifying, a system. One approach represents a system as a finite set of logical states, possibly with an internal memory. Operations are modelled as actions, called *transitions*, that receive input, produce output, may change the logical state and may change the internal memory. Where there is no internal memory, and the actions are simply input/output pairs, this is a *finite state machine (FSM)*. Where there is an internal memory the model is an *extended finite state machine (EFSM)*.

FSMs and EFSMs have proved effective in modeling a variety of systems and forms of EFSMs have been proposed as the basis of a general software development process ([23], [9]). Since the structure of an EFSM may be represented by an FSM, FSM based test methods are often used to test the control structure of an implementation under test (IUT) modelled as an EFSM. There has thus been much interest in the automatic generation of tests from FSMs ([1, 22, 4, 5, 7, 18, 8, 10]).

Many systems may be more naturally and simply modelled by a set of FSMs, rather than a single FSM, that operate concurrently and may interact by exchanging messages. The FSMs, that may communicate, are often given input queues and are called *communicating finite state machines (CFSMs)*. Two CFSMs interact by one machine producing an output that is placed in the input queue of the other. CFSMs have been used to describe the control structure of specifications written in languages such as Statecharts and SDL ([2], [12]). CFSMs are thus relevant to a number of fields including embedded systems ([20, 16, 3]) and communications protocols ([19, 17]).

Under certain conditions a model consisting of CFSMs $M_1, \ldots, M_n$ can be converted into an equivalent (single) FSM called the *product machine*. The product machine shall be described in Section 3. Tests can be generated from the product machine using standard FSM test techniques ([13]). Suppose n_i denotes the number of states of M_i. Then the number of states of the product machine is of $O(\Pi_i n_i)$ and thus this approach may suffer from a combinatorial explosion. This paper explores approaches that avoids this combinatorial explosion.

When a model M, comprising of CFSMs, receives an input this input triggers a sequence of *local transitions* within the individual CFSMs, forming a *global transition* in M. The number of global transitions is exponential in terms of the number of CFSMs while the number of local transitions is simply the sum of the number of transitions of the CFSMs. However, each global transition consists of a number of local transitions and, if the local transitions are correct, the global transitions are also correct. Thus, where feasible, it makes sense to test the local transitions, rather than the individual global transitions. While there are generally far fewer local transitions that global transitions, faults in local transitions may be masked. Section 4 considers the problem of generating a set of tests, for a local transition, that limits the opportunity for fault masking. Given a local transition t, this involves testing a (polynomial) number of global transitions that contain t.

In order to test a global transition t it is necessary to follow t by further transitions that check the final state of t. This paper introduces a new approach to checking the state of a set of CFSMs. This approach is based on checking each local state separately using sequences with associated constraints. A constraint defines when a sequence may be used to check a state. The approach, of using sequences with constraints, will be described in Section 5.

The paper then focuses upon one particular type of constrained state identification sequence, called a *constrained identification sequence (CIS)*. Section 6 shall show that CISs may be used to check the global state of a set of CFSMs. Section 7 then considers sequencing CISs in order to reduce test effort. Finally, in Section 8, conclusions are drawn.

2 Preliminaries

2.1 Finite state machines

A (deterministic) *finite state machine* (*FSM*) M is defined by a tuple $(S, s_1, \delta, \lambda, X, Y)$ in which S is a finite set of *states*, $s_1 \in S$ is the *initial state*, δ is the *state transfer function*, λ is the *output function*, X is the finite *input alphabet*, and Y is the finite *output alphabet*. If M receives input x while in state s it produces output $y = \lambda(s, x)$ and moves to state $s' = \delta(s, x)$. This defines a *transition* $(s, s', x/y)$. The functions δ and λ can be extended, to take input sequences, to give δ^* and λ^* respectively.

The FSM M is *completely specified* if δ and λ are total. Given an FSM M that is not completely specified it is possible to complete M by either adding an error state or assuming that, where the response to an input is not given, this input leads to no change in state and null output.

FSM M has *reset capacity* if there is some input $r \in X$ whose associated transitions all go to s_1: $\forall s \in S.\delta(s, r) = s_1$. Often there is some reset operation, called a *reliable reset*, that is known to have been implemented correctly. This allows test sequences to be separated by resets and sometimes is simply implemented through the system being switched off and then on again. Throughout this paper it shall be assumed than any implementation considered has a reliable reset operation.

An FSM M is *initially connected* if every state of M can be reached from the initial state of M. If M is not initially connected the unreachable states may be removed. M is *strongly connected* if, for every ordered pair of states (s_i, s_j) from M, there is some input sequence that takes M from s_i to s_j. Clearly if M is initially connected and has a reset operation then M is strongly connected. It will be assumed that any FSM (or CFSM) considered is strongly connected.

An input value x *distinguishes* states s_i and s_j if $\lambda^*(s_i, x) \neq \lambda^*(s_j, x)$. Two states s_i and s_j are said to be *distinguishable* if there is some input value that distinguishes them. This shall be denoted $s_i \not\sim s_j$. If s_i and s_j are not distinguishable they are said to be *equivalent* and this is denoted $s_i \sim s_j$. Two FSMs are *equivalent* if their initial states are equivalent. $M \sim M'$ shall represent M and M' being equivalent and otherwise $M \not\sim M'$.

An FSM M is *minimal* if there is no FSM M' that is equivalent to M and has fewer states that M. It is known that M is minimal if and only if it is initially connected and every pair of states from M is distinguishable. There are algorithms that take an FSM and return an equivalent minimal FSM ([14]) and thus only minimal FSMs will be considered. For more on FSMs see, for example, [11].

Many approaches, that generate tests from an FSM, produce tests for individual transitions. There are two types of faults for a transition: *output faults*, in which the wrong output is produced, and *state transfer faults* in which the state after the transition is wrong. In order to detect state transfer faults it is necessary to follow a transition by further input values that check the state after the transition. One common approach to state checking is the use of a *unique input/output sequence (UIO)*. Suppose input sequence $u(s)$ leads to output o when executed from state s of M. Then $u(s)/o$ is a UIO for state s of M if, for every state s', if $s' \neq s$ then $u(s)$ distinguishes between s and s'.

