Test Sequence Generation and Model Checking Using Dynamic Transition Relations

Sérgio Campos[1] Orna Grumberg[2] Karen Yorav[2] Copty Fady[3]

1 Federal University Minas Gerais, Belo Horizonte, Brasil; scampos@dcc.ufmg.br
2 The Technion, Haifa, Israel; {orna,laster}@cs.technion.ac.il
3 Intel Corporation, Haifa, Israel; fady.copty@intel.com

Abstract. The task of finding a set of test sequences that provides good coverage of industrial circuits is infeasible because of their size. For small critical sub-circuits of the design, however, designers can create a set of test sequences that achieve good coverage. These sequences cannot be used on the full design because the inputs to the sub-circuit may not be accessible. In this work we present an efficient *test generation algorithm* that receives a test sequence created for the sub-circuit and finds a test sequence for the full design that reproduces the given sequence on the sub-circuit. The algorithm uses a new technique called *dynamic transition relations* to increase its efficiency.
The most common and most expensive step in our algorithm is the computation of the set of predecessors of a set of states. To make this computation more efficient we exploit a partitioning of the transition relation into a set of simpler relations. At every step we use only those which are necessary, resulting in a smaller relation than the original one. A different relation is used for each step, hence the name *dynamic* transition relations. The same idea can be used to improve symbolic model checking for the temporal logic CTL.
We have implemented the new method in SMV and run it on several large circuits. Our experiments indicate that the new method can provide gains in time and space during verification of up to two orders of magnitude. These results show that dynamic transition relations can make it possible to verify circuits that were previously unmanageable due to their size and complexity.

1 Introduction

In recent years we have seen a rapid growth in the complexity and size of industrial designs. The verification task of such systems has become extremely complex, requiring new techniques that can handle large scale designs. Formal methods are an example of a new technology that has gained popularity recently in this context. One such method, symbolic model checking has been very successful in hardware verification. Model checkers have been able to find several previously unknown errors in industrial circuits. Many companies are starting to use symbolic model checking in their design cycles as a complement to dynamic validation. We propose a new technique based on symbolic model checking called

dynamic transition relations that enhances both formal verification and dynamic validation.

Dynamic validation (or *simulation*) checks that a given run of a system is correct by inputing a sequence of signals (a *test sequence*) to the design and observing the resulting outputs. Since the number of runs is infinite, the method cannot be exhaustive, i.e., it cannot check all possible runs. Thus, it is important to obtain a set of test sequences that provides a good coverage of the design. However, this task might not be feasible if the design is too large. Creating a set of test sequences covering most of the behaviors of the circuit is extremely difficult due to the large number of behaviors that it can exhibit. There are situations, however, in which a set of test sequences can be assumed to exist.

- It is important to test small critical sub-circuits extensively. For small circuits, it is easier to generate efficient test sequences. For critical circuits it is worth the extra effort to identify test sequences that can provide a good coverage.
- Sub-circuits are frequently reused in new designs, and in these cases test sequences often have been developed for them.
- Sub-circuits can also be verified separately (using simulation or formal methods, for example). If an error is found in a sub-circuit the designer must check if it is a real error. It may not exist in the full design because the sequence of inputs that lead to the error can never occur in the complete circuit.
- Important conditions in the full design can often be stated as test sequences. For example, the violation of mutual exclusion can be stated as a condition on internal signals (e.g. $granted_0 \wedge granted_1$), which can be seen as a test sequence of length one.

However, test sequences for sub-circuits cannot be used directly in the full system. The inputs to the sub-circuit can be internal signals of the full circuit which may not be accessible from the exterior of the design. We address the problem of creating a test sequence for the full design that will reproduce the test sequence of the small sub-circuit and thus achieve good coverage of the sub-circuit by running tests on the full design.

The problem can be defined as follows (see figure 1). Let D be a large design with a set of inputs I, and let Sub be some relatively small critical sub-circuit of D, with a set of inputs I_{sub}. D is a single clock synchronous circuit, or an equivalently transformed one. Assume that the designer produces a set of test sequences that guarantees a good coverage of Sub. In general, the inputs to Sub are not accessible from the exterior of D. Thus, we need to provide a test generation algorithm that given a test sequence $\Pi_{sub} = t_0, \ldots, t_n$ on I_{sub}, either produces a test sequence Π on I that induces Π_{sub}, or reports that no such sequence exists. In the latter case, it means that Π_{sub} can never occur when Sub is run within D.

