Partial Model Checking with ROBDDs*

Henrik Reif Andersen Jørgen Staunstrup Niels Maretti

Department of Information Technology, Building 344,
Technical University of Denmark, DK–2800 Lyngby, Denmark.

Abstract. This paper introduces a technique for localizing model checking of concurrent state-based systems. The technique, called *partial model checking*, is fully automatic and performs model checking by gradually specializing the specification with respect to the concurrent components one by one, computing a "concurrent weakest precondition." Specifications are invariance properties and the concurrent components are sets of transitions. Both are expressed as predicates represented by Reduced Ordered Binary Decision Diagrams (ROBDDs). The self-reducing properties of ROBDDs are important for the success of the technique.

We describe experimental results obtained on four different examples.

1 Introduction

The major problem with automatic model checking is what has been known as the state-explosion problem: the combinatorial explosion of global states when combining loosely coupled concurrent components. Many techniques to overcome this problem have been proposed. Some techniques require manual assistance in decomposing the verification tasks, others are fully automatic. Examples of the first kind are [5,7,11,16]. Examples of the second kind are partial order methods [10] and Reduced Ordered Binary Decision Diagrams [2] (ROBDDs). Of course, it is to be expected that manual methods are more powerful than fully automatic methods but due to the appealing ease of use, it is of great interest to push the border of automatic verification as far as possible.

The technique described in this paper is of the second, fully automatic, kind and it tries to push the border by improving on the ROBDD-techniques. The central idea is to utilize the structure of the concurrent system in a compositional fashion: the specification is gradually transformed by specializing it with respect to the concurrent components one by one, akin to computing weakest preconditions for sequential programs [8]. Each residual specification provides a partial answer to the original verification problem, hence the name *partial model checking*. This idea has been successfully applied to the event-based model CCS and the specification formalism known as the modal μ-calculus [1]. We apply the same idea to state-based systems with specifications given as state predicates and observe the same positive result although

* Work supported by the Danish Technical Research Council, project CoDesign.
E-mail and WWW addresses of the first two authors: {hra,jst}@it.dtu.dk, http://www.it.dtu.dk/{~hra,~jst}.

the necessary steps turn out to be quite different. First of all, we utilize the compactness of ROBDDs instead of the carefully designed minimization algorithms in [1]. Secondly, the ROBDD version is a variation of a backwards reachability computation whereas the event-based version did not contain any explicit state-space computation.

The new technique is evaluated and compared with more standard techniques by a series of experiments carried out on four examples. The results are clearly in favour of the new technique.

2 Models

Our system model consists of n concurrent components referred to as *processes*, working on a global set of *binary variables* V. States $\mathbb{B}^V$ of the system are total functions assigning a binary value $\mathbb{B} = \{0, 1\}$ to each variable. Each process consists of a set of *transitions* $T_i \subseteq \mathbb{B}^V \times \mathbb{B}^V$, i.e., sets of pairs of states. The system starts in any of a set of *initial states* $I \subseteq \mathbb{B}^V$ and proceeds stepwise by non-deterministically performing one of the transitions of the processes. We shall use $s, t, u \in \mathbb{B}^V$ to range over states, $I, P, B, F, Q \subseteq \mathbb{B}^V$ to range over subsets of states and $R, S, T \subseteq \mathbb{B}^V \times \mathbb{B}^V$ to range over relations. Thus a system is described as a triple

$$(V, \{T_i \mid 1 \leq i \leq n\}, I).$$

This is a very simple model with plenty of room for improvements in various directions. For instance, we could split the variables into sets of *local* variables, each "read" and "written" only by one process, together with a set of *shared* variables. However, the technique makes no explicit use of such information and the simple model suffices to demonstrate it. Of course, we expect that transitions that are somehow related belong to the same process. Moreover, typically there are variables that are read and written by one process only, thereby providing local state for this process. This is certainly the case for all the examples. It turns out that the amount of local state and the simplicity of interdependence between processes has an important impact on the usefulness of partial model checking. The mathematical explanation of the technique is nevertheless completely independent of these parameters.

The verification problem we consider is the following reachability problem: given a property (a set of states) P, will the system when started in one of the initial states always stay in states that belong to P? This problem covers what is often known as *safety* properties.

3 Three Verification Techniques

Before explaining the verification techniques some notation is introduced. To avoid introducing any particular syntax only elementary notions of set

theory are used. Moreover, for uniformity we often consider a subset $P \subseteq \mathbb{B}^V$ as being a relation $P_\bullet \subseteq \mathbb{B}^\emptyset \times \mathbb{B}^V$ between the singleton set $\mathbb{B}^\emptyset = \{\bullet\}$ and $\mathbb{B}^V$ such that a state s belongs to P if and only if $(\bullet, s)$ belongs to $P_\bullet$. Sometimes we shall by a slight abuse of notation refrain from making this distinction explicit.