2.2 Communicating finite state machines

While many systems may be modelled as FSMs, it is sometimes more natural and efficient to model a system as a set of entities, each of which behaves like an FSM, that may interact. These entities are called communicating finite state machines and essentially each is an FSM with an input queue. Since this paper is concerned with the testing of the transition structure only, a CFSM will be defined in the same way as an FSM: the queues will not be explicitly mentioned in the definition.

Throughout this paper M shall be a model defined by CFSMs $M_1, \ldots, M_n$. Here M_i denotes a CFSM $(S_i, s_{i1}, \lambda_i, \delta_i, X_i, Y_i)$, with an implicit input queue, in which $S_i = \{s_{i1}, s_{i2}, \ldots, s_{im_i}\}$. The transitions of a CFSM M_i shall be called *local transitions*. If M receives a value a, from either the environment or another CFSM, contained in the input alphabet of M_i but no other CFSM, a enters the input queue of M_i. If a is contained in the input alphabet of more than one M_i, this value non-deterministically enters the input queue of exactly one M_i with $a \in X_i$. Since one CFSM can receive input from another, it is possible for the output from a local transition to trigger a local transition in some other machine. It is worth noting that, if a message is passed between two CFSMs, this message is not observed by the environment. A CFSM cannot output a value from its input alphabet and thus $\forall 1 \leq i \leq n . X_i \cap Y_i = \emptyset$.

Some input values may be hidden from the environment: they can only be received from CFSMs within M. Let $H \subseteq \cup_i X_i$ denote the set of hidden input values for M. Then M, which is $M_1 \mid \ldots \mid M_n$ with H hidden, shall be written $(M_1 \mid \ldots \mid M_n) \backslash H$.

This paper shall only consider deterministic systems. Thus each M_i is deterministic and the input alphabets of the M_i are disjoint. A CFSM is *minimal* if the corresponding FSM is minimal. It shall be assumed that each M_i is minimal and completely specified and that M is deadlock and livelock free.

At any moment M is in a *global state*, which is defined by the states of the M_i and the contents of the queues. In this paper Σ shall denote the set of global states of M. A global state is *stable* if all of the queues are empty and otherwise it is *unstable*. The state of an individual CFSM M_i shall be called a *local state*. In testing the transition structure it is normally sufficient to test in stable states.

The input of some value a to M, while M is in stable global state $\sigma \in \Sigma$, will lead to a sequence of local transitions being executed. The final local transition will output some value x to the environment and leave M in stable global state σ'. Then $(\sigma, \sigma', a/x)$ is a *global transition* of M.

Consider, for example, the CFSMs N_1, N_2 and N_3 shown in Figure 1. This collection of CFSMs shall be denoted M_0 throughout the paper and $M_0 = N_1 \mid N_2 \mid N_3$. Suppose each CFSM is in its initial state and has empty input queue. Then if M_0 receives input f, f enters the input queue of N_3. This triggers a local transition, in N_3, that outputs a and moves N_3 to state s_{32}. The value a is in the input alphabet of N_1 and thus enters its input queue. N_1 reacts to the value a by executing a local transition that moves N_1 to state s_{12} and outputs the value x. As x is not in the input domain of any of the CFSMs it is sent to the environment as output. Thus the input of a triggered the global transition $((s_{11}, s_{21}, s_{31}), (s_{12}, s_{21}, s_{32}), a/x)$.

Throughout this paper a local transition shall be called an *internal transition* if its output is contained within some X_j and otherwise it shall be called an *external transition*. In [6] internal transitions are called communicating transitions and external transitions are called non-communicating transitions. The next section shall further discuss the role of the input queues and define the product machine.

