A Comparison of Modular Verification Techniques*

Henrik Reif Andersen Jørgen Staunstrup Niels Maretti

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

Abstract. This paper presents and compares three techniques for mechanized verification of state-oriented design descriptions. The goal of this work is to gain insight into quantitative aspects of different modular verification techniques. One of the three verification techniques presented here is a traditional forward generation of a fixed point characterizing the reachable states. This does not utilize any modularity provided by the designer, and therefore it forms the basis for the comparison, whereas the two others do utilize such a modularity. One requires a substantial manual effort by the designer, but is computationally very efficient, while the other requires almost no manual assistance with a much better performance than the simple forward generation. The performance of the three techniques is compared on a set of examples.

1 Introduction

Verification is an important part of any non-trivial design project. It covers a wide range of techniques for uncovering errors, and ideally one would like to do an exhaustive check, where all behaviors of the design are exercised. However, this is seldomly possible in practice. The common practise is to test a sample of the behaviors by execution and/or simulation. Recently, advances in algorithms, data structures, and design languages have provided formal (exhaustive) verification techniques which are powerful enough to handle some significant practical examples [12,13]. In order to use the formal techniques, both the intended and actual behavior must be expressed in formal notation, e.g., as a program in a programming language or as a logic formula. Although these techniques have been demonstrated to work on significant examples, scaling is often difficult. The reason is that the modularity found in most large-scale practical designs has been difficult to exploit in an efficient way in formal verification.

The goal of this work is to gain insight into quantitative aspects of different modular verification techniques. One of the three verification techniques presented here is a traditional forward generation of a fixed point characterizing the reachable states. This does not utilize any modularity provided by

* 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 designer, and therefore it forms the basis for the comparison, whereas the two others do utilize such a modularity. The difference between the two is in the amount of automation. One requires very little effort from the designer while the other assumes that the designer formalizes the behavior at the interfaces between all modules. The analysis done here focuses on both the computational and the manual effort needed for the verification. To stress that the techniques discussed apply to both hardware and software we use the generic term *design*.

The paper is organized as follows. Section 2.1 introduces a simple state-based model that is used to explain and compare different verification techniques. In section 2.2 modularity is added to the simple model. Sections 3–5 give a brief introduction to the three verification techniques that are compared. Section 6 contains the actual comparison.

2 Model

This section defines the model used to describe a design, it is closely related to UNITY[6] and SYNCHRONIZED TRANSITIONS[14]. However, to simplify the presentation, it uses a simplified notation avoiding issues like typing and scope rules.

2.1 States and transitions

A *design* is specified by a *transition system*, that consists of a fixed number of *state variables*: $s_1, s_2, \ldots, s_n$, and a fixed number of transitions: $t_1, t_2, \ldots, t_m$. Each state variable has a value from a fixed domain. A *state* is a mapping of state variables to values: $(s_1 \mapsto v_1, s_2 \mapsto v_2, \ldots, s_n \mapsto v_n)$, where v_i $(1 \leq i \leq n)$ is a value in the domain of state variable s_i. A *transition* is a binary relation on states, called *pre-states* and *post-states*. A design defines a set of *computations* as sequences of states: $S_0, S_1, \ldots$, such that S_0 is an initial state, and for each pair: S_i, S_{i+1}, there is a transition t such that S_i is a pre-state of t, and S_{i+1} is a post-state of t. Furthermore, it is required that $S_i \neq S_{i+1}$. A set of initial states is specified by a predicate.

Notation Transition relations are described by an assignment controlled by a boolean expression, called a *precondition*, for example:

$$s_0 \neq s_2 \rightarrow s_1 := s_0$$

This *transition description* defines a relation that holds only if $s_0 \neq s_2$ in the pre-state; the value of s_1 in the post-state is the value of s_0 in the pre-state, and state variables other than s_1 hold the same value in the pre- and post-states. Sometimes it is convenient to interpret a transition, t, as a predicate $t(pre, post)$ which is true, if and only if t can make a state change from *pre* to *post*. A number of transition descriptions, $t_1, t_2, \ldots, t_n$ can be combined by *asynchronous composition*, $\|$, to one description: $t_1 \parallel t_2 \parallel \cdots \parallel t_n$. The transition relation defined by this composition is the union of the individual relations.

Example 1. A simple oscillator. Let s_0, s_1, and s_2 be three boolean state variables. Together they define a state space with 8 possible states.