Our test generation algorithm performs this task in two phases. The first is a backwards search on D that creates a sequence of sets of states in reverse order, from last to first. Any computation path that goes through these sets

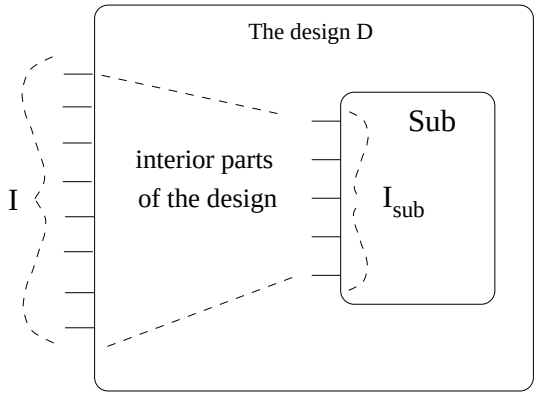

Fig. 1. A schematic view of the test generation problem

induces the required Π_{sub}. The second phase traverses the sets from first to last determining one such path by choosing an appropriate state from each set of states. A sequence of inputs Π that corresponds to the chosen path is then projected from the path. The resulting Π is a correct input for a simulation tool that will reproduce Π_{sub}. If there is no such path, the algorithm returns "false".

Dynamic Transition Relations The main operator in the backwards search of our algorithm is $Pred(A)$, which computes the set of all predecessors of the states in A. A state is defined as an assignment of values (valuation) to all variables in the model. A variable is the output of a latch or any state holding signal in the circuit in addition to its inputs. $Pred(A)$ uses as a model for D a transition relation defining the value of system variables in the next state as a formula on the value of system variables in the current state. This operation might be very costly if the transition relation is large. To avoid this problem, we exploit a partition of the transition relation into simpler relations. Each such relation defines the value in the next state for a single system variable. In computing the predecessors of A we apply only those relations that define the next value for variables on which A depends. Thus, the transition relation used in the backwards search is determined *dynamically* for each application of $Pred$, according to the set it is applied to. The new dynamic operator is called $Pred_D$. As the search proceeds, the number of variables A_i depends on may grow larger or smaller, and the sizes of the dynamic transition relations vary accordingly. The search applied in the second phase uses at each step the transition relation determined dynamically during the backwards search.

$Pred_D$ is particularly useful if the algorithm is implemented symbolically, using a BDD [2] representation of the design. BDDs can often provide very concise representations. However, their size tend to grow with the number of variables they depend on. But the next value of each variable typically depends

on a small number of other variables, and so each of the dynamic transition relations will involve fewer variables than the global transition relation. As a result, the BDD representations will often be smaller and the computation of $Pred_D$ faster.

Dynamic Transition Relations and Model Checking The introduction of dynamic transition relations and the $Pred_D$ operator has been motivated by the test-generation algorithm that depends on it. However, the operation of finding the set of predecessors of a set is also the core operation in other algorithms that manipulate finite-state machines. The most prominent example is in *Model Checking* where a finite-state machine that models a system is checked against a given specification to prove that it works correctly or find a counterexample. We show that the same idea of dynamic transition relations can be used in order to improve *symbolic model checking* [3] for the temporal logic CTL.

Similarly to the test generation algorithm, one of the most common and most expensive steps in symbolic model checking algorithms is the computation of $Pred$ (usually referred to by the temporal operator **EX**). Replacing $Pred$ by $Pred_D$ reduces space and time consumption and allows the verification of larger systems. As before, the partial transition relations are computed dynamically for each application of $Pred_D$. Note that, the result of our computation is *exact*, i.e., we obtain the same set of states as we would have if we used the global transition relation. Our changes only affect some of the intermediate results.

The method proposed has several advantages. As opposed to several algorithms frequently used for test sequence generation, dynamic transition relations guarantee that test sequences will be found when they exist. Also, the gains obtained by using as little of the model as possible are automatic, there is no need to manually decide which signals to consider and which not. It can potentially never consider large parts of the circuit, if they do not affect the behavior of the sub-circuit under consideration. Because of this it can be applied to much larger circuits than most other methods. Moreover, the method only changes the $Pred$ function, not affecting the behavior of the tool in any other way. As a consequence, it can be applied *in addition* to most other optimizations such as partitioned transition relations, assume-guarantee and others for even better results.

Experimental Results We have implemented dynamic transition relations and the test sequence generation algorithm within the symbolic model checking tool SMV [13]. We have obtained a significant improvement in time and space on a variety of examples such as the PCI local bus [4], a distributed heterogeneous real-time system [5] (for model checking) and test sequence generation for several Intel circuits. The new algorithm used on average only 34% of the variables. Moreover, our algorithm has been able to determine test sequences (or show that they do not exist) in many cases when the static algorithm was unable to do so. The static algorithm in these cases used more than 800M of memory and could not finish. The dynamic algorithm, on the other hand, finished on average in 16 seconds using 5.1M of memory.

These results show that dynamic transition relations can provide significant gains for both test sequence generation and symbolic model checking. It can enable the analysis of larger systems than previously possible: In some cases we have been able to obtain gains of two orders of magnitude. The fact that it changes only the *Pred* function enables its use with minimal changes in several verifiers.