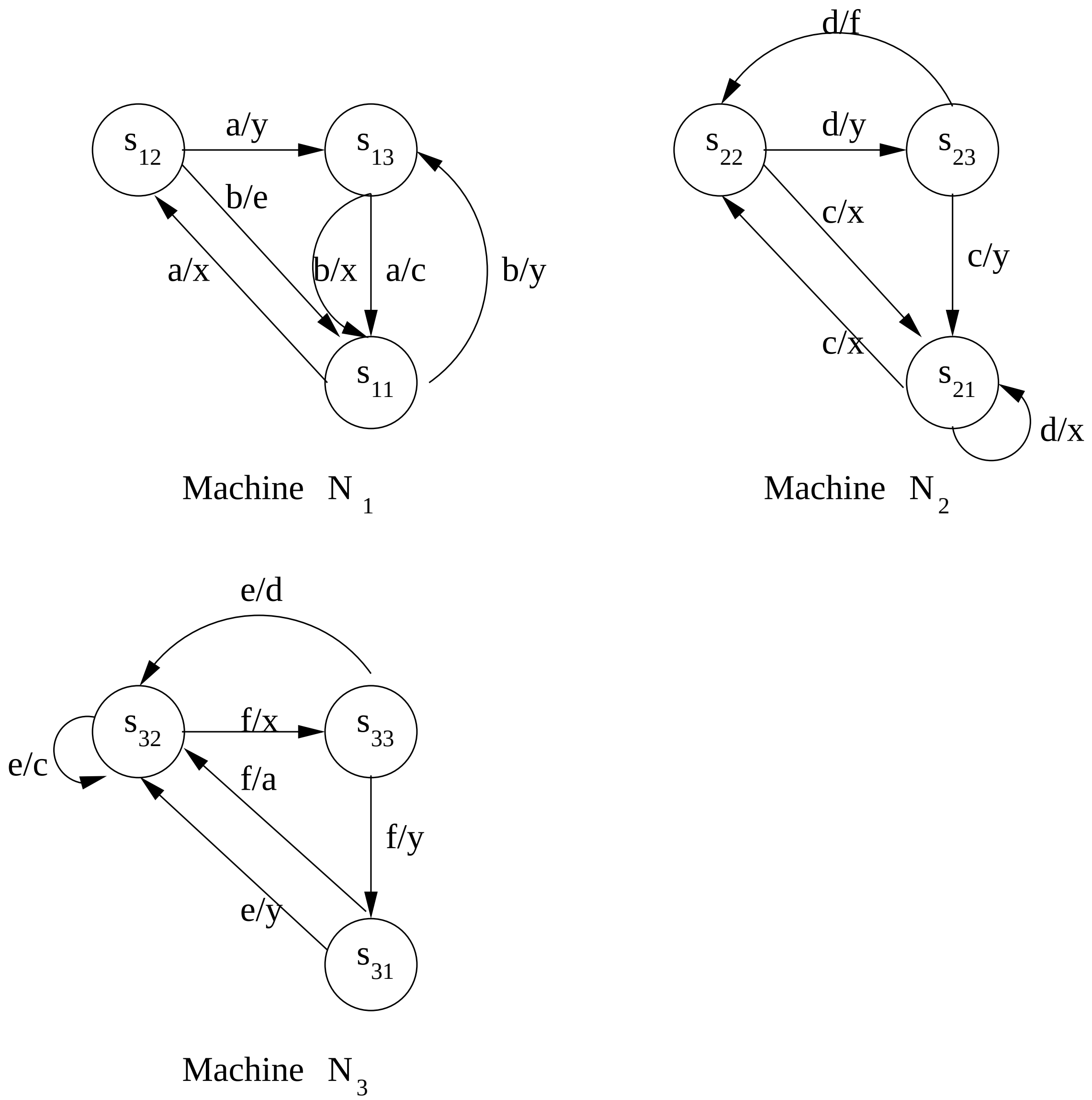

Figure 1: The model M_0

3 The Product Machine

The existence of unbounded queues may lead to M having an infinite number of states. M will, however, have a finite number of stable states. Since M is livelock and deadlock free, when only stable states are considered M is equivalent to an FSM called the *product machine*.

Under some conditions, input values will only be received in stable states and thus the full behaviour of M is equivalent to the product machine ([13]). Even where there are queues, it is normally possible to test the transition structure of a system while only inputting values in stable states. Naturally, where there are queues in the IUT this testing should be supplemented by further tests that check the queues. This paper deals with the problem of generating tests for the transition structure and thus shall only consider testing transitions from stable states.

Let $P(M)$ denote the product machine generated from M, X and Y denote the input and output alphabets of M, and $\Sigma_s = S_1 \times \ldots \times S_n$ denote the set of stable global states. Clearly some elements of Σ_s may be unreachable. Then $X = \bigcup_i X_i \setminus H$ and $Y = \bigcup_i Y_i \setminus \bigcup_i X_i$. The input and output alphabets of $P(M)$ are X and Y respectively and its state set is Σ_s^r: the set of reachable states from Σ_s. The initial state of $P(M)$ is $s_1^P = (s_{11}, s_{21}, \ldots, s_{n1})$.

The next state and output functions, δ and λ, of $P(M)$ shall now be defined. These give the behaviour exhibited when M receives an input value when in a stable state. These functions define the transitions of the product machine and thus $P(M)$ is defined by $(\Sigma_s^r, s_1^P, X, Y, \delta, \lambda)$.

A tuple $p = (p_1, \ldots, p_m)$ ($\forall 1 \leq i \leq m. p_i \in D$ for set D) may be seen as a function from $\{1, \ldots, m\}$ to D and thus, if $1 \leq k \leq m$, $p(k)$ denotes p_k. Let $\oplus$ denote functional overwriting. Thus $p \oplus \{i \rightarrow p_i'\}$ is p with its ith component changed to p_i'. Given $x \in X_k$ and stable global state $\sigma \in \Sigma_s$, next state and output functions δ and λ for the product machine are defined by the following.

If $\lambda_k(\sigma(k), x) \notin \bigcup_i X_i$

$$\delta(\sigma, x) = \sigma \oplus \{k \rightarrow \delta_k(\sigma(k), x)\}$$

$$\lambda(\sigma, x) = \lambda_k(\sigma(k), x)$$

If $\lambda_k(\sigma(k), x) \in \bigcup_i X_i$

$$\delta(\sigma, x) = \delta(\sigma \oplus \{k \rightarrow \delta_k(\sigma(k), x)\}, \lambda_k(\sigma(k), x))$$

$$\lambda(\sigma, x) = \lambda(\sigma \oplus \{k \rightarrow \delta_k(\sigma(k), x)\}, \lambda_k(\sigma(k), x))$$

The first case defines the behaviour when the input triggers an external transition and thus the output of the local transition forms the output of the global transition. The second case defines the behaviour when an internal transition t is triggered: this is simply the behaviour produced if t is executed and then the output from t is fed back into M as input. Since M is livelock and deadlock free, δ and λ are total functions. Since these functions are defined for hidden values it is necessary to restrict their input domains to $\Sigma_s^r \times (\bigcup_i X_i \setminus H)$.

It is worth noting that it is possible to determine the action of δ and λ on certain values without stating the complete behaviour associated with δ and λ. This is important since generating δ and λ is equivalent to generating the product machine. Later δ and λ shall be used to define Sequences for checking local and global states but it shall not be necessary to produce the full behavior of δ and λ.

4 Testing Global and Local Transitions

This section shall consider the problem of testing the transitions of M. The section will start by discussing the problem of testing global transitions and shall then consider the problem of testing local transitions.

Given a stable global state σ, a global transition $(\sigma, \sigma', x/y)$ of M may be tested in the same way as a transition from an FSM. The test starts with a sequence that moves M to state σ, then x is input and finally sequences that check state σ' are executed. While the problem of generating an input sequence to reach some global state is non-trivial, there are a number of conditions under which it has been solved ([15]). The problems, of finding input sequences to reach global or local states, and of determining which local and global states are reachable, are outside the scope of this paper. Sections 6 and 7 shall describe approaches that may be used to check the global state of a set of CFSMs.

