Test Generation for Networks of Interacting FSMs Using Symbolic Techniques

F. Ferrandi # F. Fummi # E. Macii 1 M. Poncino 1 D. Scuito #

# Politecnico di Milano
Dip. di Elettronica e Informazione
Milano, ITALY 20133

1 Politecnico di Torino
Dip. di Automatica e Informatica
Torino, ITALY 10129

Abstract
This paper presents a new testing strategy for networks of interacting FSMs. The approach allows us to generate test patterns for faults in the network by separately handling the network's components. The proposed algorithms are fully symbolic; therefore, they allow the manipulation of large designs. Experimental results, though preliminary, are promising.

1 Introduction
For control dominated systems, high-level synthesis tools usually produce descriptions of synchronous circuits in terms of small, interacting components [1], each of which can be represented by a finite state machine (FSM). The size and the complexity of the global circuit often suggests the separate synthesis and optimization of each component, and the subsequent interconnection of the so obtained modules. Test generation is then performed on the overall circuit using either structural techniques or approaches based on symbolic state space traversal.

The purpose of this paper is the definition of a testing strategy that allows an independent test generation for faults located in some components of a complex system, thus overcoming the problems of the gate-level test generation approaches that analyze a circuit in its entirety.

Our approach simplifies the testing problem, since it considers only one FSM at a time, that is, the one containing the fault for which a test needs to be generated. Obviously, this increases the complexity of the test generation algorithm, because we are constrained to verify that the identified test sequences can be generated by the surrounding FSMs; in addition, we have to make sure that the symptoms of the fault can be propagated through other FSMs up to the circuit primary outputs. On the other hand, the proposed testing methodology may have some advantages over traditional test generators, since it implicitly analyses circuits which cannot be handled in an explicit way only because of their sizes; thus, our approach should out-perform test generation programs based on symbolic reachability analysis, such as Veritas [2]. In fact, even though with the advent of BDDs [3], symbolic state space exploration algorithms have become extremely powerful [4], when the sizes of the state graphs increase, exact computations may be still infeasible. Different approaches to this problem have been explored; for example, in [5], Cho et al. resorted to approximate computations to calculate an upper bound on the set of reachable states of a large finite state machine. The technique has appeared to be very useful for logic optimization and verification, but not applicable to test generation, except for very special cases [6].

The testing methodology we propose in this paper takes advantage of the implicit enumeration technology to handle complex systems of interacting FSMs. In fact, once a test pattern is generated for a given FSM, the corresponding global pattern (i.e., the pattern to be applied to the whole circuit in order to test the fault under consideration) is computed through symbolic operations on the remaining system components.

The rest of this paper is organized as follows. Section 2 presents some background material used in the sequel; the reader familiar with BDD-based symbolic computations may skip directly to Section 3, where we introduce our test generation strategy and we detail its component algorithms. Section 4 presents preliminary experimental results. Finally, Section 5 is devoted to conclusions and gives directions for future research.

2 Background
2.1 Boolean Operators
Given a n-variable Boolean function, \( f(x_1, x_2, \ldots, x_n) \), the positive and the negative cofactors of \( f \), with respect to variable \( x_i \), are defined by:

\[
f_{x_i} = f(x_1, \ldots, \hat{x_i}, \ldots, x_n)
\]

and

\[
f_{\neg x_i} = f(x_1, \ldots, \hat{x_i}, \hat{\neg x_i}, \ldots, x_n).
\]

Given a Boolean function \( f(x_1, \ldots, x_n) \) over \( B^n \), the existential quantification of \( f \) with respect to \( x_i \) is defined as:

\[
\exists_{x_i} f = f_{x_i} + f_{\neg x_i}.
\]

2.2 Image and Pre-Image
Let \( \Sigma \subseteq A \times A \) be a relation between the elements of set \( A \), and let \( x \) and \( y \) represent generic elements of set \( A \); we have that

\[
\Sigma(x, y) = 1, \text{ with } x \in A \text{ and } y \in A, \text{ if and only if element } x \text{ is in relation } \Sigma \text{ with element } y. \text{ Let } Q \text{ be a sub-set of elements of } A; \text{ the image of } Q \text{ under } \Sigma \text{ is the set of elements of } A \text{ which are in relation with the elements of } A \text{ belonging to } Q. \text{ Formally:}
\]

\[
\text{Im}(\Sigma, Q) = \exists_x (\Sigma(x, y) \cdot Q(x)).
\]

