Efficient Symbolic Traversal Algorithms using Partitioned Transition Relations*

Zmago Brezočnik, Aleš Časar, Tatjana Kapus
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova 17, SI–2000 Maribor, Slovenia
{brezocnik,casar,kapus}@uni-mb.si

Abstract

This paper presents an efficient tool for symbolic state space traversal of finite state machines. Both algorithms for searching reachable states and for model checking in CTL owe their efficiency primarily to an improved state variables substitution algorithm and to the use of partitioned transition relations. Partitioning of the relations is fully automatic. We give experimental results on the performance of the algorithms applied to a set of synchronous benchmark circuits.

1 Introduction

Searching reachable states of a finite state machine (FSM) is a computation that is needed for several problems in digital circuit design: verification, automatic test pattern generation, state transition graph minimization, finding minimal reset sequencies, etc. Symbolic state space traversal techniques [5] show their superiority over the enumeration-based methods. Sets of states are represented by characteristic functions, and because they are boolean, we can represent them compactly by BDDs. Transitions between states can be represented either by transition functions or transition relations. We choose the latter, because relations are more expressive. Transition relations can be considered as boolean functions and thus efficiently represented by BDDs. The transition relation of a large circuit may result in a huge BDD that either does not fit into computer memory or leads to unacceptably long computation times. There are several approaches to overcome this problem: the use of range restrictor "constrain" [5], the use of generalized cofactor and image restrictor [3], partitioning techniques [2, 10], introducing auxiliary variables, etc. In this paper we describe the use of partitioned transition relations for an efficient symbolic state space traversal and model checking in CTL. We also present an algorithm for substitution of state variables that further improves the efficiency of the tools.

In Section 2 we briefly review BDDs and show how to represent FSMs with them. In Section 3 we describe the methods of searching reachable states. The stress is given on the relational product computation based on partitioned transition relations and on the improved state variables substitution algorithm. Algorithms for symbolic model checking in CTL with partitioned transition relation are described in Section 4. Experimental results are given for benchmark circuits. We conclude with some discussion and plans for further work.

2 Preliminaries

Binary decision diagrams (BDDs) are a canonical representation for boolean functions, more compact than traditional normal forms [1]. Their size is closely related to the variable ordering. Manipulating boolean functions represented by BDDs is very efficient. Hence, BDDs have become widely used in various CAD applications, including symbolic state space traversal and model checking.

BDDs can be used also for representing and manipulating sets if we represent sets by means of their characteristic functions. If $\mathcal{A}$ is a subset of $\{0,1\}^n$, its characteristic function $\chi_{\mathcal{A}} : \{0,1\}^n \rightarrow \{0,1\}$ is defined by

$$\chi_{\mathcal{A}}(a) = \begin{cases} 1 & ; \quad a \in \mathcal{A} \\ 0 & ; \quad a \notin \mathcal{A} \end{cases} \tag{1}$$

Thus, set operations can be performed by boolean operations over the corresponding characteristic functions (e.g., $\chi_{\mathcal{A}^c} = \overline{\chi_{\mathcal{A}}}$, $\chi_{\mathcal{A}\cup\mathcal{B}} = \chi_{\mathcal{A}} + \chi_{\mathcal{B}}$, $\chi_{\mathcal{A}\cap\mathcal{B}} = \chi_{\mathcal{A}} \cdot \chi_{\mathcal{B}}$). For the sake of shorter notation we will denote $\chi_{\mathcal{A}}$ by A in the rest of the paper. Since the characteristic functions are boolean, we can efficiently manipulate them with BDDs.

*Work supported in part by Ministry of Science and Technology, Republic of Slovenia.

A deterministic *finite state machine* (FSM) M is a sixtuple $M = (\Sigma, \mathcal{S}, \mathcal{O}, \delta, \lambda, s_0)$, where Σ is a finite set of input symbols, $\mathcal{S}$ a finite set of states, $\mathcal{O}$ a finite set of output symbols, $\delta : \mathcal{S} \times \Sigma \rightarrow \mathcal{S}$ a state transition function, $\lambda : \mathcal{S} \times \Sigma \rightarrow \mathcal{O}$ an output function, and $s_0 \in \mathcal{S}$ an initial state.

If we want to realize a FSM by a digital circuit, we will have to encode the sets $\mathcal{S}$, Σ, and $\mathcal{O}$ by binary symbols (e.g., 0 and 1). States are encoded by state variables. At least $n = \lceil \log_2 |\mathcal{S}| \rceil$ state variables, $m = \lceil \log_2 |\Sigma| \rceil$ input variables, and $l = \lceil \log_2 |\mathcal{O}| \rceil$ output variables of the circuit are needed. Let $\mathcal{Y}$, $\mathcal{X}$, and $\mathcal{Z}$ represent the set of state variables, the set of input variables, and the set of output variables, respectively.