If $S \subseteq \mathbb{B}^U \times \mathbb{B}^V$ and $T \subseteq \mathbb{B}^V \times \mathbb{B}^W$ are two relations between sets of states we denote their relational composition by

$$T \circ S = \{(s,t) \mid \exists u \in \mathbb{B}^V.\ (s,u) \in S, (u,t) \in T\}.$$

In particular, if P is a set of states and T is a relation between states, then $T \circ P_\bullet$ is the image of P under T, a set of states, represented as a relation. If $T \subseteq \mathbb{B}^V \times \mathbb{B}^V$ is a set of transitions we denote by T^* its *reflexive, transitive closure*:

$$\begin{aligned} T^0 &= \{(s,s) \mid s \in \mathbb{B}^V\} \\ T^{n+1} &= T \circ T^n \\ T^* &= \bigcup_n T^n \end{aligned}$$

The f-technique

Taking $T = T_1 \cup \cdots \cup T_n$ for a system with transition relations T_i, we can now formulate our verification problem formally as determining whether $T^* \circ I_\bullet$ is a subset of $P_\bullet$ or, by the aforementioned slight abuse of notation, whether:

$$T^* \circ I \subseteq P. \tag{1}$$

An obvious way to answer this question is to compute the left-hand side and check whether the set inclusion holds. We can do this by a well-known *forwards fixed-point iteration*, computing F_i's until a fixed point is reached, where

$$\begin{aligned} F_0 &= \emptyset \\ F_{i+1} &= (T \circ F_i) \cup I. \end{aligned}$$

The forward fixed-point iteration can be realized using ROBDDs to represent the set of states and the transitions. We refer to it as the **f**-technique. Section 5 describes experiments with the **f**-technique and compares it with the two other techniques explained below.

The b-technique

The inclusion (1) can be formulated differently by the use of an operator $\multimap$. Assume $T \subseteq \mathbb{B}^W \times \mathbb{B}^V$ and $S \subseteq \mathbb{B}^U \times \mathbb{B}^V$, then $T \multimap S \subseteq \mathbb{B}^U \times \mathbb{B}^W$ is defined by

$$T \multimap S \;=\; \{(u,w) \in \mathbb{B}^U \times \mathbb{B}^W \mid \forall v \in \mathbb{B}^V.\ (w,v) \in T \;\Rightarrow\; (u,v) \in S\}.$$

The notation $T \multimap S$ represents "the set of states (s,t) for which t through T always leads to states u such that (s,u) is in S." As a special case, for a subset P (i.e., $U = \emptyset$) and relation T^*, $T^* \multimap P$ is the simpler "set of states that through sequences of T-transitions only can lead to states in P." In program verification this is known as the weakest precondition.

From the definitions of $\circ$ and $\multimap$ it is now easy to check the following relationship:

$$T^* \circ I \subseteq P \quad \Leftrightarrow \quad I \subseteq T^* \multimap P$$

More generally, for any relation R it is also easy to deduce that

$$(T \circ R) \multimap S \;=\; R \multimap (T \multimap S), \tag{2}$$

This shows how composition can be replaced by $\multimap$. The verification problem can now be rephrased as

$$I \subseteq T^* \multimap P. \tag{3}$$

We can compute the right-hand side by a *backwards fixed-point iteration*,

$$B_0 = \mathbb{B}^V$$
$$B_{i+1} = (T \multimap B_i) \cap P$$

and check the inclusion (3). We refer to this as the **b**-technique.

Using terminology from fixed-point theory [17], the **f**- and **b**-techniques can be characterized as computation of the least and greatest fixed-points of the functions $(T \circ \_) \cup I$ or $(T \multimap \_) \cap P$, respectively, followed by tests for implication, $\_ \subseteq P$ and $I \subseteq \_$, respectively.

The q-technique

Neither the **f**- nor the **b**-technique make use of the structure of the system. The transitions are treated like a single relation. The *quotienting* technique introduced here, which we shall refer to as the **q**-*technique*, is a refinement of the backwards iteration. The idea is to exploit the (modular) structure of the system by taking one of the processes at a time and work it into the specification giving a new specification. The new specification should hold for the remaining system if and only the original specification holds for the complete system.

It amounts to computing $T^* \multimap P$ without first computing $T = T_1 \cup \cdots \cup T_n$ and use T in a backwards iteration. Instead we shall compute $T_k^* \multimap P$ for some k and proceed with the other T_i's. Since the T_i's can be interchanged freely prior to the quotienting, when explaining the technique k is simply chosen decreasingly from n down to 1. There is an important choice in this numbering when it comes to actual experiments. The choice can greatly influence the efficiency of the verification, as we shall see in section 5. Removing a T_k does not suffice on its own. We need to keep a simplified version of the removed transitions as captured by the following theorem:

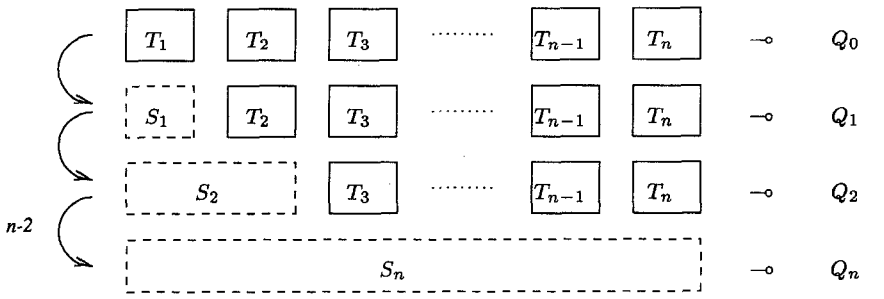

Fig. 1. Sketch of the quotienting technique. After one step T_1 is replaced by S_1 and the original $P = Q_0$ by Q_1. After two steps T_1 and T_2 are replaced by S_2 and Q_1 by Q_2. After $n-2$ further steps all the T_i's are replaced by S_n and Q_0 by Q_n. The final answer is obtained by computing $S_n^* \multimap Q_n$.

Theorem 1 (Quotienting theorem). *Assume $P \subseteq \mathbb{B}^V$ is a set of states and let $T_i \subseteq \mathbb{B}^V \times \mathbb{B}^V$ for $1 \leq i \leq n$ be n sets of transitions with $T = \bigcup_{i=1}^n T_i$. Take $Q_0 = P$ and $S_0 = \emptyset$. Define inductively for $i \in \{0, \ldots, n-1\}$:*

$$Q_{i+1} = (S_i \cup T_{i+1})^* \multimap Q_i$$
$$S_{i+1} = (S_i \cup T_{i+1}) \cap (Q_{i+1} \times Q_{i+1})$$

Then for all $i \in \{0, \ldots, n\}$,

$$T^* \multimap P \;=\; (S_i \cup T_{i+1} \cup \ldots \cup T_n)^* \multimap Q_i \tag{4}$$

The proof is given in appendix A.

We cannot remove T_i completely, but must incorporate a simplified version of it in S_i as indicated in Figure 1. This is needed since our logic for expressing Q_i is not powerful enough to capture the "full effect" of T_i. It was not needed for the event-based version in [1].

As a special application of the theorem we take $i = n$ to obtain

$$T^* \multimap P \;=\; S_n^* \multimap Q_n.$$

The **q**-technique consists of computing iteratively first the Q_i's and S_i's then $S_n^* \multimap Q_n$ and finally check whether $I \subseteq S_n^* \multimap Q_n$ holds.

It is not difficult to see that for all relations R, $R^* \multimap Q_i$ is a subset of Q_i, which implies that $Q_{i+1} \subseteq Q_i$. In fact, each step in the iterative computation of Q_{i+1} employs sets that are included in Q_i. This implies that if the system does *not* satisfy the original property, this could be discovered earlier by checking each time for inclusion of I.

Example. We illustrate the **q**-technique by a small example consisting of two components T_1 and T_2 which should satisfy the property P:

$$(T_1 \cup T_2)^* \multimap P$$

We assume that the only variables are x and y, and use the notation $\|\phi\|$ for the set of states $s \in \mathbb{B}^{\{x,y\}}$ in which the Boolean expression ϕ evaluates to 1. The property is $P = \|\neg x\|$. The transitions are $T_1 = \{(s, s[1/x]) \mid s(y) = 1\}$, which changes x to 1 in states where y is 1, and $T_2 = \{(s, s[0/y]) \mid s(x) = 1\}$, which changes y to 0 when x is 1.

The sequence of quotients is computed as follows. Firstly, $Q_0 = P = \|\neg x\|$ and $S_0 = \emptyset$. Secondly,

$$\begin{aligned} Q_1 &= (S_0 \cup T_1)^* \multimap Q_0 \quad = T_1^* \multimap \|\neg x\| \quad = \|\neg x \wedge \neg y\| \\ S_1 &= (S_0 \cup T_1) \cap Q_1 \times Q_1 = T_1 \cap \|\neg x \wedge \neg y\| \times \|\neg x \wedge \neg y\| = \emptyset. \end{aligned}$$

Thirdly,

$$\begin{aligned} Q_2 &= (S_1 \cup T_2)^* \multimap Q_1 \quad = T_2^* \multimap \|\neg x \wedge \neg y\| \quad = \|\neg x \wedge \neg y\| \\ S_2 &= (S_1 \cup T_2) \cap Q_2 \times Q_2 = T_2 \cap \|\neg x \wedge \neg y\| \times \|\neg x \wedge \neg y\| = \emptyset, \end{aligned}$$

and finally,

$$(T_1 \cup T_2)^* \multimap P \;=\; S_2^* \multimap Q_2 \;=\; \|\neg x \wedge \neg y\|.$$

From this we see that the initial state must belong to $\|\neg x \wedge \neg y\|$. The only satisfactory choice is therefore $s(x) = 0$ and $s(y) = 0$. All other choices would yield a system not satisfying P.

The relation S_1 turns out to be empty since T_1 is not enabled inside the domain of interest, $Q_1 \times Q_1$. However, if we instead use the transition relation $T_1' = T_1 \cup \{(s, s[0/x]) \mid s(y) = 0\}$ we get $S_1 = S_2 = \{(s, s[0/x]) \mid s(y) = 0\}$. The result nevertheless remains the same: $\|\neg x \wedge \neg y\|$.