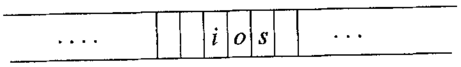
initially
$s_0 \neq s_1 \vee s_1 \neq s_2$
transitions
$s_0 \neq s_2 \rightarrow s_1 := s_0 \;\|$
$s_1 \neq s_0 \rightarrow s_2 := s_1 \;\|$
$s_2 \neq s_1 \rightarrow s_0 := s_2$

The first transition description defines two possible state changes, one leading from a pre-state where s_0 is true and s_2 is false, and another where s_0 is false and s_2 is true. Similarly, the second and third transitions define each two state changes. The transition system is initialized such that the values of the three state variables are *not* the same. The transitions describe a computation where the value of s_0 is propagated to s_1 (and from there to s_2 and back to s_0). **End of example**

The FIFO queue presented next is used as a running example in the rest of the paper. In section 6 other examples are given and used for a quantitative comparison of the different verification techniques. The particular FIFO used here is a fundamental building block in asynchronous circuits [14].

Example 2. The FIFO queue. A FIFO (queue) is a data structure that can hold a sequence of elements. For our purposes, an element has one of three values: E (for empty), T (for true), and F (for false). Elements are inserted into the queue as sequences of E, T and F values such that any T and F are separated by at least one E, for example, $ETTTTEEEFFETETTE$ representing the sequence of values T, F, T, T. Given a state variable, s, the predicate $e(s)$ is true if s has the value E and false otherwise.

The FIFO queue is realized as a number of state variables (each of which can hold an element), and a number of transitions for moving elements down the sequence. When an element is inserted, it moves down the queue. Meanwhile, further elements can be inserted, and several elements can move in parallel. The following transition describes how elements move:



$$e(i) \neq e(s) \rightarrow o := i$$

For simplicity the transitions for input from or output to the environment are not shown. Intuitively, the elements move in a worm-like fashion, where a particular value might be stretched out over several state variables, or it can be compressed into a single state variable surrounded by Es. It is this worm-like behavior that makes the FIFO a key component in asynchronous circuits.

2.2 Modular designs

It is rarely practical to handle a large design as a single monolithic transition system. To be manageable, it must be broken into a number of (almost) independent modules; such modules are called *cells* in this paper. A cell describes a generic (i.e., parameterized) set of state variables and transitions. A specific *instance* contains a distinct set of state variables and transitions, called the *local state variables* and *local transitions.* Any number of instances of a cell may coexist.

The *interface* of a cell is a set of formal parameters; within the cell these are indistinguishable from other state variables, for example,

cell $queue(in, out)$

Here there are two formal parameters in the interface: in, out. When the cell is instantiated, an actual parameter (a state variable) is specified for each formal parameter. Several cell instances can share a particular state variable by making it an actual parameter of the cells.

The sets of transitions of different cell instances are disjoint, therefore, any transition belongs to exactly one cell instance. The collection of all cell instances in a modular design defines a transition system, where the set of state variables is the union of the state variables of the cell instances, and the set of transitions is the union of their transitions. The computation is a sequence of states, corresponding to executions where transitions are executed one at a time.

Notation The notation for describing modular transition systems is not formalized in this paper. This leaves some ambiguity with respect to scope rules, typing etc. However, these details are not necessary for the quantitative comparisons that are the focus of this paper. The notation is a simplification of the design language SYNCHRONIZED TRANSITIONS [14].

Example 3. The modular FIFO. The FIFO can be used to illustrate a modular design consisting of a number of similar segments. The following shows a segment that contains five elements (the choice of five is somewhat arbitrary, see also section 5):

cell $element(i, o, s)$: $e(i) \neq e(s) \rightarrow o := i$
instantiations
element(in, s1, s2) || *element(s1, s2, s3)* ||
element(s2, s3, s4) || *element(s3, s4, s5)* ||
element(s4, s5, out) **End of example**

Structuring a design into cells is primarily a pragmatic concern which may simplify the (development and) verification. The use of cells may contribute to this in several ways. One potential contribution is the generic nature of a cell, which means that the cell is only verified once, even if a large or unknown number of instances of the cell are used. In [11] a similar benefit of generic specifications is used. Another contribution is that even for irregular designs, without re-use of generic cells, it is important to localize the verification to concentrate on one cell at a time.