Instead of testing the global transitions the tester might check the individual local transitions. This approach has the advantage that there are $O(\sum_{i=1}^{n} |X_i| m_i)$ local transitions but $O(|X| \prod_{i=1}^{n} m_i)$ global transitions. Thus, by testing local transitions it may be possible to avoid the combinatorial explosion associated with the generation of the global transitions. Testing a local transition does, however, raise new issues.

Suppose the problem is to test some local transition $t = (s, s', a/x)$ from M_i. Whenever local transition t is tested it forms part of some global transition, although this global transition might just contain t. If t is tested as part of a global transition T, the test can only check the output and final state of T, not of t. While the output and final state of T might provide some information about t, a fault in t might be masked by the local transitions that follow it. A fault in t, that is masked when it forms part of T, might lead to faulty behaviour if executed, by the user, as part of a different global transition.

Fault masking, and an approach that limits the opportunity for fault masking, shall now be described in further detail.

4.1 Fault Masking

This section shall describe ways in which faults may be masked, showing that both output and state transition faults may be masked. The following section shall describe an approach that limits the opportunity for fault masking.

Consider the local transition $t = (s_{11}, s_{12}, a/x)$ from the CFSM N_1 in M_0 and the output fault that makes the transition become $(s_{11}, s_{12}, a/d)$. Suppose also that t is to be tested by entering a when M_0 is in the initial state (s_{11}, s_{21}, s_{31}). Then rather than produce output x, the fault leads to t producing output d. The value d is not observed directly by the environment: instead it is passed to N_2. The value d triggers the local transition $(s_{21}, s_{21}, d/x)$ in N_2 and the value x is sent to the environment. Thus the output is x and the final state of this global transition is (s_{12}, s_{21}, s_{31}) as expected. Thus, it is not possible to detect this fault by executing t from the initial global state.

The output fault described above was masked by a *loop*: a local transition with the same initial and final states. Another way of masking an output fault is through the existence of *parallel transitions*: two local transitions with the same initial and final states. Parallel transitions with the same output may mask output faults. Suppose, for example, that in the configuration used to test some local transition t, t is expected to trigger a local transition $t' = (s_p, s_q, x/y)$ that is parallel to some other local transition $t'' = (s_p, s_q, x'/y)$. If t produces erroneous output x' it triggers t'' rather than t' but this leads to the expected output and final global state. This test fails to detect the output fault in t. Naturally, loops and parallel transitions are not the only ways of masking output faults.

State transfer faults may also be masked, though to mask a state transfer fault in some local transition t of M_i it is necessary for M to be in a global state that leads to one or more further transitions being executed in M_i. A local transition t from M_i *feedbacks* in a global transition T if T contains at least one local transition from M_i after t. If the execution of t as part of T feedbacks an incorrect final state for t might be masked by further local transitions in M_i.

4.2 Avoiding fault masking

While both output faults and state transfer faults in a local transition may be masked, faults in global transitions are not masked. However, there will often be many more global transitions than local transitions and thus a fault in a local transition may appear in many global transitions. Interestingly, this observation is consistent with empirical evidence that suggests that most faults are observed from many global states ([21]).

When testing local transition t it is sufficient to find a set of global transitions that contain t and that, when used in testing, allow any fault in t to be detected. Once such a set has been found, these can be tested. It is worth noting that there may be some changes to t that are not capable of affecting any global transition and thus do not introduce a fault. Clearly testing need not, and indeed cannot, detect such changes.

When testing a local transition t, it shall be assumed that all other transitions executed are correct. It is thus assumed that faults in other local transitions cannot mask a fault in t. While it might be argued that, in practice, it is unlikely that multiple faults will mask one another, there will be cases where this assumption is not acceptable. It would thus be interesting to develop alternative strategies that guarantee the detection of combinations of faults.

Suppose $t = (s, s', a/x)$ is a local transition from M_i. Let $F(t)$ denote the set of alternative versions of t: versions that have an output fault, a state transfer fault or both. Thus $F(t) = \{(s, s'', a/y) | s'' \in S_i \wedge y \in Y_i \wedge (s, s'', a/y) \neq (s, s', a/x)\}$. Further let $F^{NE}(t)$ denote the largest subset of $F(t)$ such that if t is replaced by any element t' of $F^{NE}(t)$ in M to get $M[t'/t]$ then $M[t'/t] \not\sim M$. Thus, $F^{NE}(t)$ is the set of mutants of t that lead to a change in the behaviour of M and thus lead to a fault in M. Clearly $|F^{NE}(t)| \leq |S_i||Y_i| - 1$.

It is possible to consider the effect of replacing t by some mutant $t' \in F^{NE}(t)$ in a global transition $T = (\sigma, \sigma', \alpha/\beta)$ that contains t. Suppose the input of α in global state σ of $M[t'/t]$ leads to final state σ'' and output β'. Then T *kills* t', by distinguishing M and $M[t'/t]$, if and only if either $\sigma' \neq \sigma''$ or $\beta \neq \beta'$. This shall be denoted $kills(T, t')$. Then it is sufficient to test some set $\mathcal{T}$, of global transitions, that satisfy the following property.

$$\forall t' \in F^{NE}(t).\exists T \in \mathcal{T}.kills(T, t')$$

Such a set of global transitions might be produced by a breadth-first search through global transitions that contain t. The search starts with t, if this forms a global transition. The search then considers global transitions that contain 2 local transitions (either t followed by another local transition or t preceded by another local transition). At each step the number of local transitions contained in global transitions considered increases by 1 until a sufficient set is found. It is worth noting that $|F^{NE}(t)|$ is polynomial in terms of the size of the M_i and the number of local transitions is also polynomial in terms of the size of the M_i. This contrasts with the number of global transitions which may be exponential in terms of the size of the M_i.