We believe that major improvements to the technique should go through simplifying the S_i's in the iteration. They represent a simplified version of the T_i's that have been quotiented out. We see no way of avoiding them in general apart from manual assistance by guessing invariants and eliminating iterations altogether. It is a subject for future work to determine situations where the S_i's could be left out or simplified.

4 ROBDD Implementation

Having explained the mathematics behind the tree verification techniques we now focus on the implementation of the techniques using Reduced Ordered Binary Decision Diagrams [2]. Section 5 describes experiments carried out with this implementation.

The computations are implemented as ROBDD-operations. Sets of states $P \subseteq \mathbb{B}^V$ are represented by their characteristic Boolean function f_P from $\mathbb{B}^V$ to $\mathbb{B}$, yielding 1 for the values of variables that correspond to states in P. Realizing this function as an ROBDD requires fixing the ordering of variables. We have in all examples consistently chosen the order in which they occur naturally in the system ("from left to right"). To represent relations $R \subseteq$

$\mathbb{B}^V \times \mathbb{B}^V$ a copy of all the variables of $x \in V$ denoted by $x' \in V'$ is introduced. These will be jointly ordered by interleaving, taking x' immediately after x, and otherwise respect the ordering of V [12]. Relations are represented by Boolean functions f_R from $\mathbb{B}^{V \cup V'}$ to $\mathbb{B}$ yielding 1 exactly for values of the variables that correspond to a pair in R.

The set theoretic operations needed to compute the S_i's, the Q_i's, $S_n^* \multimap Q_n$, and $I \subseteq S_n^* \multimap Q_n$ are 1) the image of a relation when applied to a set of states, 2) the union of two sets of states, 3) the intersection of two sets of states, 4) the product relation $Q \times Q$ of a set of states Q, and finally 5) the "implication" $\multimap$. All five are implementable by standard ROBDD-operations: 1) by existential quantification and conjunction: $\exists \boldsymbol{x}.R \wedge P$, 2) by disjunction, 3) by conjunction, 4) by renaming of variables and conjunction, and 5) by universal quantification and implication: $\forall \boldsymbol{x}'.R \rightarrow P$. These are all described in Bryant's paper [2].

Moreover, in the computation of S_{i+1} we make use of the ROBDD restrict operator of Coudert, Berthet and Madre [6]. This operator allows for the simplification of an ROBDD f to an ROBDD g such that $f \wedge d = g \wedge d$ for some domain of interest d. As d we can use at the i'th step Q_i since Q_i keeps decreasing.

5 Experiments

We carried out a series of experiments with the three verification techniques (**f**, **b**, and **q**) on four examples: a *modulo N-counter*, an *arbiter*, *Milner's Scheduler*, and a *FIFO queue*. We shall briefly describe each example along with the experimental results. All examples share the feature that they are parameterized by a size, n, making possible an analysis of the verification time as a function of the size. Our main emphasis will be on how these running times grow for each of the three techniques. We found no need to precisely enumerate the transitions of all the examples although we do describe Milner's Scheduler in greater detail to illustrate how this is done.

All results are measured using an ROBDD-package implemented in Standard ML of New Jersey version 0.93 running on a SUN Sparc 20.[1] The running time is determined as the total CPU time spent by the Unix process (including garbage collections). The examples have been chosen with the purpose of being simple enough to be easily understandable, and more importantly, to be scalable. In some of the examples we tried a slight variation of the iteration schemes by precomputing the transitive closure of each individual T_i. This precomputation was only rarely an advantage for any of the techniques and is therefore not shown in the examples[2].

[1] The ROBDD package is available from http://www.it.dtu.dk/~hra

[2] The transitive closure of a relation was computed by a straightforward iteration. We did not experiment with any more advanced way of computing it, such as

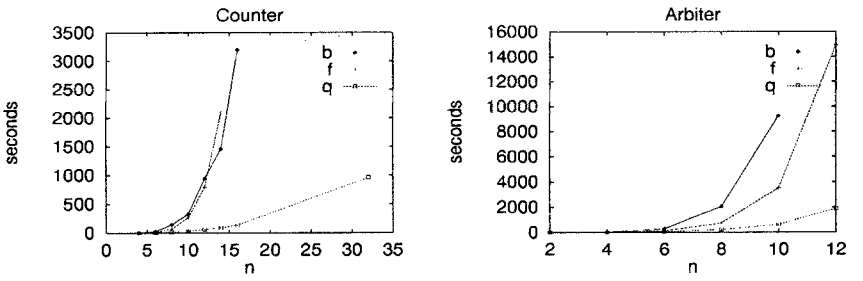

Fig. 2. Running times for the counter and arbiter. Times are in seconds.