Once the states and the input symbols of the circuit are encoded, we denote a state transition function as $\delta : \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$. Then, next state variables are functions of present state variables and input variables. We denote next state variables by an added prime ($'$) and write a transition function of a state variable y_i as

$$y_i' = \delta_i(y_0, y_1, \ldots, y_{n-1}, x_0, x_1, \ldots, x_{m-1}) \; , \tag{2}$$

for $i = 0, 1, \ldots, n-1$. We rather introduce transition relations

$$T_i = y_i' \Leftrightarrow \delta_i(y_0, y_1, \ldots, y_{n-1}, x_0, x_1, \ldots, x_{m-1}) \; . \tag{3}$$

Namely, relations have much greater expressive power as functions. With relations it is very easy to handle nondeterminisms and to perform backward reachability search that is relatively difficult with functions. Transition relations T_i can be combined by taking their conjunction to form the transition relation[1] of a FSM:

$$T = T_0 \cdot T_1 \cdot \ldots \cdot T_{n-1} \; . \tag{4}$$

If we insert (3) for each T_i in (4), the transition relation T can be written as

$$T = \prod_{i=0}^{n-1} (y_i' \Leftrightarrow \delta_i(y_0, y_1, \ldots, y_{n-1}, x_0, x_1, \ldots, x_{m-1})) \; . \tag{5}$$

3 Searching Reachable States of FSM

Many methods in symbolic verification, including model checking, face the problem of computing reachable state sets. Therefore, let us show how we search reachable states of a FSM. Let $\mathcal{S}_i$ denote a set of states reachable in at most i steps. $\mathcal{S}_0$ represents a set of initial states. In our case we have $\mathcal{S}_0 = \{s_0\}$. A set of states reachable in at most one step is given by

$$\mathcal{S}_1 = \mathcal{S}_0 \cup \{s' \mid \exists a \exists s [a \in \Sigma \wedge s \in \mathcal{S}_0 \wedge (s, a, s') \in T]\} \; . \tag{6}$$

After finding states reachable in at most one step we search for those reachable in at most two steps. In general, a set of states reachable in at most i steps is represented by

$$\mathcal{S}_i = \mathcal{S}_{i-1} \cup \{s' \mid \exists a \exists s [a \in \Sigma \wedge s \in \mathcal{S}_{i-1} \wedge (s, a, s') \in T]\} \; . \tag{7}$$

We continue with this procedure until in step k no new state is reached. In any case, this happens sooner or later, because we deal with FSMs, where the set of states $\mathcal{S}$ is finite. Then, $\mathcal{S}_k = \mathcal{S}_{k-1}$ is a set of all reachable states.

Introducing the characteristic functions for the sets (e.g., the $\mathcal{S}_i$ is represented by its characteristic function $S_i = \chi_{\mathcal{S}_i}$), we can rewrite (7) into the following logical expression:

$$S_i' = S_{i-1}' + \exists_{v \in \mathcal{Y} \cup \mathcal{X}} (S_{i-1} \cdot T) \; . \tag{8}$$

The function S_i' depends on next state variables y_j'. To obtain S_i we have to replace each y_j' with the corresponding y_j. Therewith, next states are considered to be new present states for the next step. The procedure terminates in step k, when $S_k = S_{k-1}$.

3.1 Relational Product Computation

The most time and space consuming operation in (8) is computing

$$\exists_{y \in \mathcal{Y} \cup \mathcal{X}} (S_{i-1} \cdot T) \; . \tag{9}$$

We call it a *relational product* computation [2]. Relational products can be computed using the normal BDD algorithms for boolean connectives and the existential quantification

$$\exists v[f] = f|_{v=0} + f|_{v=1} \; , \tag{10}$$

where v is a boolean variable to be existentially quantified and f is a formula. In order to increase the efficiency of the computing, we rather use a much more efficient algorithm similar to the one described in [2]. However, if the possible transitions of a FSM are represented by a so-called *monolithic transition relation*, i.e., using a single BDD to represent T, even these algorithms might not be efficient enough if the size of T is large. In order to avoid this problem, we do not build a single BDD for the whole transition relation T, but rather partition T in some groups of transition relations of individual state variables and

[1] Many authors (e.g., [3]) understand the transition relation T of a whole FSM as $T = \exists_{x \in \mathcal{X}} T_0 \cdot T_1 \cdot \ldots \cdot T_{n-1}$.

represent each group by a (smaller) BDD. T is then called a *partitioned transition relation*.

Formula (8) can be written in an extended form as

$$\begin{aligned} S'_i \;=\;& S'_{i-1} + \exists x_0 \exists x_1 \ldots \exists x_{m-1} \exists y_0 \exists y_1 \ldots \\ & \ldots \exists y_{n-1} (S_{i-1} \cdot T_0 \cdot T_1 \cdot \ldots \cdot T_{n-1}) \;. \end{aligned} \tag{11}$$

Although existential quantification does not distribute over conjunction, conjuncts can be moved out of the scope of an existential quantification if they do not depend on any of the variables being quantified. FSMs that represent circuit behaviour exhibit locality, so it is very common that many of the T_j depend on only small number of the input and state variables. We developed a heuristic algorithm for partitioning and reordering of functions and variables for existential quantification in (11) that allows us to apply some existential quantifiers before the BDD of the whole transition relation has been built. The function S_{i-1} in (11) represents a set of present states in step $i-1$. Therefore, it may depend on all state variables y_j, so we leave it within the scope of all existential quantifiers of state variables y_j. In contrast to the function S_{i-1}, functions T_j are constant throughout the computation and at the beginning we can determine their place with respect to the existential quantifiers in (11) once and for all.