The formula above says, essentially, that the set of elements of \( A \) which are in relation \( \Sigma \) with all the elements in \( Q \subseteq A \) is obtained by selecting from \( \Sigma \) all the pairs \((x,y)\) such that \( x \in Q \), and then by eliminating from such pairs element \( x \). Similarly, we can define the pre-image of \( Q \) under \( \Sigma \) as:

\[
\text{Pre}(\Sigma, Q) = \exists_y (\Sigma(x, y) \cdot Q(y)).
\]
2.3 Finite State Machines
A finite state machine (FSM), \( M \), is defined as the 5-tuple
\[
M = (X, Z, S, S^0, R)
\]
where \( X \) is the input alphabet, \( Z \) is the output alphabet, \( S \) is the finite set of states, \( S^0 \) is the reset state, and \( R \subseteq X \times X \times X \times X \times \{0,1\} \) is the global relation. We have that \( R(x, z, s, t) = 1 \) if and only if, under input \( x \in X \), the FSM makes a transition from present state \( s \in S \) to next state \( t \in S \) outputting \( x \in Z \). A FSM can be represented by a state transition graph (STG), whose vertices are elements of \( S \) and edges are labeled with pairs \((x, z)\).

Given the global relation \( R \) of a FSM, \( M \), the transition relation, \( \Delta \), and the output relation, \( \Lambda \), of \( M \) are defined as:
\[
\Delta(x, z, t) = \exists s R(x, z, s, t) \quad \text{and} \quad \Lambda(x, z, s) = \exists t R(x, z, s, t).
\]

The product machine, \( M = M_1 \times M_2 \), of two FSMS, \( M_1 = (X, Z, S_1, S^0_1, \text{R}_1) \) and \( M_2 = (X, Z, S_2, S^0_2, \text{R}_2) \), is defined as:
\[
M = (X, B, \text{S}_1 \times S^0_1, \text{R}_1)
\]
where \( B = \{0,1\} \), \( S_1 = S_1 \times S_2 \), \( S^0_1 = S^0_1 \times S^0_2 \); in addition, \( \text{R}_1(x, b, s_1, s_2, t_1, t_2) = 1 \) if and only if, under input \( x \in X \), \( M_1 \) makes a transition from \( s_1 \in S_1 \) to \( t_1 \in S_1 \) outputting \( z_1 \in Z \), and \( M_2 \) makes a transition from \( s_2 \in S_2 \) to \( t_2 \in S_2 \) outputting \( z_2 \in Z \). If \( s_1 = s_2 \), then \( b = 1 \), otherwise \( b = 0 \).

The cascade machine, \( M = M_1 \rightarrow M_2 \), of two FSMS, \( M_1 = (X_1, Z_1, S_1, S^0_1, \text{R}_1) \) and \( M_2 = (X_2, Z_2, S_2, S^0_2, \text{R}_2) \), is defined as:
\[
M = (X, Z, S_1 \times S^0_1, \text{R}_1)
\]
where \( X = X_1 \), \( Z = Z_2 \), \( S = S_1 \times S_2 \), \( S^0 = S^0_1 \times S^0_2 \), and \( \text{R}(x, s_1, s_2, t_1, t_2) = 1 \) if and only if, under input \( x \in X \), \( M_1 \) makes a transition from \( s_1 \in S_1 \) to \( t_1 \in S_1 \) outputting \( z_1 \in Z_1 \), and \( M_2 \) makes a transition from \( s_2 \in S_2 \) to \( t_2 \in S_2 \) outputting \( z_2 \in Z_2 \).

2.4 Fault Modeling and Detection
2.4.1 STG-Level Descriptions
Given a FSM, \( M \), a fault \( f \) in \( M \) is such that, at a given time instance, either machine makes an erroneous transition (transition fault), or it outputs an unexpected value (output fault), or both.