5.1 Modulo-N Counter

The first example, a modulo-N counter with constant response time, is a speed-independent hardware design [9]. For simplicity we assume that N is a power of two and thus the counter is a modulo-2^n counter. The counter has one input, a, and two outputs, p and q. Every signal change on the input a is acknowledged by a signal change of either p or q. The first 2^n-1 up-going changes (i.e., from 0 to 1) on a are acknowledged by up-going changes on p and the last change, the 2^n-th, is acknowledged by an up-going change on q. The same with down-going changes. The requirement of constant response time makes the construction non-trivial. The design was done by Christian D. Nielsen [14] and has many similarities with the design described in [9]. We verify the simple property that p and q are never set to one simultaneously, i.e., that for all reachable states $\neg(p \wedge q)$ holds.

The modulo-2^n counter is constructed as the composition of n identical components. The **q**-technique was realized by removing the components in order from the component closest to the output. The experiments (figure 2) show that the quotienting is considerable faster than both the forwards and backwards iterations.

5.2 Arbiter

The second example is an arbiter. An arbiter is a circuit that provides indivisible access to a shared resource, e.g., a bus or a peripheral. The arbiter described here is implemented as a binary tree in which all internal nodes are identical. The arbitration algorithm is based on passing a unique token around the tree. An external process using the arbiter is connected to a leaf of the tree, and it may use the resource only when that leaf has the token. We model the external device by a transition that non-deterministically can choose to issue a request. We verified that no two external devices can be granted access to the shared resource at the same time. (A full description

iterative squaring [4]. The purpose of the experiments was not to validate such tricks but to compare the three iteration techniques.

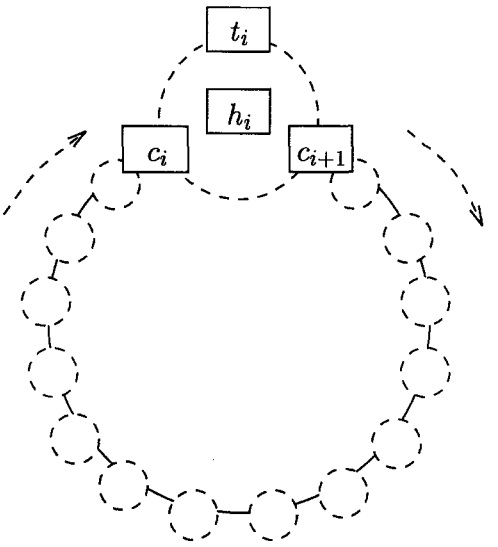

Fig. 3. Milner's Scheduler

can be found in [15].) The running times are shown in figure 2. Again we observe that the quotienting is clearly fastest. We quotiented out from the right-most leave towards the left and up. The other direction from the top and downwards, turned out to be a catastrophe. This seems to indicate that components that assign to variables present in the specification should be quotiented out first.

5.3 Milner's Scheduler

The third example also passes a token but now between processes arranged in a ring. The example is Milner's Scheduler [13]. The system consists of n cyclers, connected in a ring, that co-operates on starting and detecting termination of n tasks that are not further described, see figure 3. The scheduler must make sure that the n tasks are always started in order but the tasks are allowed to terminate in any order. This is one of the properties that has to be shown to hold for the model. The cyclers fulfill this by passing a token: the holder of the token is the only process allowed to start its task.

All cyclers are similar except that one of them has the token in its initial state the others do not. For each cycler i there are three variables t_i, h_i, and c_i. The variable t_i is 1 when task i is running and 0 when it is terminated; h_i is 1 when cycler i has a token, 0 otherwise; c_i is 1 when cycler $i-1$ has put down the token and cycler i has not yet picked it up. Hence, a cycler starts a task by changing t_i from 0 to 1 and detects its termination when t_i is again changed back to 0. It picks up the token by changing c_i from 1 to 0 and puts it down by changing c_{i+1} from 0 to 1.

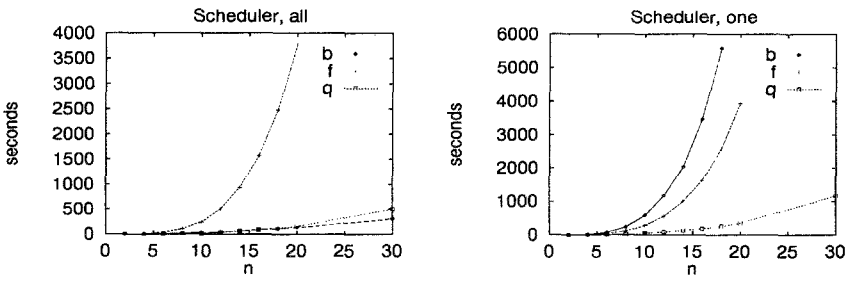

Fig. 4. Running times verifying P_{all} and P_{one} for Milner's Scheduler.

In describing the transitions of the systems, we use for a state s the notation of substitution. For a state s, $s[v/x]$ denotes the state that is identical to s except that it maps x to v. We shall also allow sequences of substitutions to be written as $s[v_1/x_1, \ldots, v_n/x_n]$. Substitutions provide the semantical counterpart of assignments.