The dependencies of functions T_j on individual input and state variables are represented by matrix $\boldsymbol{A}$ in Figure 1. Matrix $\boldsymbol{A}$ is of size $(m+n)\times n$ and $a_{i,j}$ has value 1 if and only if the function T_j depends on the variable that points to row i from the right side, and value 0 otherwise. First, we fill the matrix with 0s and 1s. Then we choose one of the rows (there might be several) with the minimal number of 1s and thus one of the variables that the least number of the functions depend on. We put into the scope of its existential quantifier all the functions that depend on this variable. We will be no more concerned with these functions during the partitioning procedure. Therefore, the selected row and all the columns whose functions were taken are no more needed and we delete them. The whole procedure is then repeated over the reduced matrix as long as we do not run out of functions and variables.

A group of functions which find themselves together in one step of this procedure is called a *block*. We denote it T_{ρ_i} if we got it in the i-th step and ρ_i represents the i-th block of partition ρ over the set $\{0, 1, \ldots, n-1\}$. Accordingly, the block T_{ρ_i} can be written as $T_{\rho_i} = \prod_{j \in \rho_i} T_j$. Let us denote a set of variables whose existential quantifiers are immediately in front of block T_{ρ_i} by $\mathcal{Y}_{\rho_i}$, where

$$\begin{aligned} \mathcal{Y}_{\rho_i} = \{v \mid\;& v \in \mathcal{Y} \cup \mathcal{X} \wedge \\ & \forall j \in \rho_i : \; T_j \text{ depends on } v \wedge \\ & \forall j \notin \rho_i : \; T_j \text{ does not depend on } v\} \;. \end{aligned} \tag{12}$$

Thus, using partitioned transition relation we got the following formula for a general step in computing reachable states:

$$\begin{aligned} S'_i \;=\;& S'_{i-1} + \exists_{v \in \mathcal{Y}_{\rho_{k-1}}} (T_{\rho_{k-1}} \cdot \exists_{v \in \mathcal{Y}_{\rho_{k-2}}} \\ & (T_{\rho_{k-2}} \cdot \ldots \cdot \exists_{v \in \mathcal{Y}_{\rho_0}} (T_{\rho_0} \cdot S_{i-1}) \ldots)) \end{aligned} \tag{13}$$

The algorithm based on (13) has proved itself very well. Just in few cases it was considerably worse (see the example of an up/down counter in Section 5) than the method with monolithic transition relation. However, in many cases it has been shown to be better for many orders of magnitude.

3.2 State variables substitution algorithm

The characteristic function S'_i of a set of reachable states obtained in step i of the symbolic state traversal algorithm depends on next state variables y'_j. Before the step $i+1$ we have to perform a substitution of each next state variable y'_j with the corresponding present state variable y_j. A common algorithm to accomplish the substitution of a variable w for a variable v in a formula f, denoted $f_{v \leftarrow w}$ is

$$f_{v \leftarrow w} = \exists v [(v \Leftrightarrow w) \cdot f]. \tag{14}$$

Using (14) one can get S_i from S'_i as follows:

$$\begin{aligned} S_i \;=\;& \exists y'_0 \exists y'_1 \ldots \exists y'_{n-1} [(y'_0 \Leftrightarrow y_0) \cdot \\ & \cdot (y'_1 \Leftrightarrow y_1) \cdot \ldots \cdot (y'_{n-1} \Leftrightarrow y_{n-1}) \cdot S'_i] \end{aligned} \tag{15}$$

First we have to build the conjunction within square brackets and then apply existential quantification on all next state variables to the result. Computing S_i from (15) may be very time and space consuming, especially if S_i is large.

Substitution of state variables has to be done after each step of the symbolic state space traversal, so it is important to compute it in an efficient manner. Therefore we use another algorithm, shown in Figure 2. The algorithm is realized by a recursive *ite*-based function *Substitute*. It takes the function (BDD) f in which we want to substitute variables and returns a function (BDD) h with the substituted variables. Trivial cases, where $f = 0$ or $f = 1$, are solved immediately. For non-trivial cases the *Substitute* function recurs on both 'else' (h_0) and 'then' (h_1) successors of f and finally combines the results. If the current

$$
\boldsymbol{A} = \begin{array}{c@{}c}
\begin{bmatrix}
a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\
a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \\
a_{n,0} & a_{n,1} & \cdots & a_{n,n-1} \\
a_{n+1,0} & a_{n+1,1} & \cdots & a_{n+1,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n+m-1,0} & a_{n+m-1,1} & \cdots & a_{n+m-1,n-1}
\end{bmatrix}
&
\begin{array}{cc}
\leftarrow & y_0 \\
\leftarrow & y_1 \\
\vdots & \vdots \\
\leftarrow & y_{n-1} \\
\leftarrow & x_0 \\
\leftarrow & x_1 \\
\vdots & \vdots \\
\leftarrow & x_{m-1}
\end{array}
\\
\begin{array}{cccc}
\uparrow & \uparrow & \cdots & \uparrow \\
T_0 & T_1 & \cdots & T_{n-1}
\end{array} &
\end{array}
$$

Figure 1: Matrix for searching dependencies of functions T_j on individual input and state variables

```
TEdge Substitute(TEdge f)
  {
    if (f == 0 || f == 1)
      return f;
    if (result is in computed table of the function Substitute)
      return result;
    let v be the top variable of the function f;
    h0 = Substitute(f|v=0);
    h1 = Substitute(f|v=1);
    if (v should be substituted)
      h = ite(w, h1, h0);        /* h = w · h1 + w̄ · h0 = w · f|v=1 + w̄ · f|v=0 */
    else
      h = ite(v, h1, h0);        /* h = v · h1 + v̄ · h0 = v · f|v=1 + v̄ · f|v=0 */
    insert h in computed table of the function Substitute;
    return h;
  }
```

Figure 2: State variables substitution algorithm

BDD has a top variable v that has to be substituted by a variable w, the restricted formulas h_0 and h_1 are combined as $ite(w, h_1, h_0)$. Otherwise, they are combined as $ite(v, h_1, h_0)$, thus no substitution of the variables at the current top BDD node takes place. A computed table is used to store already computed results in order to avoid computations that have been done already. The use of the new state variables substitution algorithm instead of a common algorithm based on (15) has dramaticaly improved the efficiency of our symbolic traversal tools.