Related Work Test sequence generation from precomputed tests has been studied out of the context of symbolic model checking [14, 6, 12, 11, 15, 10]. However, all of these techniques depend on the internal structure of the circuit being tested, (e.g. in some cases they rely on a regular bus structure in the design), and consequently are not as general as the one presented here. In the context of model checking the problem has been addressed in [9]. But their work is concerned mostly with expressing the test sequence and not with the complexity of dealing with large circuits. In fact, in [9] the authors state that they have only used the method in small examples. Another work relating to model checking and test sequence generation is [1]. However, in that work ATPG algorithms (Automatic Test Pattern Generation) are used to perform model checking, and ATPG algorithms do not guarantee that they will eventually find a test sequence even in cases when it does exist.

One important aspect of our work is that is does not build a model of the complete circuit beforehand and that it may never actually construct such model. In this aspect it relates to techniques such as partitioned transition relations [7]. It differs from it, however, because in that case all partitions are used in every iteration, and this may not be necessary. Examples of techniques that may not consider all parts of the circuit are the cone of influence reduction [7]. However, these techniques are static in the sense that they determine only once which parts of the circuit can be ignored. Our method does it dynamically taking advantage that not all parts of the circuit are relevant during all iterations. Because of this our method produces better results, since it can use less of the circuit during most iterations. In fact, the cone of influence can be seen as an upper bound on the behavior of our algorithm.

Our paper is organized as follows. Section 2 first presents our test generation algorithm, using a global transition relation. It then shows how to convert the test generation algorithm to use dynamic transition relations, and explains how this algorithm can be implemented with BDDs. Next, in Section 3 we show how dynamic transition relations are used for symbolic model checking. In Section 4 we present experimental results that show the usefulness of our ideas. Finally, section 5 gives our conclusions from this work and some future directions of research.

2 Test Sequence generation

We model synchronous hardware designs as *finite state machines*. Let I_{sub} be the set of inputs to the sub-circuit and I be the set of circuit input variables,

and let V be the set of circuit variables that are not inputs, $I_{sub} \subseteq I \cup V$. We assume that all circuit variables are boolean.

We use boolean formulas to represent sets of states and transition relations. Each state in the finite state machine representing a circuit is an assignment of values to all variables in $V \cup I$. A formula f with free variables (I, V) represents the set of states (assignments) that satisfy f. A transition relation is represented by a formula R with free variables (I, V, I', V') so that an assignment that satisfies R represents an edge from (I, V) to (I', V'). We use the same letter for a set of states (or transition relation) and the formula that represents it.

A finite state machine modeling a circuit is a tuple $M = (Init, S, \{N_1, \ldots, N_l\})$, where

- The set of states S is a set of valuations for $I \cup V$.
- $Init \subseteq S$ is the set of initial states.
- The transition relation is defined by a set of functions N_i, each defining the next-state value for $v_i \in V$ by $v_i' = N_i(I, V)$. The *global* transition relation is $R = \bigwedge_{v_i \in V}(v_i' = N_i(I, V))$. We assume that every N_i is a deterministic total function. Note that the inputs are unrestricted.

A *trace* of M is a sequence $\Pi = s_0, s_1, \ldots, s_n$ such that $(s_i, s_{i+1}) \in R$. Let $U \subseteq (I \cup V)$ be a set of variables. A *partial assignment* with respect to U is an assignment that gives values only to variables in U (as opposed to a state, which is an assignment that gives values to all the variables of the circuit). A partial assignment σ with respect to U represents the set of states s that *agree* with σ on U, i.e. for every $v \in U$, $s(v) = \sigma(v)$. The *projection* of a state s on U is a partial assignment over U that agrees with s. The projection of Π on U is the trace obtained by taking the projection of each state in Π on U. An *expansion* of σ to U' (s.t. $U \subset U'$) is a partial assignment σ' over U' that agrees with σ on U. A test sequence over U is a series of partial assignments over U.

Formally, our problem is defined as follows. Given a set I_{sub} of variables from $I \cup V$, which are the inputs to the critical sub-circuit Sub, and a test sequence $\Pi_{sub} = t_0, \ldots, t_n$ over I_{sub}, we must produce an initial state $s_{init} \in S$ and a test sequence $\Pi_{in} = in_m, \ldots, in_0, \ldots, in_n$ over I $(m \leq 0)$ where every in_i is a partial assignment over I. Let $\Pi = s_m, \ldots, s_0, \ldots, s_n$ be the trace generated by Π_{in} in M from s_{init} $(s_m = s_{init})$ [1]. We require that the projection of $s_0, \ldots, s_n$ on I_{sub} will be identical to Π_{sub}.