2.3 Design verification

Properties of a design are formalized as predicates, called *invariants*, constraining the transition system, for example, that no two neighboring elements in a FIFO have different non-empty values. An invariant defines a subset of the state space containing the initial state. Furthermore, there must not be any transitions from a state within the subset to a state outside. Hence, invariants describe properties which hold throughout the computation, because no transition will go to a state violating it. An invariant is written as a predicate, $I(S)$, on a state S.

Example 4. The FIFO queue (continued). The transitions of the FIFO queue ensure that there will never be a state where two neighboring elements, $s1, s2$, contain two different non-empty values, this property is expressed as an invariant, called the *alternation invariant.*

$$\textbf{invariant } A(s1, s2) : (e(s1) = e(s2)) \quad = \quad (s1 = s2)$$

Note that = is an overloaded binary operator used for comparing both boolean values and the ternary values stored in the queue. **End of example**

Invariants are used to formalize safety properties of a design. The modularization provided by the cell mechanism can also be reflected in the invariants, because cells may have their own local invariants stating internal properties. The next three sections (sections 3–5) present three different *verification techniques* for showing that a given predicate is an invariant. There are many interesting properties of a design, e.g., liveness properties, that cannot be expressed as invariants. This paper does not advocate using only invariants for designing and verifying realistic designs. However, invariants are sufficient to demonstrate the quantitative differences between the verification techniques presented here.

3 Localized verification

This section describes an induction based verification technique for verifying an invariant [9]. Assume that I is an invariant and that t is a transition of a design, then t is said to *maintain the invariant* if,

$$I(pre) \wedge t(pre, post) \Rightarrow I(post)$$

i.e., if the invariant holds in the pre-state then it is shown to hold in the post-state. By showing that the invariant holds in the initial state and by showing the implication for *each transition description*, t, of the design one may conclude that the invariant holds throughout the computation. This verification technique is really an induction proof [9] (over the computations of the design) where the implication shown above corresponds to the induction step. The effort needed to do the induction step is proportional to the number of transition descriptions and cell instantiations in the textual description of the design; but *independent* of the size of the state space or the length of the computation.

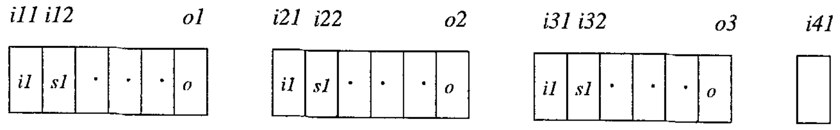
Example 5. The FIFO queue (continued). One segment of the FIFO queue design has five transitions of the form:

$e(i) \neq e(s) \rightarrow o := s$

To show that the invariant holds for this segment five implications (one for each transition) must be shown. If the size of a segment is increased by adding more transitions then the verification efforts grows proportionally. However, a better way to describe a large FIFO queue is to instantiate a number of cells (segments). **End of example**

When a large design is divided into cells, it is possible to divide the verification in a similar modular fashion. This means that a cell description needs only to be verified once, no matter how many times it is instantiated. In [15] it is shown how the latter can be exploited to yield a *localized verification technique* with a constant verification effort for each instantiation (two implications). This is also an inductive technique.

Example 6. The FIFO queue (continued). The localized verification technique can be illustrated on the FIFO queue realized as three segment cells:



cell *segment*$(i1, s1, o, s)$
  invariant $I_{segment}$: $A(i1, s1) \wedge A(s1, s2)$ ...
  transitions ...
cell *FIFO*
  invariant I_{FIFO} : $A(o1, i21) \wedge A(i21, i22)$ ...
  instantiations
    segment$(i11, i12, o1, i21)$ || *element*$(o1, i21, i22)$ ||
    segment$(i21, i22, o2, i31)$ || *element*$(o2, i31, i32)$ ||
    segment$(i31, i32, o3, i41)$

To verify this design, one first shows that each of the transitions in the cell description of a segment maintains the invariant ($I_{segment}(pre) \wedge t(pre, post) \Rightarrow I_{segment}(post)$), this is called *local invariance*. Note that each transition is only verified once, no matter how many times it is instantiated.