4 Symbolic Model Checking in CTL

The logic that we use to specify properties of FSMs (sequential circuits) is a propositional temporal logic of branching time, called *Computation Tree Logic*—CTL [4]. In this logic each of the usual future time operators of linear temporal logic (***G***—*globally* or *invariantly*, ***F***—*sometimes in the future*, ***X***—*next time*, and ***U***—*until*) must be immediately preceded by a *path quantifier*. The path quantifier can either be an ***A*** (for all computation paths) or an ***E*** (for some computation path). For example, the operator $\boldsymbol{AG}f$ will hold in a state provided that f holds in all states along all possible computation paths starting from that state, and $\boldsymbol{EF}f$, which will hold in a state provided that there is a computation path from that state to the future state where f holds.

CTL formulas are constructed from *atomic propositions* using boolean connectives and temporal operators. In our case of verifying FSMs, the set of atomic propositions is equal to the set $\mathcal{Y}$ of state variables of the circuit. If y_j is an atomic proposition in $\mathcal{Y}$, then the formula y_j is true in a state s (we say that s satisfies y_j and write $s \models y_j$) if and only if the state variable y_j is true in this state.

Model checking is the problem of determining whether a given CTL formula f is true in a given model (in our case the state transition graph of a FSM). In this section, we present an efficient symbolic model checking algorithm for CTL with partitioned transition relations. The algorithm recurses over the structure of a given CTL formula f and determines in a bottom up fashion in which states of the model the subformulas of f are true. Finally, it returns the BDD (the function) S that represents exactly the set of states $\mathcal{S}$ of the model in which f is true.

After determining the set $\mathcal{S}$ of states that satisfy a given CTL formula f, the algorithm checks whether this set includes all initial states[2] $\mathcal{S}_0$ ($\mathcal{S}_0 \subseteq \mathcal{S}$). In this case the formula f is true in the whole model (FSM) M. In other words, we can say that the model satisfies f and write $M \models f$. Checking the condition $\mathcal{S}_0 \subseteq \mathcal{S}$ is easy in our BDD-based algorithm. It has to be checked whether the boolean expression $\overline{S_0} + S \Leftrightarrow 1$ is a tautology.

The CTL formulas constructed only from the atomic propositions (state variables) and ordinary boolean connectives are handled using the standard algorithms for computing boolean connectives with BDDs. The temporal formulas are treated otherwise.

Although there are eight different CTL operators, it is sufficient to realize only three of them, since the remaining five can be easily expressed with the three realized. We decided to realize the operators $\boldsymbol{EX}f$, $\boldsymbol{E}[f\, \boldsymbol{U}\, g]$ and $\boldsymbol{EG}f$.

First, let us have a look at the formula $\boldsymbol{EX}f$. It is true in a state if and only if that state has at least one immediate successor state, where the formula f is true. In other words, $\boldsymbol{EX}f$ is true in all immediate predecessor states of the states, where the formula f is true. Thus, to get the set of states of the model which satisfy the formula $\boldsymbol{EX}f$, we compute the relational product

$$\boldsymbol{EX}f = \exists_{v \in \mathcal{Y}' \cup \mathcal{X}} (f' \cdot T) \,, \tag{16}$$

where f' represents the function f with each state variable y_i substituted by the corresponding next state variable y_i' and T is a transition relation of a given FSM. The relational product (16) is quite similar to the relational product (9) defined in Section 3 except that in (16) we compute one step in a backward reachability search instead of a forward reachability search.

The formula $\boldsymbol{E}[f\, \boldsymbol{U}\, g]$ is true in a state s if there exists a computation path beginning in s such that g is true in a future state t on the path, and f is true in all states between s and t. This means that either g is true in the current state, or f is true in the current state and there exists a successor state where $\boldsymbol{E}[f\, \boldsymbol{U}\, g]$ is true. Consequently, the BDD that represents the states where $\boldsymbol{E}[f\, \boldsymbol{U}\, g]$ is true can be computed by finding the least fixed point of the predicate transformer F defined by

$$F(S) = g + f \cdot \boldsymbol{EX} S \,. \tag{17}$$

The formula $\boldsymbol{EG}f$ means that there exists a computation path beginning in the current state in which f is globally (invariantly) true. In other words, f is true in the current state, and there exists a successor state where $\boldsymbol{EG}f$ is true. The BDD that represents the states where $\boldsymbol{EG}f$ is true is computed as the greatest fixed point of the predicate transformer

$$F(S) = f \cdot \boldsymbol{EX} S \,. \tag{18}$$

Computing any of the CTL temporal formulas is finally reduced to computing the relational product (16). If we use the monolithic transition relation (i.e., construct one BDD for T), we will face the same problem for large T as already mentioned in computing the relational product in (8). We suggest to introduce the partitioned transition relation as in subsection 3.1. There we partitioned transition relations of state variables into blocks with respect to the dependencies of the relations on present state variables and inputs. In (16) we have to do existential quantification on next state variables and inputs. Therefore, we use another matrix $\boldsymbol{A}$ for the partitioning and reordering of relations T_j in (16), where the first n rows of the matrix correspond to the next state variables y_i' as shown in Figure 3.