The set of variables is $V = \{c_i, h_i, t_i \mid 1 \leq i \leq n\}$, the single initial state is $I = \{s\}$ with $s(c_1) = 1$ and $s(x) = 0$ for all $x \in V \setminus c_1$, and finally the transitions of the i'th cycler are

$$\begin{array}{rcll} T_i & = & \{(s, s[1/t_i, 0/c_i, 1/h_i]) & \mid s(c_i) = 1, s(t_i) = 0, s(h_i) = 1\} \\ & & \cup \{(s, s[1/c_{i \bmod n+1}, 0/h_i]) & \mid s(h_i) = 1, s(t_i) = 0\} \\ & & \cup \{(s, s[0/t_i]) & \mid s(t_i) = 1\}. \end{array}$$

The first two transitions are performed by the cycler, the last by the task it is controlling. We verified two properties, P_{all} and P_{one}. The first expresses that at most one of $c_1, \ldots, c_n$ is true:

$$P_{\text{all}} = \{s \mid \forall i, j.\ 1 \leq i, j \leq n \text{ and } s(c_i) = s(c_j) = 1 \Rightarrow i = j\}$$

and the second that there is not a token on both place c_1 and c_n:

$$P_{\text{one}} = \{s \mid s(c_1) = 0 \text{ or } s(c_n) = 0\}.$$

The cyclers are quotiented out from number 1 and upwards. Figure 4 shows the running times. We observe that the **q**-techniques is again faster than the **f**-technique, comparable to the **b**-technique when verifying P_{all}. It is considerably faster than the **b**-technique when verifying P_{one}. This indicates that the quotienting performs better when the property to be verified is simple.

5.4 Asynchronous FIFO Queue

The fourth example is of a quite different nature. It has the benefit, from an experimental point of view, that we can vary both the amount of internal state and the total number of processes. The example is an asynchronous

FIFO queue consisting of n *FIFO processes* each of which is built up from m *cells*. A cell can hold one of three values: E (for empty), T (for true) and F (for false). Values are sent into the queue as sequences of E, T and F's such that T's and F's are separated by at least one E. A valid input sequence could for instance be $ETTTTEEEFFETETTE$ representing the value sequence T, F, T, T. We shall verify that the output sequence of the buffer also respects the property of having at least one E between changes of T and F. Since we use only Boolean variables we shall encode these three values as the pairs $E = (0,0), F = (0,1), T = (1,0)$ and leave $(1,1)$ as an invalid value.

A FIFO with n processes of m cells contains $2(nm+2)$ variables:

$$V = \{x_0, \ldots, x_{nm+1}, y_0, \ldots, y_{nm+1}\}.$$

Each pair of variables (x_j, y_j) stores one value of a cell. The transitions of FIFO process $i \in \{1, \ldots, n\}$ are as follows:

$$\begin{aligned} T_i \;=\; \{(s, s[s(x_{j-1})/x_j, s(y_{i-1})/y_j]) \mid \\ im \leq j \leq im + m - 1, \\ s(x_{j-1}) = s(y_{j-1}) = 0 \text{ xor } s(x_{j+1}) = s(y_{j+1}) = 0\} \end{aligned}$$

The conditions ensure that the value of the pair (x_{j-1}, y_{j-1}) is only copied to (x_j, y_j) if not both of (x_{j-1}, y_{j-1}) and (x_{j+1}, y_{j+1}) are $E = (0,0)$.

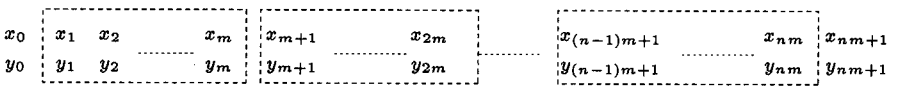

Fig. 5. The FIFO queue with n processes of m cells

Again we quotiented out the FIFO processes starting with 1 and upwards. The running times are shown in figure 6. Quotienting is again fastest. When increasing m to 10 and 20 we observed that the difference becomes slightly larger. Relatively, the quotienting technique is around 3.5 times faster than the forwards iteration for $m = 6$ and this increases to 5 times faster when $m = 20$. For the backwards iteration the increase is from a factor of 2.5 to 3.5.

5.5 Comments on the Experiments

The **q**-technique is better than the **f**-technique in all experiments and, with one exception, it is also better than the **b**-technique. In this one exception (the property P_{all} for Milner's Scheduler) the running times are comparable for the **b**- and **q**-technique. Moreover, it seems that partial model checking behaves better *the simpler the property* to be verified and *the more local state*

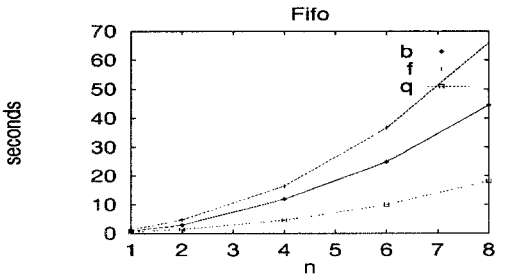

Fig. 6. Running times for the FIFO with $m = 6$

in each process. This was confirmed by Milner's Scheduler and the FIFO queue. However, we observed a curious exception with the arbiter. When we changed the property to simply be that the leaves with numbers $n/2$ and $n-1$ should not both simultaneously be granted access instead of the requirement used in section 5.2, the performance of the quotienting *degenerates* compared to forwards iterations so much that they have almost identical running times when the transitive closure (per process) is precomputed and otherwise the forwards iteration is faster. We have found no good explanation for this.