In general, there may be no efficient way of deciding whether $M[t'/t] \not\sim M$. In such cases it may not be practical to generate $F^{NE}(t)$ and instead $F(t)$ might be used, the search terminating either when all mutants are killed or when some pragmatic limit (on the number of local transitions contained in a global transition) has been reached.

Consider now the local transition $t_1 = (s_{11}, s_{12}, a/x)$ of N_1. Then the input of a, when M_0 is in its initial state, will detect a fault in t_1 unless the fault is an output fault giving output d. To detect a fault consisting of the output fault d it is sufficient to input a in state (s_{11}, s_{22}, s_{31}). Thus, these two tests, combined with approaches that check the final state of each test, suffice to check the local transition t_1.

5 Checking local states

This section shall consider the problem of checking the state of a CFSM M_i, within M, without generating the product machine. Since a global state consists of a number of local states, it seems natural to use techniques developed for checking local states to check global states. The problem of checking a global state shall be considered in Sections 6 and 7.

It is possible to extend the notion of a UIO for a single FSM to an input sequence $u(s) \in X^*$ that is capable of checking the local state s of M_i within M. Then $u(s)$ is a *local state identifying sequence (LSIS)* for local state s of M_i if the following holds.

$$\{\lambda^*(\sigma, u(s))|\sigma(i) = s\} \cap \{\lambda^*(\sigma, u(s))|\sigma(i) \neq s\} = \emptyset.$$

LSISs are simply called UIOs in [6]. While the presence of LSISs simplifies the problem of state checking, it is likely that many CFSMs will not have LSISs for each state. This section shall consider a number of alternatives.

Suppose s is a local state of M_i. Then for a global state σ, with $\sigma(i) = s$, there may exist some input sequence u that distinguishes σ from all other global states in which only the local state of M_i differs. These are the global states of the form $\sigma \oplus \{i \rightarrow s'\}$ for $s' \in S_i \setminus \{s\}$. Then if the state of each $M_j \neq M_i$ is known to be consistent with σ, u might be used to check the state of M_i. If u is used to check the local state of M_i it depends upon the states of the other CFSMs being those in σ and thus u has associated with it the constraint that takes a global state σ' and returns *true* if and only if $\forall 1 \leq j \leq n . i \neq j \Rightarrow \sigma(j) = \sigma'(j)$.

In general, there may be an input sequence u that is capable of checking local state s of M_i for some set of states of the other M_j. This set of states defines a constraint c. If input sequence u has constraint c then u may be used whenever the global state satisfies c.

Consider the example, M_0, given in Figure 1. The only transition that has input a and output y is $(s_{12}, s_{13}, a/y)$. This suggests the use of a to check local state s_{12}. If, however, N_1 is in state s_{13} then, in response to a, it outputs c which triggers a local transition in N_2. This local transition, in N_2, outputs y if and only if the state of N_2 is s_{23}. Thus, a checks that N_1 is in state s_{12} if and only if N_2 is not in state s_{23}.

The following defines the notion of a *constrained identification sequence (CIS)*.

Definition 1 *The tuple (u, c), in which $u \in X^*$ and c is a constraint on the global states of M, is a* constrained identification sequence *for state s of M_i if and only if $\{\lambda^*(\sigma, u) | c(\sigma) \wedge \sigma(i) = s\} \cap \{\lambda^*(\sigma, u) | c(\sigma) \wedge \sigma(i) \neq s\} = \emptyset$.*

A set U_s is said to be a *complete set* of CISs for local state s of M_i if, for every global state σ with $\sigma(i) = s$, there is some $(u, c) \in U_s$ such that σ satisfies c. Let U denote a mapping that takes a local state s and returns a set of CISs $U(s)$. If every local state of M has a known complete set of CISs given by U then U gives a *complete set* of CISs.

There need not be a complete set of CISs. Instead, it is possible to weaken this notion to allow the use of a set of sequences for state checking. The following defines the notion of a *constrained state identification set (CSIS)*.

Definition 2 *The tuple (w, c), in which $w = \{w_1, \ldots, w_k\}$, $w_i \in X^*$, and c is a constraint on the global state of M, is a* constrained state identification set *for state s of M_i if and only if $\{(\lambda^*(\sigma, w_1), \ldots, \lambda^*(\sigma, w_k)) | c(\sigma) \wedge \sigma(i) = s\} \cap \{(\lambda^*(\sigma, w_1), \ldots, \lambda^*(\sigma, w_k)) | c(\sigma) \wedge \sigma(i) \neq s\} = \emptyset$.*

This paper shall concentrate on the use of CISs. It is possible to see the existence of a (predefined) relatively small set of CISs as a design for test condition.

The following result, which states that where there is a complete set of LSISs these provide a complete set of CISs, follows immediately from the definitions.

Lemma 1 *Suppose M has an LSIS for each local state. Then this set of LSISs forms a complete set of CISs for M.*

The use of CISs thus generalizes the notion of an LSIS. The use of CISs may also reduce the test effort where CISs are shorter than LSISs. The potential for reducing test effort can be seen in M_0: s_{11} has a LSIS aa but, when the state of M_2 is s_{23}, a will suffice.

Every state of M_0 has a complete set of CISs. Such a set, with constraints, is given in Figure 2 in which σ denotes the global state. In this case the output, corresponding to a CIS, is unique and so this is given.

While it is not the focus of this paper, it is worth briefly considering the problem of generating CISs. As noted earlier, the existence of a set of predefined CISs might be seen as a design for test condition.