The upper part of matrix $\boldsymbol{A}$ has values '1' in the diagonal and the values '0' elsewhere. After we fill the matrix with 0s and 1s, we proceed further the same way as in forward reachability search. Due to the exchanged role of the state variables, we come out with different sets of variables $\mathcal{Y}'_{\rho_i}$ that appear under existential quantifiers immediately before the corresponding blocks T_{ρ_i}:

$$\begin{aligned} \mathcal{Y}'_{\rho_i} = \{v \mid \; & v \in \mathcal{Y}' \cup \mathcal{X} \wedge \\ & \forall j \in \rho_i : \; T_j \text{ depends on } v \wedge \\ & \forall j \notin \rho_i : \; T_j \text{ does not depend on } v\} \,. \end{aligned} \tag{19}$$

Thus, using partitioned transition relation, the relational product (16) becomes

$$\begin{aligned} \boldsymbol{EX}f \;\; = \;\; & \exists_{v \in \mathcal{Y}'_{\rho_{k-1}}} (T_{\rho_{k-1}} \cdot \exists_{v \in \mathcal{Y}'_{\rho_{k-2}}} \\ & (T_{\rho_{k-2}} \cdot \ldots \cdot \exists_{v \in \mathcal{Y}'_{\rho_0}} (T_{\rho_0} \cdot f') \ldots)) \,. \end{aligned} \tag{20}$$

The results show that the symbolic model checking in CTL with partitioned transition relation is much more efficient (except for very small examples) than with a monolithic transition relation.

[2] If there is a single initial state s_0, we have $\mathcal{S}_0 = \{s_0\}$.

$$
\boldsymbol{A} \;=\; \begin{array}{cc}
\left[\begin{array}{cccc}
1 & 0 & \cdots & 0 \\
0 & 1 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 1 \\
a_{n,0} & a_{n,1} & \cdots & a_{n,n-1} \\
a_{n+1,0} & a_{n+1,1} & \cdots & a_{n+1,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n+m-1,0} & a_{n+m-1,1} & \cdots & a_{n+m-1,n-1}
\end{array}\right] &
\begin{array}{cc}
\leftarrow & y'_0 \\
\leftarrow & y'_1 \\
\vdots & \vdots \\
\leftarrow & y'_{n-1} \\
\leftarrow & x_0 \\
\leftarrow & x_1 \\
\vdots & \vdots \\
\leftarrow & x_{m-1}
\end{array} \\
\begin{array}{cccc}
\uparrow & \uparrow & \cdots & \uparrow \\
T_0 & T_1 & \cdots & T_{n-1}
\end{array} &
\end{array}
$$

Figure 3: Matrix for searching dependencies of functions T_j on individual input and next state variables

5 Experimental results

Experiments have been done on a 113.5 SPECint92 Silicon Graphics Indy workstation with Mips R4600SC/133MHz processor, 64 MB of physical memory and 124 MB of virtual memory under IRIX 5.3 operating system. Results marked with asterix (*) are from 115.1 SPECint92 HP 715/100 workstation with 128 MB of physical memory and 391 MB of virtual memory under HP-UX 10.01 operating system. We have used our own BDD package [6] that is an efficient *ite*-based implementation of reduced ordered binary decision diagrams with complemented edges.

Table 1 shows the results obtained by symbolic searching of reachable states on the ISCAS'89 circuits and on the parametric up/down counters. From left to right the columns in Table 1 refer to the circuit name, the number of reachable states, the number of steps needed to compute all reachable states, the maximal number of BDD nodes whenever generated during the symbolic state space traversal of a given circuit together with the CPU time in seconds both for the search using monolithic transition relation and partitioned transition relation.

We compared our results on ISCAS'89 circuits with the results published in [5] and [3]. In [5] the behaviour of a FSM is represented by a transition function while the symbolic computation of reachable states is based on the strict range restrictor *constrain*. As in our approach, a transition relation represents the behaviour of a FSM in [3], but the relational product computation uses a so-called *generalized cofactor and image restrictor*. Although the CPU times are difficult to compare directly because of different platforms being used, it seems that our symbolic state space traversal program performs much better on these benchmarks.

The parametric n-bit counter counts up(down) modulo 2^n if a control input *UP* equals '1'('0'). Its state transition graph is very regular having a topology of a double ring with 2^n states. All of them are reachable in 2^{n-1} steps. In each step of a symbolic state space traversal only two new states are explored. This means that symbolic state space traversal only halves the number of steps needed for a non-symbolic traversal. Therefore, the number of BDD nodes and CPU times grow exponentially with n. Surprisingly, much better results were achieved using monolithic transition relation. In this case, combining all T_i together into one BDD results in a smaller representation than using partitioned transition relation where each T_i falls in a single block. Smaller representation speeds up the computation of relational products.