Of course, measured times can always be questioned, and certainly there are more efficient ROBDD packages around on which the experiments could be repeated. We expect that the three techniques would benefit equally well from a more efficient package.

6 Related Work

The closest related work seems to be Burch et al [3]. They also try to avoid building the complete transition relation $T = T_1 \cup T_2 \cup \cdots \cup T_n$ (using our notation) and instead keep a list of the individual transition relations. When computing the reachable states by a forward iterations, they repeatedly iterate each transition relation independently until a fixed point is reached. Our approach differs in at least three respects.

Firstly, it is a *backwards* iteration that utilizes the property to be verified in simplifying the computation. This avoids constructing the complete set of reachable states. Secondly, a T_i is only used for one fixed-point iteration, whereafter it is added, in a simplified version, to the accumulating set of transitions S_i. Finally, we exploit the modular structure provided by the designer by quotienting out one process – and not only a single transition – at a time. The examples in this paper show that this can reduce the verification effort significantly.

7 Conclusion

A new technique for proving safety properties that attempts to utilize the *structure* of the system under consideration has been presented. A series of experiments has been carried out to validate the new technique. We find that the experiments are promising: In all our experiments the running times are better than for a forwards fixed-point iteration and with one exception (the property P_{all} for Milner's Scheduler) also a backwards iteration, thereby improving on the ROBDD technique which already is a big improvement over naive state-space exploration.

Acknowledgment

Thanks are due to Henrik Hulgaard for his detailed and constructive comments on a draft. Thanks are also due to the anonymous referees.

References

1. Henrik R. Andersen. Partial model checking (extended abstract). In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 398–407, La Jolla, San Diego, 26–29 July 1995. IEEE Computer Society Press.
2. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 8(C-35):677–691, 1986.
3. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proc. 1991 Int. Conf. on VLSI*, August 1991.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE Computer Society Press, 1990.
5. E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 353–362, Asilomar Conference Center, Pacific Grove, California, June 5–8 1989. IEEE Computer Society Press.
6. Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems. Proceedings*, volume 407 of *LNCS*, pages 365–373. Springer-Verlag, 1989.
7. Mads Dam. Compositional proof systems for model checking infinite state systems. In I. Lee and S. Smolka, editors, *CONCUR'95, 6th International Conference*, volume 962 of *Lecture Notes in Computer Science*, pages 12–26, Philadelphia, PA, USA, August 21 - 24 1995.
8. E.W. Dijkstra. *A discipline of programming.* Englewood Cliffs, N.J. : Prentice-Hall, 1976.
9. Jo C. Ebergen and Ad M. G. Peeters. Design and analysis of delay-insensitive modulo-N counters. *Formal Methods in Systems Design*, 3(3), December 1993.

10. Patrice Godefroid and Pierre Wolper. A partial approach to model checking. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 406–415, Amsterdam, The Netherlands, 15–18 July 1991. IEEE Computer Society Press.
11. R.P. Kurshan and Ken McMillan. A structural induction theorem for processes. In *Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 239–248. ACM, 1989.
12. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993.
13. Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.
14. Christian D. Nielsen. *Performance Aspects of Delay-Insensitive Design.* PhD thesis, Department of Computer Science, Technical University of Denmark, 1994.
15. Jørgen Staunstrup. *A formal approach to hardware design.* Kluwer Academic Publishers, 1994.
16. Colin Stirling. A complete compositional modal proof system for a subset of CCS. volume 194 of *Lecture Notes in Computer Science*, pages 475–486. Springer-Verlag, 1985.
17. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

A Proof of the Quotienting Theorem 1

Theorem 1 (Quotienting theorem) *Assume $P \subseteq \mathbb{B}^V$ is a set of states and let $T_i \subseteq \mathbb{B}^V \times \mathbb{B}^V$ for $1 \leq i \leq n$ be n set of transitions with $T = \bigcup_{i=1}^{n} T_i$. Take $Q_0 = P$ and $S_0 = \emptyset$. Define inductively for $i \in \{0, \ldots, n-1\}$:*

$$\begin{aligned} Q_{i+1} &= (S_i \cup T_{i+1})^* \multimap Q_i \\ S_{i+1} &= (S_i \cup T_{i+1}) \cap (Q_{i+1} \times Q_{i+1}) \end{aligned}$$

Then for all $i \in \{0, \ldots, n\}$,

$$T^* \multimap P \;=\; (S_i \cup T_{i+1} \cup \ldots \cup T_n)^* \multimap Q_i \tag{5}$$