2.1 A Basic Static Algorithm

In this section we present an algorithm that uses the global transition relation R to determine a test sequence for the circuit. The algorithm has two stages. The first is a backwards search that creates a series of sets of states. Any computation path that goes through these sets (starting at an initial state) is a solution to

[1] Notice that the transition relation R defines the next-state value for variables in V. By choosing a next-state value for the inputs we deterministically define a successor state. Therefore, s_{init} and Π_{in} (together with R) uniquely determine Π.

our problem. Given $\Pi_{sub} = t_0, \ldots, t_n$ we construct a series of sets of states A_m, $\ldots$, A_0, $\ldots$, A_n in reverse order, i.e. we start by computing the set A_n and end with A_m. For every $m \leq i \leq (n-1)$, every state $s \in A_i$ has a successor in A_{i+1} (see figure 2). Using a slight abuse of notation, we view every t_i in the test sequence as a set of states, the set of states (assignments to all variables) that agree with t_i. When creating the sets A_0, $\ldots$, A_n we make sure that $A_i \subseteq t_i$. Thus, A_0, $\ldots$, A_n represent the set of traces that agree with Π_{sub}. In order to make sure that a trace that runs through these sets can be created starting at an initial state, we continue to compute A_{-1}, A_{-2} and so on, until we arrive at a set A_m in which there is an initial state. If such an initial state can be found, we know that there is a trace Π that, from some point on, reproduces the test sequence Π_{sub}. If, however, we arrive at a set $A_i = \emptyset$ or $A_i \subseteq \bigcup_{j=i+1}^{0} A_j$, we can conclude that there is no input sequence Π_{in} that can be used from an initial state to reproduce Π, and we report this.

In the second stage we traverse the sets, from A_m up to A_n and find one suitable trace $\Pi = s_m, \ldots, s_n$. The test sequence Π_{in} that generates this trace is created by taking the projections of the states along Π on I. The output of the algorithm is an initial state s_m, and a sequence of inputs in_m, $\ldots$, in_n.

This algorithm uses the operator $Pred$, which computes the set of predecessors of a set of states A according to the transition relation R. It is defined by: $Pred(A) = \exists I', V'.R(I, V, I', V') \wedge A(I', V')$. This can be informally interpreted as there exists a state (an assignment to I', V') that satisfies A ($A(I', V')$ is true), and this state is the end of a transition in R ($R(I, V, I', V')$ is true). Once an assignment to I', V' is fixed, R also determines all of its predecessors, expressed in terms of I, V [13]. It also uses the function $choose(A)$, which receives a set A and produces a single state (assignment to $I \cup V$) $s \in A$, and the function $succ(s)$, which returns the set of successors of a state according to R.

Stage I:

1: $A_n := t_n$
2: for $j = n - 1$ downto 0 do
3: $\quad A_j := t_j \cap Pred(A_{j+1})$
4: endfor
5: $All := \emptyset$
6: while $(A_j \neq \emptyset) \wedge (A_j \cap Init = \emptyset) \wedge (A_j \not\subseteq All)$ do
7: $\quad All := All \cup A_j$
8: $\quad j := j - 1$
9: $\quad A_j := Pred(A_{j+1})$
10: endwhile
11: $m := j$

Stage II:

12: if $(A_m = \emptyset) \vee (A_m \subseteq All)$ then
13: $\quad$ print "sequence generation failed"
14: else
15: $\quad s_m := choose(A_m \cap Init)$;

16: in_m := the projection of s_m on I;
17: for $j = m + 1$ to n do
18: $s_j := choose(succ(s_{j-1}) \cap A_j)$;
19: in_j := the projection of s_j on I;
20: endfor
21: endif

Fig. 2. The data structure created by the test sequence generation algorithm

2.2 Dynamic Transition Relations: A More Efficient Algorithm

The algorithm presented in the previous section might not be practical for very large circuits. In such circuits, the transition relation R is too big even if it is represented by a BDD. As a result, for a set of states A the operation $Pred$ becomes too expensive.

It is possible, however, to exploit a partition of the transition relation into functions N_i that define the next state variables v_i'. Recall that a state of our model gives values to all the variables in $I \cup V$. We say that a set of states A *is independent* of a variable v_i, if for every state s in A, the state that differs from s only on v_i is also in A. In such a case, we can exclude the function N_i from the computation of $Pred(A)$. We say that a formula f is independent of a variable v if for every two assignments σ and σ' that differ only on v, $\sigma \models f$ iff $\sigma' \models f$. Notice that if the formula f is independent of v then the set of states that it represents is independent of v.

Let f be a formula (representing a set of states or a transition relation), and let the support of f be the set of variables on which f depends. We define $sup(f) = \{v \in (V \cup I) \mid f \text{ depends on } v\}$ and $sup'(f) = \{v \in (V \cup I) \mid f \text{ depends on } v'\}$. Therefore, $sup(f)$ is the set of current-state variables that f depends on and $sup'(f)$ is the set of next-state variables that f depends on.