Great power of a symbolic state space traversal is demonstrated in the next example. It is a parametric arbiter with n inputs taken from [10]. The arbiter controls the access to the bus by requesters with a decreasing priority. Request 0 has the highest and request $n-1$ the lowest priority. An instance of such an arbiter for $n = 3$ is shown in Figure 4.

Results on computing reachable states for the parametric arbiter using partitioned transition relation are shown in Table 2. The number of reachable states is written in an exponential notation with 16 valid digits after a decimal point. For each n only 3 steps of symbolic state space traversal are required to find all reachable states of the arbiter. Although the number of reachable states grows with n extremly fast, the number of BDD nodes (Figure 5) and CPU time grow much slower. The number of reachable states of a n-input arbiter can be determined analitically as $2^n(n+1)$, because input latches (see Figure 4) may hold any boolean value (2^n combinations) and at most one output latch may hold boolean value 1 ($n+1$ combinations). Our algorithm was able to cope arbiters up to $n = 850$ in a reasonable time.

We verified the behaviour of a parametric arbiter

Table 1: Computing reachable states for ISCAS'89 circuits and a parametric up/down counter

circuit	# reachable states	# steps	monolithic TR # BDD nodes	monolithic TR CPU time [s]	partitioned TR # BDD nodes	partitioned TR CPU time [s]
s27	6	3	87	0.02	72	0.03
s208.1	256	255	2390	0.07	5568	0.15
s298	218	19	3991	0.08	3024	0.09
s344	2625	7	11966	0.33	19462	0.47
s349	2625	7	11966	0.33	19462	0.48
s382	8865	151	43324	4.24	66182	1.77
s386	13	8	953	0.05	928	0.05
s444	8865	151	13660	0.31	20863	0.48
s510	47	47	1611	0.07	1633	0.07
s526	8868	151	19610	0.41	36560	0.78
s526n	8868	151	19608	0.41	36558	0.78
s641	1544	7	1299567	360.85	23649	0.83
s713	1544	7	1299622	357.87	23673	0.84
s820	25	11	1543	0.06	1680	0.08
s832	25	11	1568	0.07	1705	0.08
s953	504	11	37187	0.71	11978	0.37
s1196	2616	3	236067	61.80	28788	0.96
s1238	2616	3	236086	62.69	28810	0.96
s1488	48	22	3444	0.10	3444	0.11
s1494	48	22	3452	0.12	3452	0.13
count4	16	8	154	0.02	167	0.03
count8	256	128	2078	0.12	5059	0.13
count12	4096	2048	28558	0.82	112744	2.67
count16	65536	32768	442619	20.00	*2326796	*104.80

with a symbolic model checking in CTL. The properties being verified together with the corresponding CTL formulas are as follows:

- *Exclusivity.* At most one output is set to '1'. In order to decrease the complexity of the CTL formula we weaken the exclusivity and rather write (as in [10]) that either odd number of outputs is set to '1' or all outputs are set to '0'.

$$\boldsymbol{AG}(OUT_0 \oplus OUT_1 \oplus \ldots \oplus OUT_{n-1} + \overline{OUT_0} \cdot \overline{OUT_1} \cdot \ldots \cdot \overline{OUT_{n-1}})$$

- *Causality.* If input i ($i = 0, 1, \ldots, n-1$) requests the bus, the access will eventually be granted to input i.

$$\boldsymbol{AG}(IN_i \Rightarrow \boldsymbol{EF}\, OUT_i)$$

- *Starvation.* There is a possibility that a request from unit i is never serviced. This behaviour will happen if another input being granted the bus never releases its request.

$$\boldsymbol{EG}(IN_i \Rightarrow \boldsymbol{EG}\, \overline{OUT_i})$$

- *Allocation.* If a request from input i ($i = 0, 1, \ldots, n-1$) is loaded and no higher priority unit requests the bus, then the bus is granted to input i at the next state.

$$\boldsymbol{AG}(\overline{OUT_0} \cdot \ldots \cdot \overline{OUT_{n-1}} \cdot IN_0 \Rightarrow \boldsymbol{AX}\, OUT_0)$$

$$\boldsymbol{AG}(\overline{OUT_0} \cdot \ldots \cdot \overline{OUT_{n-1}} \cdot \overline{IN_0} \cdot IN_1 \Rightarrow \boldsymbol{AX}\, OUT_1)$$

$$\vdots$$

$$\boldsymbol{AG}(\overline{OUT_0} \cdot \ldots \cdot \overline{OUT_{n-1}} \cdot \overline{IN_0} \cdot \ldots \cdot \overline{IN_{n-2}} \cdot IN_{n-1} \Rightarrow \boldsymbol{AX}\, OUT_{n-1})$$

In Table 3 we collected the results obtained from the verification of the parametric bus arbiter.[3] The

[3] Although an arbiter with $n = 1$ is a totally senseless circuit, it is included in Table 3 for the sake of completeness. Anyway, model checking algorithm does not care whether it handles useful or senseless circuits.