Proof. We shall prove (5) by (bounded) induction on i. For $i = 0$ it holds by definition of T, S_0 and Q_0. For $1 \leq i \leq n$, we compute as follows:

$$\begin{aligned} T^* \multimap P &= (S_{i-1} \cup T_i \cup \cdots \cup T_n)^* \multimap Q_{i-1} \\ &\qquad \text{by the induction hypothesis} \\ &= (S_{i-1} \cup T_i)^* \circ (S_{i-1} \cup T_i \cup \cdots \cup T_n)^* \multimap Q_{i-1} \\ &\qquad \text{by a simple property of transitive closure} \\ &= (S_{i-1} \cup T_i \cup \cdots \cup T_n)^* \multimap ((S_{i-1} \cup T_i)^* \multimap Q_{i-1}) \\ &\qquad \text{by (2)} \\ &= (S_{i-1} \cup T_i \cup \cdots \cup T_n)^* \multimap Q_i \\ &\qquad \text{by definition of } Q_i \end{aligned}$$

We shall prove that for arbitrary R, U, and Q satisfying $R \multimap Q = Q$:

$$(R \cup U)^* \multimap Q = ((R \cap Q \times Q) \cup U)^* \multimap Q \tag{6}$$

This allows us to proceed with:

$$\begin{aligned}
&(S_{i-1} \cup T_i \cup \ldots \cup T_n)^* \multimap Q_i \\
&= (((S_{i-1} \cup T_i) \cap Q_i \times Q_i) \cup T_{i+1} \cup \ldots \cup T_n)^* \multimap Q_i \\
&\qquad \text{since } (S_{i-1} \cup T_i) \multimap Q_i = Q_i \\
&= (S_i \cup T_{i+1} \ldots \cup T_n)^* \multimap Q_i \\
&\qquad \text{by definition of } S_i
\end{aligned}$$

proving (5) for i. Now, to prove (6) recall that $(R \cup U)^* \multimap Q$ is the largest fixed point of the map $f : \_ \mapsto (R \cup U \multimap \_) \cap Q$ and $(R_Q \cup U)^* \multimap Q$ of the map $g : \_ \mapsto (R_Q \cup U \multimap \_) \cap Q$ where we abbreviate $R \cap Q \times Q$ by R_Q. Since $\multimap$ is easily seen to be anti-monotone in its left component it is clear that $(R \cup U)^* \multimap Q \subseteq (R_Q \cup U)^* \multimap Q$. We shall prove the other direction by showing that $(R_Q \cup U)^* \multimap Q$ is a postfixed point of f and then since $(R \cup U)^* \multimap Q$ is the largest such fixed point, the inclusion follows from Tarski's theorem [17]. Hence, we apply f to $(R_Q \cup U)^* \multimap Q$ and compute:

$$\begin{aligned}
&R \cup U \multimap ((R_Q \cup U)^* \multimap Q) \\
&= (R_Q \cup U)^* \circ (R \cup U) \multimap Q \\
&= \{s \mid \forall t.\ \exists u.\ (s,u) \in R \cup U, (u,t) \in (R_Q \cup U)^* \ \Rightarrow\ t \in Q\} \\
&= \{s \mid \forall t.\ (\exists u.\ (s,u) \in R, (u,t) \in (R_Q \cup U)^* \ \Rightarrow\ t \in Q)\ \&\ \\
&\qquad\quad (\exists u.\ (s,u) \in U, (u,t) \in (R_Q \cup U)^* \ \Rightarrow\ t \in Q)\} \\
&\supseteq \{s \in Q \mid \forall t.\ (\exists u.\ (s,u) \in R, (u,t) \in (R_Q \cup U)^* \ \Rightarrow\ t \in Q)\ \&\ \\
&\qquad\quad (\exists u.\ (s,u) \in U, (u,t) \in (R_Q \cup U)^* \ \Rightarrow\ t \in Q)\} \\
&= \{s \in Q \mid \forall t.\ (\exists u.\ u \in Q, (s,u) \in R, (u,t) \in (R_Q \cup U)^* \ \Rightarrow\ t \in Q)\ \&\ \\
&\qquad\quad (\exists u.\ (s,u) \in U, (u,t) \in (R_Q \cup U)^* \ \Rightarrow\ t \in Q)\} \\
&\qquad \text{since } R \multimap Q = Q \text{ implies } R \circ Q \subseteq Q, \text{ thus } s \in Q, (s,u) \in R \text{ implies } u \in Q \\
&= \{s \in Q \mid \forall t.\ (\exists u.\ (s,u) \in R_Q, (u,t) \in (R_Q \cup U)^* \ \Rightarrow\ t \in Q)\ \&\ \\
&\qquad\quad (\exists u.\ (s,u) \in U, (u,t) \in (R_Q \cup U)^* \ \Rightarrow\ t \in Q)\} \\
&= \{s \in Q \mid \forall t.\ \exists u.\ (s,u) \in R_Q \cup U, (u,t) \in (R_Q \cup U)^* \ \Rightarrow\ t \in Q\} \\
&= (R_Q \cup U)^* \multimap Q.
\end{aligned}$$

This completes the proof.