Let \( M = (X, Z, S, S^0, R) \) be a finite state machine, and let \( M = (X, Z, S, S^0, R) \) be the FSM obtained from \( M \) by injecting a fault, \( f \), in it. A test sequence for fault \( f \) is a sequence of input vectors such that, when applied to machines \( M \) and \( M \) (both started in their reset states), produces two different output vectors for \( M \) and \( M \). A test sequence, \( TS \), for fault \( f \) is composed of three parts: \( (JS, TV, DS) \). In particular:
- \( TV \) is the test vector, \( x_0, s_0, x_0 \in X \) and \( s_0 \in S \); the application of input vector \( x_0 \) to machines \( M \) and \( M \), both in state \( s_0 \), is such that \( M \) and \( M \) reach different states, say \( s \) and \( s' \).
- \( JS \) is the justification sequence, \( j_1, \ldots, j_n \in X \); the application of \( JS \) to \( M \) (in the reset state, \( S^0 \)) and \( M \) (in the reset state, \( S^0 \)) is such that both machines reach state \( s_0 \).
- \( DS \) is the distinguishing sequence, \( d_1, \ldots, d_p \in X \); the application of \( DS \) to \( M \) (in state \( s \)) and \( M \) (in state \( s' \)) is such that \( M \) and \( M \) output different vectors, say \( z \) and \( z' \).

If \( s_0 \) is the reset state for both \( M \) and \( M \), then \( JS \) is of length zero. Similarly, if \( z \neq z' \) after the application of \( TV \), then \( DS \) is of length zero.

A test sequence is said to be of minimum length if, when applied to the primary inputs of the product machine \( M \times M \), none of the states of \( M \times M \) is visited more than once.

2.4.2 Gate-Level Descriptions
Given a FSM, \( M \), a gate-level implementation can be obtained through state minimization, state encoding, synthesis, and logic optimization. The resulting circuit is a synchronous sequential circuit composed of combinatorial gates and flip-flops. We assume that all flip-flops are controlled by the same clock. We also assume that a reset state is given and that both the good and the faulty circuits can be put in that state. We assume the single stuck-at fault model; because of the assumptions just mentioned, we disregard faults affecting the clock, inside the flip-flops, or inhibiting the reset mechanism. Test sequences to detect stuck-at fault have the same structure of test sequences for the STG-level faults described in Section 2.4.1

3 Test Generation Strategy
3.1 Topology of the Network of Interacting FSMS
We consider FSM networks whose connection graphs are DAGs; therefore, as pointed out by Rho and Somjen in [7], once a specific FSM of the network, \( FSM_1 \), has been selected, it is always possible to find a serial decomposition, i.e., a topological sort, of such network with respect to \( FSM_1 \), even in presence of reconvergent fanout connections. The set of FSMS driving \( FSM_1 \), originates the controlling network (C), while the set of FSMS driven by \( FSM_1 \), originates the observing network (O). As an example, Figure 1 shows the controlling and the observing networks for \( FSM_3 \).