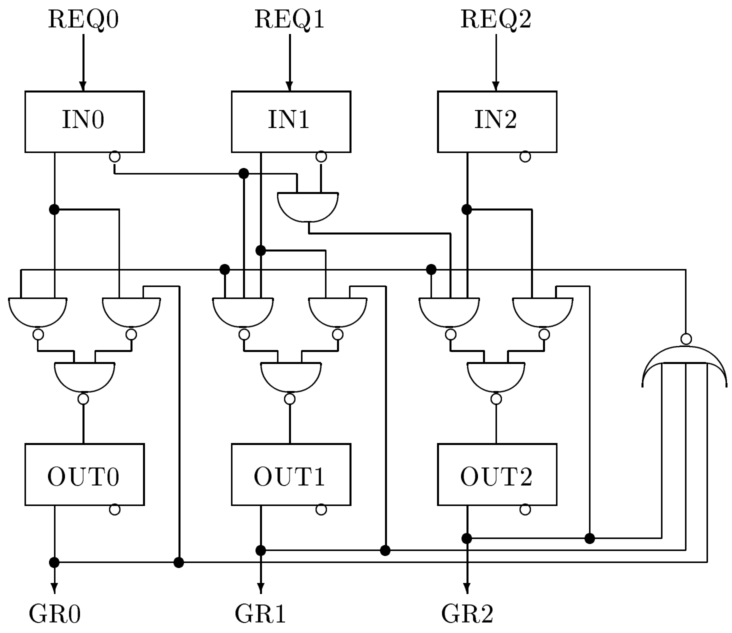

Figure 4: A 3-request bus arbiter

columns from left to right represent the number of inputs (n), the number of state variables (latches) of the circuit ($2n$), the number of CTL formulas being verified for each n ($3n+1$) and the maximal number of BDD nodes whenever generated during symbolic model checking of a given circuit together with the CPU time in seconds both for the model checking using monolithic transition relation and partitioned transition relation. An overflow on BDD nodes is denoted by a dash '—'.

The complexity of model checking grows fast with n because of two reasons: the increase of the model size and the growing number of CTL formulas ($3n+1$) that have to be verified. In addition, the size of each CTL formula expressing allocation property and the formula expressing exclusivity grows with n. Using monolithic transition relation our model checking algorithm succeeded to check arbiters up to $n = 8$. Exploiting partitioned transition relation significantly decreases the number of BDD nodes and therewith the CPU time. The largest verified arbiter was that with 12 inputs. In [10] the CPU time is given only for the 3-input arbiter. Our CPU time for $n = 3$ is much smaller than the one reported in [10].

6 Conclusions

We have reviewed two different algorithms for performing symbolic computation of reachable states in FSMs, using either monolithic transition relation or partitioned transition relation to represent state transition functions of the machines. Introducing partitioned transition relation may decrease the CPU times for large circuits dramaticaly.

We have written an efficient software for searching reachable states of FSMs based on those algorithms [6]. It uses our own fully home made BDD package [9]. Then, we have developed a program package for verifying synchronous sequential circuits by symbolic model checking [7, 8]. Properties to be verified are expressed in CTL.

The main contribution of this paper is that partitioning and reordering of individual transition relations of the circuit in the relational product is automatic and needs no user intervention as in [2]. Experimental results on the ISCAS'89 benchmark circuits show that the algorithm for searching reachable states is more efficient as the algorithms described in [3] that exploit a generalized cofactor and image restrictor and significantly faster that the algorithms described in [5] that use range restrictor "constrain".

Another contribution of this paper is an efficient algorithm for substitution of state variables, which gives much better results than the algorithm normally used.

In this work we have not been concerned much with the problem of finding a good variable ordering. A heuristic ordering where next and present state variables are interleaved and input variables are at the end is used. Variable ordering is a problem orthogonal to the problems discussed. Better variable ordering would improve the results reported here. It would also be interesting to investigate the approach

Table 2: Computing reachable states for a parametric arbiter

# inputs n	# reachable states	# steps	partitioned TR # BDD nodes	partitioned TR CPU time [s]
50	$5.7420895248973824 \cdot 10^{16}$	3	22156	1.03
100	$1.2803271062305117 \cdot 10^{32}$	3	72043	5.54
150	$2.1551440159859994 \cdot 10^{47}$	3	160606	15.60
200	$3.2299454689605705 \cdot 10^{62}$	3	284169	34.06
250	$4.5412209997759945 \cdot 10^{77}$	3	442669	61.13
300	$6.1314782887668031 \cdot 10^{92}$	3	636232	99.93
350	$8.0501801421251510 \cdot 10^{107}$	3	864795	151.91
400	$1.0354822011128503 \cdot 10^{123}$	3	1128358	222.24
450	$1.3112170586292748 \cdot 10^{138}$	3	1426921	313.20
500	$1.6399686945559671 \cdot 10^{153}$	3	1760421	431.87
550	$2.0307161094498723 \cdot 10^{168}$	3	*2128984	*673.95
600	$2.4938588568974768 \cdot 10^{183}$	3	*2532547	*861.33
650	$3.0414324142817783 \cdot 10^{198}$	3	*2971110	*1042.95
700	$3.6873552669854098 \cdot 10^{213}$	3	*3444610	*1256.83
750	$4.4477122776711747 \cdot 10^{228}$	3	*3953173	*1517.78
800	$5.3410795607367633 \cdot 10^{243}$	3	*4496736	*1773.71
850	$6.3888968213127999 \cdot 10^{258}$	3	*5075299	*2085.25

in between the one using monolithic and that one using fully partitioned transition relation, where some blocks of T_is would be combined together and represented by a single BDD.