We define a dynamic version of $Pred$, called $Pred_D$. The operator $Pred_D(A)$ computes the set of predecessors of states in A according to the *partial transition relation* $\bigwedge_{v_i \in sup(A)}[N_i(s) = v_i']$, which is a transition relation that includes N_i if and only if A depends on v_i. The operator $Pred_D$ is therefore defined as: $Pred_D(A) = \exists I', V'[A(I', V') \wedge \bigwedge_{v_i \in sup(A)}[N_i(s) = v_i']]$

Lemma 1. *For any set A, $Pred(A) = Pred_D(A)$.*

Proof: Assume that the variable v_i does not appear in the support of A. We start with the definition of $Pred$:
$\mathrm{Pred}(A) = \exists V'.[(v'_1 = N_1(V)) \wedge \ldots \wedge (v'_n = N_n(V)) \wedge A(V')]$

$$\equiv \exists v'_1, .., v'_{i-1}, v'_{i+1}, .., v'_n. \\ [(v'_1 = N_1(V)) \wedge ..(v'_{i-1} = N_{i-1}(V)) \wedge (v'_{i+1} = N_{i+1}(V)) \wedge .. \\ (v'_n = N_n(V)) \wedge \exists v'_i.[(v'_i = N_i(V)) \wedge A(V')]]$$

Since $A(V)$ does not depend on v_i, $A(V')$ does not depend on v'_i and we can move it through the $\exists v'_i$ quantifier to get:

$$\equiv \exists v'_1, .., v'_{i-1}, v'_{i+1}, .., v'_n. \\ [(v'_1 = N_1(V)) \wedge ..(v'_{i-1} = N_{i-1}(V)) \wedge (v'_{i+1} = N_{i+1}(V)) \wedge .. \\ (v'_n = N_n(V)) \wedge A(V') \wedge \exists v'_i.[v'_i = N_i(V)]]$$

We assume that every N_i is a total function, i.e. for every V there exists v'_i s.t. $v'_i = N_i(V)$, so:

$$\equiv \exists v'_1, .., v'_{i-1}, v'_{i+1}, .., v'_n. \\ [(v'_1 = N_1(V)) \wedge ..(v'_{i-1} = N_{i-1}(V)) \wedge (v'_{i+1} = N_{i+1}(V)) \wedge .. \\ (v'_n = N_n(V)) \wedge A(V')]$$

The above shows that for every variable v_i that does not appear in the support of A we can drop the term $v'_i = N_i(V)$ from the transition relation part of $Pred(A)$ without changing the result. If we do this for **all** variables not in the support of A we get $Pred_D(A)$.

If the next-state value of each variable depends only on a few of the other variables (as is often the case in circuits), the support of the sets we compute will remain small.

We recall that given a state s, the application of R to s determines the next-state values for the variables in V, but not for the variables in I. The input variables I are chosen arbitrarily by the environment. The dynamic algorithm uses partial assignments σ_i instead of the full states s_i, and partial transition relations R_i instead of R. The output sequence $in_m, \ldots, in_0, \ldots, in_{n-1}$ generated by the algorithm is a series of partial assignments over some (but perhaps not all) of the variables in I. When in_i does not give a value for a variable $v \in I$ it means that v does not influence the parts of the circuit that are being considered, and its value can be chosen arbitrarily.

In the dynamic algorithm we use the following functions:

- $choose(A, U)$ accepts a set of states represented by a formula A and a set of variables U such that $sup(A) \subseteq U$. It returns a partial assignment σ over U that satisfies A. If we view σ and A as sets, then the chosen σ is a subset of A. Notice that if A is given as a partial assignment a, then the resulting σ will be an assignment over U that agrees with a. Notice also that the function *choose* that was used in the algorithm of the previous subsection is just a call to this function with $U = I \cup V$.
- $expand(\sigma, U')$ receives a partial assignment σ over some set of variables U, and a set of variables $U' \supseteq U$. It expands σ to create a partial assignment over U'. The values for variables in $U' \setminus U$ are chosen arbitrarily.
- $project(s, U)$ receives a state and a set of variables and returns the projection of the former on the later, i.e. it returns a partial assignment over U that agrees with s.
- $apply(R_i, \sigma)$ receives a partial transition relation R_i and a partial assignment σ over $sup(R_i)$. The partial transition relation is of the form $R_i = \bigwedge_{v \in U}(v' = N_v(I, V))$ for some set of variables U. The result is a partial assignment σ' over U, such that for every $v \in U$: $\sigma'(v) = N_v(\sigma)$.

Stage I:

1: $A_n := t_n$
2: for $j = n - 1$ downto 0 do
3: $\quad R_{j+1} := \bigwedge_{v_k \in sup(A_{j+1})} v'_k = N_k(I, V)$
4: $\quad A_j := t_j \wedge Pred_D(A_{j+1})$
5: endfor
6: $All := \emptyset$
7: while $(A_j \neq \emptyset) \wedge (A_j \cap Init = \emptyset) \wedge (A_j \not\subseteq All)$ do
8: $\quad All := All \cup A_j$
9: $\quad j := j - 1$
10: $\quad R_{j+1} := \bigwedge_{v_k \in sup(A_{j+1})} v'_k = N_k(I, V)$
11: $\quad A_j := Pred_D(A_{j+1})$
12: endwhile
13: $m := j$.