3.2 Testing Algorithm
The pseudo-code of the test generation algorithm is shown in Figure 2. Inputs to the procedure are the FSM containing the fault, \( M \), the controlling network, \( C \), the observing network, \( O \), and the fault, \( f \), for which a test sequence has to be generated. The procedure starts by injecting fault \( f \) into \( M \) to obtain the faulty machine \( M \). Then the set \( TS_M \) of all minimum length test sequences for fault \( f \) are computed and stored as a BDD in terms of the primary input variables of \( M \) (procedure Generate.TS). If \( TS_M \) is non-empty, it is required to determine whether the controlling FSM, \( C \), is able to generate some of the sequences in \( TS_M \). This is accomplished by procedure Prune.TS, which returns all the sequences that \( C \) is able to generate expressed in terms of the input variables of \( M \) (Pruned.TS_M) and in terms of the input variables of \( C \) (TS_C). Clearly, if \( C \) is not able to generate any of the sequences in \( TS_M \), then fault \( f \) is marked as non-excitable with respect to set of sequences that have been considered. The set of states \( IS_C \) of
C originating sequences Pruned,TS_M is computed through procedure Find.1States; then, the sequences of vectors such that C moves from the reset state (S_0^C) to the initial states IS_C are calculated using procedure GenerateJS, and they are expressed in terms of the input variables of C (JS_C) and in terms of the input variables of M (JS_M). Now that a set of test sequences (JS_C+TS_C) has been generated, it is required to verify if the fault can be observed on the primary output lines of machine O. To do this, we first rewrite JS_C and TS_C as functions of the input variables of machine O (JS_O and TS_O); then we determine the set of states OS of O for which the outputs of machine O differ under the application of the distinguishing vectors of TS_O. If no observing state exists or is reachable, then fault f is non-observable with respect to the set of sequences that have been considered. Otherwise, the set of states IS_O, originating sequences IS_O+TS_O is determined; if such a set contains the reset state of machine O, then fault f is observable. Therefore, the set of all minimum length test sequences that detects f (JS_C+TS_C) is returned, together with the times, t_reset_O, at which the backward trace of JS_O+TS_O has hit the reset state of machine O (necessary for test application).

```
procedure Generate.Test(M,C,O,f) {
    MP = Fault.Inject(M,f);
    if((TS_M = Generate.TS(M,MP)) == 0)
        return(f is Untestable);
    ReachRel = FSM.Traaversal(C,S_0^C);
    (Pruned.TS_M,TS_C) = Prune.TS(C,TS_M,ReachRel);
    if(Pruned.TS_M == 0)
        return(f is Non-Excitable);
    IS_C = Find.1States(C,Pruned.TS_M,ReachRel);
    JS_O,JS_M = Generate(JS(C,IS_C));
    JS_O = Symbolic.Simulation(M,M,JS_M);
    TS_O = Symbolic.Simulation(M,MP,Pruned.TS_M);
    ReachRel = FSM.Traaversal(O,S_0^O);
    OS = Observing.States(O,JS_O+TS_O);
    if((OS ∩ ReachRel) == 0)
        return(f is Non-Observable);
    IS_O = Find.Origin.States(C,JS_O+TS_O);
    if((IS_O ∩ S_0^C) == 0)
        return(f is Non-Observable);
    t_reset_O = Reset.Hit(S_0^O,JS_O+TS_O);
    return(JS_O+TS_O, t_reset_O);
}
```

Figure 2: The Generate.Test Algorithm.

3.3 Representing Multiple Test Sequences
The number of test sequences for a given fault, f, in a FSM may be large. Therefore, explicitly enumerating all of them may be infeasible. In this section we introduce an implicit representation we have used to efficiently store all the minimum length test sequences for a target fault, f, in the FSM under test. We start with an example. Figure 3 shows the fault-free and the faulty STGs of a given circuit in presence of fault f.

```
procedure Generate.Test(M,C,O,f) {
    MP = Fault.Inject(M,f);
    if((TS_M = Generate.TS(M,MP)) == 0)
        return(f is Untestable);
    ReachRel = FSM.Traaversal(C,S_0^C);
    (Pruned.TS_M,TS_C) = Prune.TS(C,TS_M,ReachRel);
    if(Pruned.TS_M == 0)
        return(f is Non-Excitable);
    IS_C = Find.1States(C,Pruned.TS_M,ReachRel);
    JS_O,JS_M = Generate(JS(C,IS_C));
    JS_O = Symbolic.Simulation(M,M,JS_M);
    TS_O = Symbolic.Simulation(M,MP,Pruned.TS_M);
    ReachRel = FSM.Traaversal(O,S_0^O);
    OS = Observing.States(O,JS_O+TS_O);
    if((OS ∩ ReachRel) == 0)
        return(f is Non-Observable);
    IS_O = Find.Origin.States(C,JS_O+TS_O);
    if((IS_O ∩ S_0^C) == 0)
        return(f is Non-Observable);
    t_reset_O = Reset.Hit(S_0^O,JS_O+TS_O);
    return(JS_O+TS_O, t_reset_O);
}
```

We assume the reset state to be state 00. The two STGs differ only for the out-going edge of state 00 labeled 00/010; in the fault-free STG this edge is a self-loop, while in the faulty STG it leads to state 01. Then, all test sequences for fault f must start with input 00 applied when the circuit is in state 00, that is, TV = 00. Notice that in this case no justification sequence is needed to reach state 00, since this is the reset state.

All the test sequences are computed by building the expansion graph, BE(TV), for fault f. For example, the expansion graph for the STG in Figure 3 is depicted in Figure 4.

Figure 4: Representation of the Expansion Graph.

Each node in BE(TV) is labeled s/sP to indicate that, after the application of an input sequence, M and MP will be in states s and sP, respectively. Such a sequence is obtained by concatenation of the edge labels on the path connecting the root node of BE(TV) to node labeled s/sP. The root node is associated to the pair of states representing the reset states in M and MP, respectively. Graph BE(TV) is levelized. All nodes of a level are reached from the root node by an input sequence of the same length. Furthermore, it is acyclic by construction, as shown in Section 3.4.