In addition, we will give more attention to tuning of some data structure parameters (e.g., the size of different tables and caches) that may also influence the efficiency. Future work will consist also in building different parsers in order to accept circuit descriptions in various input formats.

References

[1] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[2] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[3] G. Cabodi and P. Camurati. Advancements in Symbolic Traversal Techniques. In G. J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods*, pages 155–166, Arles, France, May 1993.

[4] 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, April 1986.

[5] O. Coudert and J. C. Madre. Symbolic Computation of the Valid States of a Sequential Machine: Algorithms and Discussion. In *1991 International Workshop on Formal Methods in VLSI Design*, pages 1–19, Miami, Florida, USA, January 1991.

[6] A. Časar and Z. Brezočnik. Symbolic State Space Traversal of Finite State Automata. In F. Solina and B. Zajc, editors, *Proceedings of the Fourth Electrotechnical and Computer Science Conference ERK'95*, pages 85–88, Portorož, Slovenia, September 1995. Slovenia Section IEEE. In Slovene.

[7] A. Časar, Z. Brezočnik, and T. Kapus. Exploiting Partitioned Transition Relations for Efficient Symbolic Model Checking in CTL. In P. Storms, editor, *Proceedings of the European Design & Test Conference ED&TC'96*, pages 606–606, Paris, France, March 1996. IEEE Computer Society Press.

[8] A. Časar, Z. Brezočnik, and T. Kapus. Symbolic Model Checking of Finite State Machines with CTL. In B. Zajc and F. Solina, editors, *Proceedings of the Fifth Electrotechnical and Computer Science Conference ERK'96*, pages 51–54, Portorož, Slovenia, September 1996. Slovenia Section IEEE. In Slovene.

[9] A. Časar and R. Meolic. Representation of Boolean Functions with ROBDDs. To be published in 1993/1994 IEEE Student Papers Book.

[10] D. Déharbe and D. Borrione. Symbolic Model Checking of VHDL Design Entities. Technical report, ARTEMIS, Grenoble, France, November 1993.

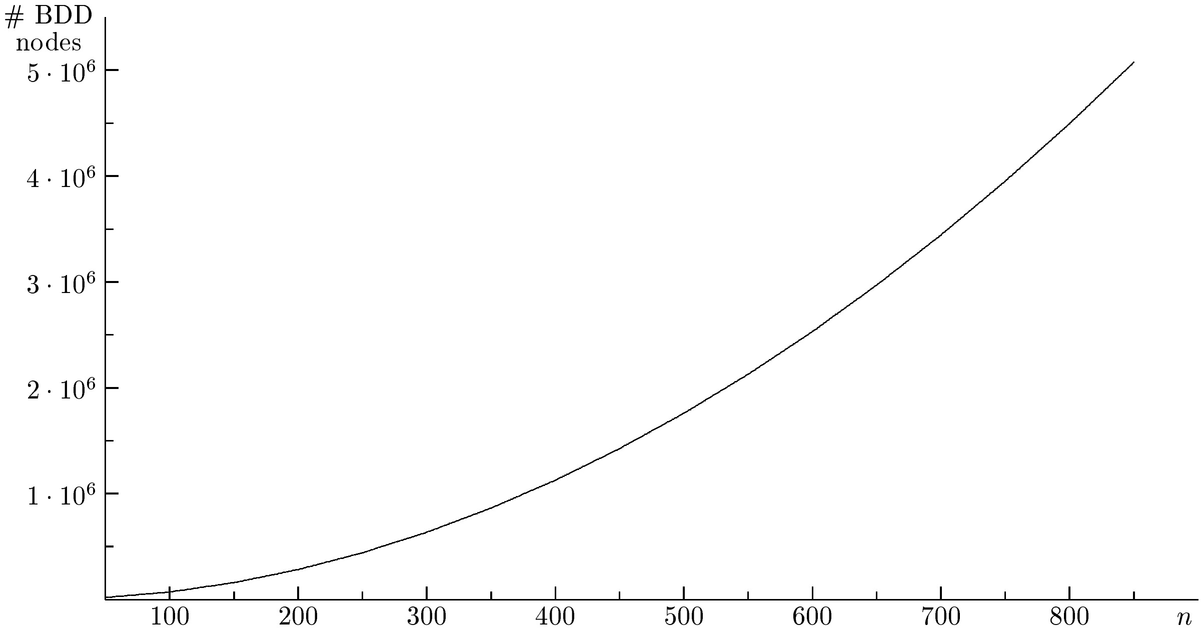

Figure 5: Number of BDD nodes for computing reachable states of n-bit arbiters

Table 3: Checking of parametric arbiter in CTL

# inputs n	# state variables	# CTL formulas	monolithic TR		partitioned TR	
			# BDD nodes	CPU time [s]	# BDD nodes	CPU time [s]
1	2	4	18	0.28	11	0.32
2	4	7	174	0.30	76	0.33
3	6	10	947	0.33	298	0.34
4	8	13	4403	0.54	929	0.36
5	10	16	19171	1.71	2299	0.40
6	12	19	78044	10.57	7540	0.60
7	14	22	297614	80.11	15095	0.87
8	16	25	1214392	641.56	53142	2.50
9	18	28	—	—	91248	4.24
10	20	31	—	—	354418	21.71
11	22	34	—	—	547160	39.66
12	24	37	—	—	*2325970	*402.46