Stage II:

14: if $(A_m = \emptyset) \vee (A_m \subseteq All)$ then
15: $\quad$ print "sequence generation failed"
16: else
17: $\quad \sigma_m := choose(A_m \cap Init, sup(R_{m+1}))$
18: $\quad s_{init} := expand(\sigma_m, I \cup V)$
19: $\quad in_m := project(s_{init}, I \cap sup(R_{m+1}))$
20: $\quad$ for $j = m + 1$ to $n - 1$ do
21: $\quad\quad tmp := A_j \wedge apply(R_j, \sigma_{j-1})$
22: $\quad\quad \sigma_j := expand(tmp, sup(R_{j+1}))$
23: $\quad\quad in_j := choose(\sigma_j, I \cap sup(R_{j+1}))$
24: $\quad$ endfor
25: $\quad \sigma_n := A_n \wedge apply(R_n, \sigma_{n-1})$

26: $in_n := choose(\sigma_n, I)$
27: endif

Stage I of the algorithm creates the same sets A_m, ..., A_n that were created in the previous algorithm, but uses $Pred_D$ instead of $Pred$. Notice that by the end of stage I we have that for every $m < i \leq n$, $sup(A_{i-1}) \subseteq sup(R_i)$ and $sup'(R_i) = sup(A_i)$. This allows the use of $expand$ in lines 18,22 and apply in lines 21,25.

In stage II, the forward search for a path through the A_i's is done using partial assignments σ_m, ..., σ_n instead of states (which are full assignments). Every partial assignment σ_i represents a set of states which differ only on variables not in the support of A_i. When moving from σ_{i-1} to σ_i, in lines 21 and 25, we use the same partial transition relation R_i that was used to create A_{i-1} from A_i (see figure 2). After applying R_i, we expand the result to the support of R_{i+1} (line 22) so that we can apply R_{i+1} in the next iteration of the loop. The output of the algorithm is the initial state s_{init} and the inputs $in_m, \ldots, in_{n-1}$. The inputs calculated by the algorithm do not necessarily give values to all input variables in I. When giving inputs to a simulation we need to decide on values for all the input variables. We therefore expand every in_i to I by choosing arbitrary values for the extra input variables.

The following theorems state the correctness of the dynamic algorithm. Their proofs are omitted for brevity, but can be found in the full paper.

Theorem 2.1: Assume the test generation algorithm returns "false". Then no computation of D produces $t_0, \ldots, t_i$.

Theorem 2.2: Assume the test generation algorithm returns an initial state s_{init} and a sequence of inputs $in_m, \ldots, in_{n-1}$. Let $s_m, \ldots, s_0, \ldots, s_n$ be the computation generated when running a simulation using s_{init} and $in_m, \ldots, in_{n-1}$ as inputs. Then, $\forall 0 \leq i \leq n$: $s_i \in t_i$.

2.3 BDD implementation

The algorithm can easily be implemented using BDD representations. The components $Init$, S and $N_1, \ldots, N_l$ of the model are represented using BDDs in the usual manner. In addition, the sets A_j computed by our algorithm are represented by BDDs. The input to the algorithm is a sequence of binary vectors over T. It is straight forward to translate each vector into a BDD that represents the set t_i needed for the algorithm. Most BDD libraries include a function to compute sup and sup', which are simply the sets of current or next state variables that appear in the BDD. All other operations used in our algorithm are standard BDD operations.

A BDD implementation of the algorithm will mostly benefit from the use of partial transition relations. The size of a BDD representing a set A is generally related to the size of $sup(A)$. In many cases, each N_i will not depend on all the variables in $I \cup V$. Thus, taking fewer N_i's will result in a smaller support for the partial transition relation $\bigwedge_{v_i \in sup(A)}[N_i(s) = v_i']$. As a result, the

BDDs computed at intermediate stages in the computation of $Pred$ using the partial transition relation will depend on less variables and therefore will often be smaller.

3 Symbolic Model Checking

3.1 Model checking using dynamic transition relations

Model checking is the problem of finding the set of states in a finite-state machine where a given temporal formula is true. One of the most efficient approaches for solving this problem is a *symbolic* algorithm that uses BDDs [2] for representing the transition relation. This representation is often very concise, allowing the verification of large and complex systems. Verification is performed by an efficient search algorithm. The core of the algorithm is the temporal operator **EX**, which given a model M and a set of states A, both represented by BDDs, determines the set of predecessors of states in A.