We can represent the expansion graph in an implicit form by associating two relations to each level of the graph. The first relation, named PDS, stores all the transitions for which M and MP are characterized by the same output vector. These transitions connect nodes of two adjacent levels in the graph. For a generic level of the graph, i, the relation is:

```
PDS[i] ⊆ S × S_p × X × S × S_p → {0,1).
```

We have that PDS[i](s, sP, x, t, tP) = 1 if and only if, under input vector x ∈ X, the fault-free machine M makes a transition from present state s ∈ S to next state t ∈ S, and the faulty machine MP makes a transition from present state sP ∈ S_p to next state tP ∈ S_p. The second relation, named SDS, stores all the transitions for which M and MP produce a different output vector. For a generic level, i, the relation is:

```
SDS[i] ⊆ S × S_p × X → {0,1).
```

Similarly to the PDS case, SDS[i](s, sP, x) = 1 if and only if, under input vector x ∈ X, the output vector generated by M with a transition out-going from state s ∈ S differs from the output vector generated by MP with the transition out-going from state sP ∈ S_p. For instance, in Figure 4, the transition from node 00/01 to node 11/10 with input 10 belongs to the PDS[i] relation, while the transition leaving node 11/10 and labeled 0— belongs to the SDS[i] relation.

To summarize, representing the expansion graph as a whole requires an array of relations; the size of such array is bounded by twice the depth of the graph. This representation allows to implicitly describe multiple sequences without mixing transitions belonging to distinct sequences.
3.4 Test Generation for Faults in Machine M

In Section 3.3 we have shown how a set of sequences can be symbolically represented as an array of BDDs. Here we describe procedure Generate.TS that generates all the test sequences for a target fault, f, in machine M (see the pseudo-code in Figure 5).

The TV set includes all test vectors for the target fault, i.e., all input vectors to the combinational part of the FSM under test that originate from a different output vector (on the outputs or next-state lines) in presence of the target fault. Such vectors correspond exactly to the transitions for which the STGs of M and Mp differ. Obviously, if no test vector can be computed, the fault is classified as combinatorially redundant. The TV set can be obtained by applying a combinational test pattern generator that does not stop the test generation at the first identified test vector, but proceeds until the entire state-space of the fault has been explored. On the contrary, since the proposed testing approach needs the faulty FSM description (Mp), we can obtain the global set of test vectors by comparing M and Mp.

Let R(z, s, t) be the global relation of the FSM under test and let Rp(z, s, t) be the global relation of the faulty FSM. Then, the set TV is given by:

\[ TV(s, s) = \exists z, R(z, s, t) \cdot \exists z, Rp(z, s, t) \]

The TV set must be pruned from all test vectors which cannot be reached starting from the reset state of M and Mp, that is, a justification sequence, JS, must be identified for each test vector. This analysis is performed by the Justify procedure. It is based on a forward traversal of M and Mp, by searching for justification sequences which drive the fault-free and faulty FSMs into the activation states of the test vectors. The JS set if stored into an array of BDDs by using the method described in Section 3.3. The absence of justification sequences (i.e., JS == 0) certifies the fault as sequentially non-excitatory [2]. Finally, procedure Distinguish calculates all minimum-length distinguishing sequences, DS, that are able to propagate to the output lines of the FSM a fault observed on next state lines only. If no distinguishing sequence exists (i.e., DS == 0) the fault is classified as non-distinguishable [2]; otherwise, the three components of the test sequences are combined to produce the TS set. Procedure Distinguish is the core of the test sequences generation. It is performed by applying the following steps, level by level, thus implicitly building the expansion graph described in Section 3.3. For each level i, the relation PDS[i] is computed as:

\[ PDS[i](s, s, s, s, p, t, t, p) = \exists u, PDS[i-1](u, s, s, p, t) \cdot R(s, s, s, t) \cdot R(s, s, s, t, p) \]

and the SDS[i] is calculated as:

\[ SDS[i](s, s, s, p, t) = \exists z, RS(s, s, s, t) \cdot \exists z, Rp(s, s, s, t, p) \cdot AS[i-1](s, s, s, p) - \exists z, z, PDS[i](s, s, s, s, p, t, t, p) \]

where:

\[ AS[i](s, s, s, p) = to.set[s](t, t, p) \cdot (t \equiv s) \cdot (t \equiv s, p) \]

if i \neq 0, and

\[ AS[0](s, s, s, p) = TV(s, s, s, p) \cdot (s \equiv s, p) \]

otherwise. Furthermore,

\[ to.set[s](t, t, p) = \exists z, s, p, PDS[i](s, s, s, s, p, t, t, p) \]