State	in/out	Constraint
s_{11}	a/x	$\sigma(2) = s_{23}$
s_{11}	$a/x, a/y$	
s_{12}	a/y	$\sigma(2) \neq s_{23}$
s_{12}	$a/y, b/x$	
s_{13}	b/x	$\sigma(3) = s_{31} \vee (\sigma(3) = s_{32} \wedge \sigma(2) = s_{23}) \vee (\sigma(3) = s_{33} \wedge \sigma(2) = s_{22})$
s_{13}	$a/x, a/x$	$\sigma(2) = s_{23}$
s_{13}	$a/y, a/x$	$\sigma(2) \neq s_{23}$
s_{21}	$c/x, d/y$	
s_{22}	$c/x, d/x$	
s_{23}	c/y	
s_{31}	e/y	$\sigma(2) = s_{21}$
s_{31}	$f/x, f/x$	$\sigma(1) = s_{11} \vee (\sigma(1) = s_{13} \wedge \sigma(2) \neq s_{23})$
s_{31}	$f/y, f/x$	$\sigma(1) = s_{12} \vee (\sigma(1) = s_{13} \wedge \sigma(2) = s_{23})$
s_{32}	f/x	$\sigma(1) = s_{12} \vee (\sigma(1) = s_{13} \wedge \sigma(2) = s_{23})$
s_{32}	$f/x, f/y$	
s_{33}	f/y	$\sigma(1) = s_{11} \vee (\sigma(1) = s_{13} \wedge \sigma(2) \neq s_{23})$
s_{33}	$f/y, f/y$	$\sigma(1) = s_{12} \vee (\sigma(1) = s_{13} \wedge \sigma(2) = s_{23})$

Figure 2: CISs for M_0

For each local state a breadth first search through the set of input sequences might be applied. For each sequence considered, a corresponding constraint is developed. This continues until a set of sequences, whose constraints cover all possibilities, has been found.

In order to follow such a search, given an input sequence, it is necessary to produce a constraint for this input sequence. The generation of the weakest constraint may not be feasible since this may effectively require the exploration of significant sections of the product machine. Instead, it is possible to apply some heuristic that leads to a relatively simple, conservative, constraint. Heuristics might, for example, be based on limiting the complexity of the constraint or on limiting the number of internal transitions that may be considered when deriving the constraint.

Consider, for example, the local state s_{11} of N_1. Then the input of a leads to the output of x. This distinguishes s_{11} from any global state in which N_1 is in state s_{12}. If, however, N_1 were in state s_{13}, the input of a would lead to an internal transition and thus the output depends upon the state of the other CFSMs. This internal transition triggers an external transition in N_2 and leads to output other than x if N_2 is in state s_{23}. Thus a forms the input of a CIS for local state s_{11} and this CIS has constraint $\sigma(2) = s_{23}$ (where σ is the global state).

Suppose now that input b is considered. This is expected to produce output y and, again, is capable of triggering an internal transition, in this case producing output e. The value e is received by N_3 and either outputs y (if in state s_{31}) or triggers another internal transition. Thus, input b does not suffice if N_3 is in state s_{31}. If N_3 is in state s_{32} the input of e leads to c being sent to N_2. This leads to output x, and thus suffices, if N_2 is not in state s_{23}. If N_3 is in state

s_{33}, the input of e leads to d being sent to N_2 and this then leads to output x (possibly after another transition in N_3 being triggered) if N_2 is not in state s_{22}. Thus, b has constraint $\sigma(3) = s_{32} \wedge \sigma(2) \neq s_{23} \vee \sigma(3) = s_{33} \wedge \sigma(2) \neq s_{22}$.

Once aa is considered, it is straightforward to confirm that this always suffices and thus has constraint *true*. Thus the sequences a, b and aa suffice. Where simple constraints are considered to be desirable, just a and aa might be used.

6 Checking global states using CISs

The notion of a CIS, which may be used to check a local state, has been introduced. The problem now is to use CISs to check global states. It shall be assumed that the tester wishes to check that the global state, after some test sequence τ, is $\sigma = (s_1, \ldots, s_n)$. Naturally, the approach outlined in this and the following section may be applied when checking some part of the global state.

One natural approach is to choose a CIS for each local state and, for each of these, separately execute τ followed by the CIS. However, a set of CISs $U_0 = \{(u_1, c_1), \ldots, (u_n, c_n)\}$, that is consistent with σ, might not be sufficient. Suppose that there is some CIS (u_i, c_i) that mentions the state of M_j. If the state of M_j is not that expected, u_i need not be able to check the state of M_i. There might be a number of such dependencies and, potentially, circularities in the set of dependencies. This happens if, for example, u_i depends on the state of M_j being from some subset of S_j and u_j depends on the state of M_i being from some subset of S_i. Here the process used for checking the correctness of the state of M_i assumes that the state of M_j is correct but in checking the state of M_j it is assumed that the state of M_i is correct. Thus, these CISs may not detect both states being incorrect. This situation, in which there is a cycle of dependencies, shall be called *dependency circularity*.

The dependencies in U may be represented by a directed graph $G_D = (V_D, E_D)$ in which the vertex set V_D is $\{d_1, \ldots, d_n\}$ and each d_i represents the corresponding M_i. Then there is an edge from d_i to d_j if and only if the state of M_j is mentioned in the constraint c_i. This directed graph G_D shall be called the *dependency digraph*. Then there is dependency circularity if and only if the dependency digraph contains one or more cycles.

Where a constraint c_i is not simply a conjunction of atomic predicates it is possible to generate more than one dependency digraph. Then c_i might be rewritten to disjunctive normal form (DNF) and each conjunct, formed by this process, considered separately. Suppose, for example, that rewriting c_i to DNF gives $c_i^1 \vee \ldots \vee c_i^{p_i}$. Then it is sufficient for *one* conjunct c_i^j, such that $c_i^j(\sigma)$ evaluates to *true*, to lead to a dependency digraph that is cycle free. Clearly, it is not necessary to distinguish between two conjuncts that mention the same machines.

The problem of choosing an appropriate set of CISs is thus that of finding a set $U = \{(u_1, c_1), \ldots, (u_n, c_n)\}$, $(u_i, c_i) \in U(s_i)$, in which

1. $\forall 1 \leq i \leq n . c_i(\sigma)$;
2. there is a corresponding cycle free dependency digraph.