The **EX** operator is part of the temporal logic CTL (Computation Tree Logic [8]), frequently used in model checkers. Formulas in CTL express properties of states in the system. They are built from atomic propositions, boolean connectives and *temporal operators*. Each operator consists of two parts: a path quantifier followed by a temporal modality. Path quantifiers indicate that the property should be true of *all* paths starting on a given state (**A**), or *some* path from a given state (**E**). Temporal modalities describe how events should be ordered with respect to time along a path specified by the path quantifier. For example, **X** p means that p holds in the next state; **F** p means that p holds in some future state; **G** p means that p holds in all states on the path. Some common CTL operators are: **AG** p which is true at a state if p is globally true in all paths from the current state, i.e., p is an *invariant*; **EF** p, which is true if p holds sometime in the future for some path, i.e., p is *reachable*.

We demonstrate how the symbolic model checking algorithm works by showing the algorithm for **EF** p. The algorithm for the other temporal operators are similar in nature. The object is to construct a BDD that represents the set of all states that satisfy **EF** p. The algorithm starts with the BDD A for the states that satisfy p. At each step, we add to A the set of its predecessors which is computed by **EX** A. The algorithm terminates when no more states can be added to A.

The most expensive computation step in the above algorithm is the application of the **EX** operator. We notice that the **EX** operator is exactly the $Pred$ operator that was defined in the previous section. We therefore replace the computation of **EX** by the operator $Pred_D$ that uses a partial transition relation. Like before, we compute the partial transition relation *dynamically* according to the set to which **EX** is applied. As before, we assume that each N_i is defined over a small number of (unprimed) variables. Thus, by referring to the smallest number of N_i's, the resulting set of states will depend on small number of variable.

The same treatment also handles model checking for Fair-CTL which is the logic CTL, extended with fairness constraints that restrict the set of paths in the model which are required to satisfy a given formula [8, 3].

4 Implementation and Experimental Results

We have implemented the dynamic transition relations method in SMV [13]. We have tested our method on models that have already been verified by SMV such as the PCI local bus [4]. In addition, we have applied our method to several large Intel circuits. The table below summarizes the results obtained for the following examples:

- #1. A simplified cache coherence protocol derived from the PCI Local bus;
- #2. The model of the PCI Local bus discussed in [4];
- #3 to #10 Circuits from Intel.

In the first two examples we have applied model checking, while in the other ones we have computed test sequences. Some examples have more than one result because they have been run with different parameters such as different length of test sequences, different width of test vectors and different inputs to the sub-circuit.

Times are given in seconds (unless otherwise stated) and measures of space are given in megabytes. The "Dyn." columns relate to the dynamic transition relations algorithms, while the "St." columns relate to the static algorithm using all of the circuit. The "Vec." column is the sub-circuit input vector width. The "Seq." column shows the length of the test sequence for the sub-circuit. The columns for number of variables used and total number of variables represent respectively the maximum number of variables in the support sets of the set of states considered at each iteration and the total number of variables in the circuit. The last column indicates if the algorithm has determined a test sequence (Successful) or if such a sequence does not exist (Fail).

We have run 19 different tests, eleven of which were successful in determining test sequences, six that reported that the test sequences for the sub-circuits were unreachable and two model checking examples. Twelve examples finished executing in both the dynamic and static algorithms. On average the dynamic transition relations algorithm had to consider only about 34% of the variables in the circuit description.

As can be seen from these results, the new algorithms are better suited for large examples. In the smallest ones (the first three), the gains were not impressive. This happens because it is necessary to reconstruct the transition relation at each iteration, which in small examples may take more time and space (due to intermediate computations) than simply creating a transition relation for the complete circuit. On large examples, however, the gains are significant, as in example number four. The new algorithm ran in 1/65 of the time taken by the static one, while using only 1/238 of the memory.

Example number five, however, is puzzling, since its gains are clearly below the average for the medium and large examples. Even by saving significantly on the number of variables (17 out of 90) we still need more space (even though less time) to complete the test sequence generation. This indicates that the method can be more efficient for some types of circuits than others. Future work includes the characterization of which are the "good" circuits and which are the "bad" ones.

A better indication of the behavior of dynamic transition relations can be seen in the examples that only finished with the dynamic algorithm. They executed on average for 15.8 seconds using only 5.1M of memory. During experiments we killed a process whenever it used all of the memory available, 800M. From this we can conclude that in these examples the dynamic algorithm ran in *less than* 1/227 (1 hour / 15.8 seconds) of the time using *less than* 1/150 of the memory (800/5.1) of the original algorithm.

These results show that the dynamic transition relations method can provide significant gains in verification time and space, in some cases up to two orders of magnitude. As expected, it does not work in the same way for all kinds of circuits, but our experiments seem to indicate that it works extremely well for several types of circuits that are used in industry today.