The computation of the expansion graph terminates whenever 3j < i : to.set[i](t, t, p) = to.set[j](t, t, p). Then, the expansion graph is implicitly pruned by all transitions belonging to paths not reaching at least a node of a SDS[i].

Notice that procedure Distinguish may be used to generate directly the entire TS set if applied to the reset state of M and Mp. The choice between the two methods depends on efficiency considerations. In fact, the separate identification of the three parts of a test sequence can prevent useless computations if redundant faults are present into the circuit. On the contrary, if the circuit associated to M is guaranteed to be 100% irredudant, when considered in isolation, the direct use of procedure Distinguish is preferable.

\begin{verbatim}
procedure Generate.TS(M, M_p) {
    TV = Identify.Test.Vectors(M, M_p);
    if (TV == 0) return(0);
    JS = Justify(TV, M, M_p);
    if (JS == 0) return(0);
    DS = Distinguish(TV, M, M_p);
    if (DS == 0) return(0);
    TS = Combine(JS, TV, DS);
    return(TS);
}
\end{verbatim}

Figure 5: The Generate.TS Algorithm.

3.5 Sequence Justification

After all the minimum length test sequences for fault f have been generated, they have to be justified through the controlling FSM, C. This is accomplished in three steps:

1. Pruning of TS_M and generation of TS_C;
2. Computation of the state set of C originating TS_C;
3. Generation of the justification sequences, JS_C, that drive machine C from the reset state to the set of initial states generated in step 2.

In the presentation of the justification algorithms, variables s, t, and z denote present state, primary input, next state, and primary output variables of C, respectively. Variables u and v denote present and next states of the product machine, Mp, of M and Mp. Clearly, the z variables also denote the input variables of Mp.

Procedure Prune.TS, shown in Figure 6, performs the first phase of the justification process. It is based on a simultaneous traversal of C and Mp; however, its advantage over a plain traversal of the composition of C and Mp is that the behavior of Mp is constrained by the given output sequences. The relation s2u, that keeps the correspondence between present states s of C and u of Mp, is first computed. Then, we iterate n times, where n is the length of the longest test sequence. At each step, we compute the relation P, which represents the sequences PDS with states u replaced by states s. P is then augmented to include all the valid transitions in Rc. The resulting relation, V, includes primary inputs z and next states t of C. In other words, V represents the transitions (s, x, t, z) of C which cause Mp to produce v as next states. By quantifying out variables z and v from V, we first compute the current input pattern X[i]. Notice that we store transitions rather than plain input patterns, as in the case of the output sequences (see Section 3.3). Quantification of input variables x yields the relation Q, which is used for updating the current frame of PDS. This is carried out by first matching states s of Q with states u of s2u, and then by quantifying out variables s and t. Intersecting the so obtained set of sequences with PDS removes the invalid sequences from

211
Then, if the current frame of $SDS$ is not empty, we try to prune it as in the case of $PDS$. If the updated $SDS$ is still not empty, we pick a set of states $UTo$ of $MP$ representing the distinguishing states at the current time frame. These states will be used in the second phase. Finally, the relation $sFu$ is updated through quantification of $s$ and $x$ from $Q$. After $n$ iterations, the procedure returns the set of input patterns, the updated sets of sequences, and the set of distinguishing states. The set $UTo$ has to be backtracked to determine the states of $C$ that can generate $TSC$. Procedure Find.$I$.States, shown in Figure 7, does this job by performing a back - forward traversal on $C$, starting from the last set of distinguishing states $UTo[m - 1]$. At each step, $i$, the set of vectors $Z^i$ implied by the updated sequences $PDS$ is computed. Then, the set of states $GS^{i - 1}$ reachable from states in $GS + UTo[i]$, through edges in $RC$ with output labels in $Z^i$ are determined using constrained pre-image computation. After $n$ iterations, the set of reachable states originating all the sequences in $PDS$ is returned.