Consider, for example, the global state $\sigma_1 = (s_{11}, s_{23}, s_{31})$ of M_0. Here it is possible to use a to check state s_{11}, c to check s_{23} and ff to check s_{31}. The

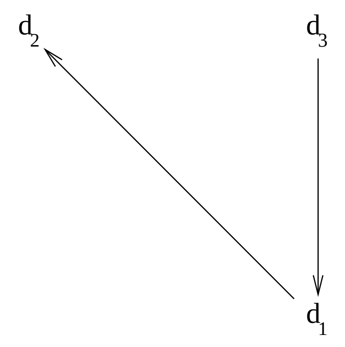

Figure 3: A dependency digraph

dependency digraph G_D^0 includes an edge from d_1 to d_2 since the constraint for the CIS used for s_{11} mentions the state of N_2. The constraint of the CIS used for s_{31} is not in DNF but the conjunct satisfied is $\sigma(1) = s_{11}$. Thus G_D^0, which is shown in Figure 3, contains an edge from d_3 to d_1. Clearly G_D^0 is cycle free and thus these CISs suffice.

Suppose a set $U_0 = \{(u_1, c_1), \ldots, (u_n, c_n)\}$ of CISs, with cycle free dependency digraph, has been found for the final state of τ. It is possible to check the final state of τ by following it by each of the u_i, separating these tests by resets. Thus, for example, in order to check that the final states of $c/x, d/y$ is (s_{11}, s_{23}, s_{31}) it is sufficient to use the test sequences:

- $c/x, d/y, a/x$
- $c/x, d/y, c/y$
- $c/x, d/y, f/x, f/x$.

In some cases it may be possible to sequence a number of the u_i before a reset, reducing the number of times τ must be executed and thus the cost of testing. This will be discussed in the next section.

7 Sequencing CISs

Suppose CIS set $U_0 = \{(u_1, c_1,), \ldots, (u_n, c_n)\}$ is to be used for checking the final state of τ, which is expected to be σ. The test effort may sometimes be reduced by following τ by a sequence composed of either all of the u_i or some subset of this. This reduces the number of times τ is executed during testing.

The edges of the dependency digraph impose an ordering on the CISs: if there is an edge from d_i to d_j then u_i depends upon the state of M_j and thus, if these CISs are being sequenced, u_i should be applied before u_j (since u_j will change the state of M_j). The rest of this section shall consider other properties required of the set of CISs used and ways in which an appropriate set of CISs may be chosen.

A CIS can impose a further ordering: if the state of M_i is affected by the transitions included in u_j then, if they are to be sequenced, u_i must be applied before u_j. The ordering imposed by the CISs can be represented by a directed graph $G_O = (V_O, E_O)$ called the *order digraph*. Then $V_O = \{o_1, \ldots, o_n\}$, in which o_i represents u_i. There is an edge from o_i to o_j in E_O if and only if one or more of the following hold.

1. u_i depends on the state of M_j (and thus there is a corresponding edge in the dependency digraph).

2. the input of u_j is expected to trigger one or more local transitions of M_i.

Suppose the order digraph contains an edge from o_i to o_j. Then, if the CISs are to be sequenced, o_i should precede o_j. The order digraph may contain one or more cycles, in which case the full set of CISs should not be sequenced. This type of situation, in which there is no valid order of execution for the full set of CISs, shall be called *order circularity*. Where the order digraph is cycle free it defines the allowed orders, of the CISs, in the natural way.

As before, where a constraint can be rewritten as a disjunction of constraints, these may be considered separately. This leads to a set of order digraphs and it is sufficient for one of these to be cycle free.

Consider, again, the global state $\sigma_1 = (s_{11}, s_{23}, s_{31})$ of M_0 and the CISs a for s_{11}, c for s_{23} and ff for s_{31}. Since none of these CISs affects another CFSM in M_0, the order digraph is the same as the dependency digraph. Thus, the CISs should be sequenced in the order: ff then a and finally c. Thus, in order to check the final state of $c/x, d/y$ it is sufficient to execute the test sequence $cdffac$ (with expected output $xyxxxy$).

The problem of producing a set of CISs, that may be sequenced, becomes one of finding a set of CISs, one for each M_i $(1 \leq i \leq n)$, that is consistent with the state σ and has a cycle free order digraph.

Where the order digraph contains one or more cycles it may still be possible to sequence some of the CISs. Suppose J is a subset of $\{1, \ldots, n\}$ and $U_J = \{u_i | i \in J\}$. Then the CISs in U_J may be sequenced if the ordering between these contains no cycles. This is the case if G_O contains no cycle whose vertices all have indices drawn from J. To be more precise, the CISs in U_J may be sequenced if and only if the directed graph $(V_0, \{e = (o_i, o_j) | e \in E_O \wedge o_i \in J \wedge o_j \in J\})$ is cycle free. Thus, given an order digraph that contains one or more cycles, it is sufficient to partition the set of CISs in a manner that leads to a set of cycle free digraphs.

Suppose, for example, that there are four CFSMs $M_1, \ldots M_4$, CISs with input sequences $u_1, \ldots, u_4$ have been chosen and the order digraph is that shown in Figure 4. Since this order digraph contains a cycle these CISs cannot be sequenced. However, there are a number of ways in which two sequences of CISs may be used. Possible pairs of sequences include, for example, (u_1u_2, u_3u_4), $(u_3u_1u_4, u_2)$ and (u_2u_3, u_1u_4).

A further refinement would be to allow the input values from the CISs to be interleaved. While this may sometimes help, it complicates the problem of finding a valid set of CISs. This paper shall not consider interleaving these inputs.

$$\begin{tikzcd}
o_1 \arrow[r] \arrow[d] & o_2 \arrow[d] \\
o_4 & o_3 \arrow[ul]
\end{tikzcd}$$

Figure 4: An order digraph

7.1 Finding sets of CISs that may be sequenced

Suppose the tester wishes to sequence CISs where possible. Then, given a global state σ to check, the problem might be: find the minimal length $u_1, \ldots, u_n$, $(u_j, c_j) \in U(s_j)$, such that σ satisfies each constraint and the order digraph is cycle free. Alternatively, if the sequence τ, whose final state is being checked, is known then the total cost may be found. If some CISs may be sequenced, to form m sequences, the test length for τ (including resets) is

$$\sum_{i=1}^{n} |u_i| + m(|\tau| + 1)$$

The problem then is: find a set of CISs, and a way of sequencing these, in order to minimize this cost. Where there are few alternative CISs for each local state, this optimization problem may be quite small. Once this problem has been solved, the order digraph and partition define the allowed orders of application.