#	Time		Space		Vec.	Seq.	# Variables		S/F
	Dyn.	St.	Dyn.	St.			used	total	
1	56	115	.29	.25			26	33	
2	161	515	.35	.15			26	31	
3	1.5	1.4	1.5	1	4	25	9	36	S
4	90	4550	2.5	598	8	15	18	113	S
	61	3947	2.5	598	8	10	18	113	S
	73	2301	2.5	595	8	5	18	113	S
5	2.1	49	1.8	1.9	8	1	9	90	S
	25	51	2.4	1.9	8	3	17	90	S
	27	51	2.4	1.9	8	5	17	90	S
	35	51	2.4	1.9	8	10	17	90	S
6	2.4	> 2h	3.7	>800	3	8	6	366	F
7	23	> 1h	5.9	>800	20	5	69	184	F
	7.6	> 1h	3.2	>800	15	5	19	184	F
	6.7	> 1h	3.2	>800	10	5	51	184	F
8	.73	14	1.7	18	8	5	10	47	S
	.44	16	1.5	18	8	5	10	47	F
9	18	> 1h	7.3	>800	8	5	70	96	F
	32	> 1h	7.3	>800	8	5	70	96	S
10	22	> 1h	5.5	>800	4	5	76	102	S

Fig. 3. Experimental Results

5 Conclusions

We have presented two major results in this paper. The first is an efficient test-generation algorithm which enables designers to translate test sequences they have created for small sub-circuits into test sequences for the full design. This algorithm guarantees that a test sequence will be found if it exists. The second is the introduction of dynamic transition relations which enhance the operation of finding a set of predecessors. This operation is the source of state-explosion in both our test-generation algorithm and symbolic model checking algorithms and any improvement in its efficiency can prove very useful.

We have implemented the new method in SMV and run it on several large circuits. Our experiments indicate that the new method can provide gains in time and space during verification of up to two orders of magnitude. These results show that dynamic transition relations can make it possible to verify circuits that were previously unmanageable due to their size and complexity.

This work can be extended in several ways. One inefficiency of the method is that it recreates the transition relation at each iteration. However, there may be cases in which the same transition relation has already been constructed. An optimization consists of implementing an algorithm that reuses previously used transition relations.

Another extension is to apply the idea of dynamic transition relations to other operators besides *Pred*. In symbolic model checking the *Pred* operator is called **EX**, and its dual operator is **AX**. Given a set A, the operator **AX** A computes the set of states for which *all* successors are in A. It is easy to show that the same idea can be applied to this operator.

Another direction we plan to pursue is to deal with an asynchronous model of computation. In this model the verified system is either an asynchronous hardware design or a distributed program, and is represented by a set of processes. At each step one of the processes is chosen and executed. The transition relation of such a model is a *disjunction* of relations, rather than a conjunction as in the synchronous case. We plan to adjust both the test-generation algorithm and the dynamic version of *Pred* to this computational model.

A forward search algorithm can also be defined using dynamic transition relations. As before, we determine which transition relations to use according to the support of the set on which we are operating. This time, however, we choose the N_i's according to their current state support, and not the next state support. Unfortunately, in this case we may take into account states which are not necessarily reachable, because a transition in N_i may conflict with another transition in N_j and therefore not exist in the global model. However, if variable v_i is in the support set of the states under consideration but not v_j the dynamic transition relations algorithm would allow the transition of N_i to be taken, identifying unreachable states as reachable. However, in model checking the set of reachable states can be used to simplify the model before verification (e.g. only traverse reachable states), and in this case an upper approximation does not introduce errors, but can speed up verification.

References

1. V. Boppana, S. Rajan, K. Takayama, and M. Fujita. Model checking based on sequential ATPG. In *Computer Aided Verification*, 1999.
2. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
3. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
4. S. Campos, E. Clarke, W. Marrero, and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In *International Conference on Computer Design*, 1995.
5. S. Campos, E. M. Clarke, and O. Grumberg. Selective quantitative analysis and interval model checking: verifying different facets of a system. In *Computer Aided Verification*, 1996. To appear in Formal Methods in System Design.
6. K. Chakrabarty, B. Myrray, and V. Iyengar. Built-in test pattern generation for high-performance circuits using twisted-ring counters. In *IEEE VLSI Test Symposium*, 1999.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
8. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
9. A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *TACAS*, 1997.
10. F. Beenker et. al. A testability strategy for silicon computers. In *Proc. Int. Test Conf.*, 1989.
11. J. Lee and J. H. Patel. An architectural level test generator for a hierarchical design environment. In *Proc. 21st Fault-Tolerant Computing Symp.*, 1991.
12. J. Lee and J. H. Patel. An architectural level test generator for a hierarchical design environment based on nonlinear equation solving. In *Journal of Electronic Testing*, 1993.
13. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
14. Brian Murray. *Hierarchical Testing using Precomputed Tests for Modules*. PhD thesis, Computer Science and Engineering, University of Michigan, 1994.
15. C-C su and C. R. Kime. Multiple path sensitization for hierarchical circuit testing. In *Proc. Int. Test Conf.*, 1990.