The third phase of the justification process is described in Figure 8. Generate.$JS$ works in two steps; in the first one, $C$ is traversed forward, starting from its reset state. At each iteration, $i$, the set $Tbol$ of destination states is intersected with the set $Init$ returned by procedure Find.$I$.States, and the result, $Just^i$, is substracted from $Init$. The iterations stop as soon as $Init$ is empty. Termination of the for loop is guaranteed by the fact that all states in $Init$ are reachable. The second step looks alike the backtracking phase of Find.$I$.States, but it is carried out on relation $RC$. Finally, sequences $JS$ and the corresponding input patterns are returned.

**Figure 6: The Prune.$TS$. Algorithm**

**Figure 7: The Find.$I$.States Algorithm**

**3.6 Fault Observation**

The last phase of the test generation process checks if the test sequences computed so far make the effects of fault $f$ propagating to the primary outputs of machine $O$. Two basic steps are required to perform such a check: 1. Finding the set of states in $O$ for which fault $f$ is observable; 2. Backtracking the test sequences through machine $O$ starting from states determined during step 1. The algorithm for step 2 is similar to procedures Find.$I$.States and Generate.$JS$, so it is not discussed here. Calculating the set of observing states for machine $O$, on the other hand, requires some non-trivial symbolic operations, which are illustrated in Figure 9.

Procedure Observing.$States$ receives, as input parameters, the BDD for the global relation $RO$ of machine $O$, and the set of sequences $ODS$ derived from $SDS$ as follows:

$$ODS\{[s, sp, z, zP] = 3_{es}(R_M(s, sp, z); SDS\{[s, z]})$$

where $R_M$ and $R_{MP}$ are the global relations for machines $M$ and $MP$, respectively. Therefore, $ODS$ represents the set of sequences that are driving machines $M$ and $MP$ to a distinguishing state.

The calculation of the observing states, $OS$, is carried out on a frame-by-frame basis (iteration on the maximum length, $n$, of the sequences in $ODS$). Relation $P$ is initially computed; $P$ identifies the states $s_0$ of machine $O$ with output edges labeled $(x_0 = z)/z_0$ that correspond to states $s$ and $sp$ in $M$ and $MP$ whose outgoing edges are labeled $xP/ zP$ respectively. Then, relation $Q$ is constructed in a similar way (in this case, edges leaving states $s_0$ are labeled $(x_0 = zP)/z_0$). Next, relation $H$ is determined by existentially quantifying the $z_0$ variables from $P$ and $Q$ and by taking the intersection of the quantified relations. $H$ represents, in essence, the set of states in machine $O$ having two outgoing edges, $e_a$ and $e_b$, labeled $x_0 = z/ z_0 = (x_0 = zP)/z_0$, respectively. Among these states, we need to select only the ones for which the output labels of edges $e_a$ and $e_b$ differ; to do this, we determine the set of states, $K$, for which $e_a$ and $e_b$ are both labeled $\sim -z_0$, by intersecting relations $P$ and $Q$ and then by quantifying out variables $z_0$ from the result. Now, we only need to subtract set $K$ from set $H$ and to quantify out the $z$ and $zP$ variables to obtain set $OS$.  

**Figure 8: The Generate.$JS$. Algorithm**
As in any BDD-based algorithm, performance of our test generation program heavily depends on the variable orderings used to build and manipulate the BDDs representing sets of states, relations, and sets of sequences. Results in Table 1 were obtained using a fixed ordering computed through inspection of the FSM network. Improvements are expected by the introduction of dynamic re-ordering in the current version of the code.

5 Conclusions and Future Work

The main contribution of this paper is the definition of a testing methodology that allows us to generate the test patterns for faults in a FSM network without explicitly constructing the global product FSM. All the proposed algorithms are fully symbolic, that is, they completely rely on BDD-based, implicit enumeration techniques to handle a large amount of information (reachable states of a FSM, set of test sequences, etc.); as a consequence, reasonably wide networks of interacting FSMs can be processed by our procedures in very short times. Experimental results are very promising.

Acknowledgments

We wish to thank Antonio Liwy, Abelardo Pardo, and Fabio Somensi for their comments on an earlier draft of this paper.

References