Where an exhaustive search is infeasible, heuristics may be applied. One possibility is to apply a greedy algorithm, starting with the lowest cost CIS. At each stage the CIS that leads to the minimum extra cost is chosen, and the corresponding edges added.

An alternative is to apply some meta-heuristic, such as Tabu search or a Genetic Algorithm. In order to apply such search techniques it is necessary to define an objective function that says how good a candidate solution is. This objective function might just be the cost function given above.

8 Conclusions

When testing, against a model consisting of communicating finite state machines, it is normal to test individual transitions. It has been shown that, rather than test the global transitions, it is possible to check the local transitions. This may significantly reduce the test effort. Faults in local transitions may, however, be masked. An approach, that limits the opportunity for fault masking, has been given.

When testing a transition it is necessary to check the global state after the transition. While it is possible to use the product machine as the basis for test generation, the process of generating a product machine may suffer from a combinatorial explosion. An alternative approach, of using constrained state identification sets, has been introduced. This approach checks local states, the process of checking a global state being seen as that of checking each individual local state. A special type of state identification set, the constrained identification sequence (CIS), has been considered in further detail.

When checking that the final global state, after input sequence τ, is σ it is necessary to find a set of CISs whose constraints are consistent with σ. The constraints may, however, define certain dependencies between the CISs and the set of CISs might generate a circuit of dependencies. The dependency digraph has been defined and the problem of finding an appropriate set of CISs has been represented as that of finding a consistent set of CISs with a circuit free dependency digraph.

It is sometimes possible, when checking the final global state after input sequence τ, to follow τ by a sequence composed of CISs. Each CIS may impose an ordering on the CISs. The order digraph has been defined and the problem of finding an appropriate sequence of CISs has been represented as that of finding a set of CISs, that are consistent with σ, with circuit free order digraph. Where there are cycles in the order digraph, it is possible to partition the set of CISs in a manner that leads to a number of cycle free order digraphs. The CISs in each set of the partition may then be sequenced.

References

[1] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours. In *Protocol Specification, Testing, and Verification VIII*, pages 75–86, Atlantic City, 1988. Elsevier (North-Holland).

[2] J. Grabowski, R. Scheurer, D. Toggweiler, and D. Hogrefe. Dealing with the complexity of state space exploration algorithms for SDL systems. *University of Bern Technical Report*, IAM-95-010, 1995.

[3] M. P. E. Heimdahl, J. M. Thompson, and B. J. Czerny. Specification and analysis of intercomponent communication. *IEEE Computer*, 31:47–54, 1998.

[4] R. M. Hierons. Extending test sequence overlap by invertibility. *The Computer Journal*, 39:325–330, 1996.

[5] R. M. Hierons. Testing from a finite state machine: Extending invertibility to sequences. *The Computer Journal*, 40:220–230, 1997.

[6] R. M. Hierons. Testing from semi-independent communicating finite state machines with a slow environment. *IEE Proceedings on Software Engineering*, 144:291–295, 1997.

[7] R. M. Hierons. Adaptive testing of a deterministic implementation against a nondetermistic finite state machine. *The Computer Journal*, 41:349–355, 1998.

[8] K. Inan and H. Ural. Efficient checking sequences for testing finite state machines. *Information and Software Technology*, 41:799–812, 1999.

[9] F. Ipate and M. Holcombe. An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics*, 63:159–178, 1997.

[10] ITU-T. *Z.500 Framework on formal methods in conformance testing.* International Telecommunications Union, 1997.

[11] Z. Kohavi. *Switching and Finite State Automata Theory.* McGraw-Hill, New York, 1978.

[12] G. Luo, A. Das, and G. von Bochmann. Generating tests for control portion of SDL specifications. In *Protocol Test Systems VI*, pages 51–66. Elsevier (North-Holland), 1994.

[13] G. Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. *IEEE Transactions on Software Engineering*, 20:149–162, 1994.

[14] E. P. Moore. Gedanken-Experiments. In C. Shannon and J. McCarthy, editors, *Automata Studies.* Princeton University Press, 1956.

[15] K. Ozdemir and H. Ural. Protocol validation by simultaneous reachability analysis. *Computer Communications*, 20:772–788, 1997.

[16] M. Romdhani, P. Chambert, A. Jeffroy, P. de Chazelles, and A. A. Jerraya. Composing activity charts/statecharts, SDL and SAO specifications in codesign in avionics. In *European design automation conference with EUOR-VDHL '95*, pages 585–590, Brighton, England, September 1995.

[17] T. Savor and R. E. Seviora. Towards automatic detection of software failures. *IEEE Computer*, 31:68–74, 1998.

[18] K.-C. Tai and Y.-C. Young. Synchronizable test sequences of finite state machines. *Computer Networks and ISDN Systems*, 30:1111–1134, 1998.

[19] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 3 edition, 1996.

[20] C. Tropper and A. Boukerche. Parallel simulation of communicating finite state machines. In *The 1993 workshop on Parallel and Distributed Simulation*, pages 143–150, San Diego, CA, USA, May 1993.

[21] C. H. West. Protocol validation in complex systems. In *ACM SIGCOMM*, pages 303–312, 1989.

[22] B. Yang and H. Ural. Protocol conformance test generation using multiple UIO sequences with overlapping. In *ACM SIGCOMM 90: Communications, Architectures, and Protocols*, pages 118–125, Twente, The Netherlands, September 24-27 1990.

[23] I. Zucconi and K. Reed. Building testable software. *Software Engineering Notes*, 21:51–55, 1996.