To verify an instantiation of the cell *segment*, one shows that no transition in the instantiated cell violates the invariant of its environment (the global level of the FIFO and the other cell instances), this is called *up-ward non-interference*. Furthermore, no transition in the environment must violate the invariant of the instantiated cell, this is verified by showing *down-ward non-interference*. To show these two non-interference properties for the first instantiation of the FIFO cell, it is sufficient to show the following two implications denoted *UP* and *DOWN*.

$$UP: \; I_{segment}(pre) \wedge I_{FIFO}(pre) \wedge I_{FIFO}(post) \wedge S_g \Rightarrow I_{segment}(post)$$

where S_g is a predicate stating consequences of the cell structure, for example, that the transitions of the first FIFO segment cannot change state variables $i11, i21, i22, o2, i31, \ldots$.

$$DOWN: \; I_{FIFO}(pre) \wedge I_{segment}(pre) \wedge I_{segment}(post) \wedge S_l \Rightarrow I_{FIFO}(post)$$

where S_l is a predicate stating consequences of cell structure, for example, that the global transitions of the FIFO cannot change local state variables of the first segment $s2, s3, s4, \ldots$. **End of example**

The verification technique, illustrated by the FIFO example, is useful in general. Further examples and a more detailed explanation is given in [15] where it is also shown that the technique is sound. Each line of a design description gives rise to zero (declarations, headers, etc.), one (local invariance) or two implications (non-interference), and this is the justification for the claim that *the verification effort grows linearly with the size of the textual design description*. In fact, for recursively defined cell, the effort needed to show non-interference is independent of the recursion depth. This is because the recursive instantiation of a cell yields just the two non-interference implications.

However, the efficiency of the localized verification has a price. First of all, the technique is not complete. One can easily construct an example of a correct design where it is not possible to show the required implications (*UP* or *DOWN*). In practice this does not seem to be a unsurmountable problem; a significant number of examples have been verified [14]. A more important practical problem is inherent in the inductive approach on which the technique is based. It is based on showing implications such as:

$$I(pre) \wedge t(pre, post) \Rightarrow I(post)$$

Note that the invariant I appears both as an assumption and as a conclusion. This means that one has to find the right balance when stating an invariant. If it is made very strong ($I(pre)$ very restrictive), it means that $I(post)$ also becomes very strong and hence difficult to prove. On the other hand making the assumptions very weak, can make it difficult to conclude that $I(post)$ holds. This has turned out to be a significant practical problem. To show an invariant using the inductive approach, it is often necessary to find a stronger assertion than the straightforward formulation of the desired property. To illustrate this, assume that the designer for some reason only wants to verify the invariant $A(o3, i41)$, i.e. that the last two elements in the FIFO queue does not contain different non-empty values. In order to verify this using the inductive hypothesis, it is necessary for the designer to formulate a much stronger invariant. Identifying and formulating this is often a significant part of the verification effort. Hence, the linear growth of the verification effort has its price, namely an added effort by the designer to identify and state auxiliary invariants.

4 Computation of reachable states

This section describes a technique that overcomes the difficulty of finding invariants by computing the strongest of them all: A predicate characterizing exactly the set of reachable states. Having computed the set of reachable states, the verification task is reduced to checking that the predicate characterizing this set implies the property of interest. The set of *reachable states* is the subset of the state space that can be reached by a sequence of transitions from any of the initial states. This set is often computed as an increasing sequence of approximations starting with the initial states and in each step adding what can be reached by making one further transition [7]. If the system is finite, this sequence of approximations will always converge to the full set of reachable states in a finite number of steps. This is called the *forward* generation technique.

This computation requires choosing a representation for sets of states. Using an explicit representation very quickly leads to a combinatorial explosion of the number of states generated, resulting in poor performance. However, *implicit* representations of state sets with clever datastructures can in many real examples overcome the problem. We shall use the implicit representation known as *Reduced Ordered Binary Decision Diagrams* [3], ROBDDs. They provide compact representations of boolean functions using a special kind of directed acyclic graphs. All the standard boolean operations are reflected by ROBDD-operations that are implemented as efficient algorithms on the underlying datastructure. Representing sets of states by their characteristic boolean functions provides the needed representation.

The use of ROBDDs requires choosing an ordering of the boolean variables in the design. This choice greatly influences the efficiency of the ROBDD representation. McMillan [13] gives some advice for choosing an ordering for circuit designs which we have followed: The variables must be ordered according to how they appear in a depth-first traversal of the circuit. Furthermore, pre- and post-variables should be interleaved when representing transitions.

Using these orderings, the initial states, the set of transitions, and the reachable states are all represented as boolean functions. After computing the set of reachable states, the verification task is reduced to checking that the boolean function characterizing this set implies the property of interest. This

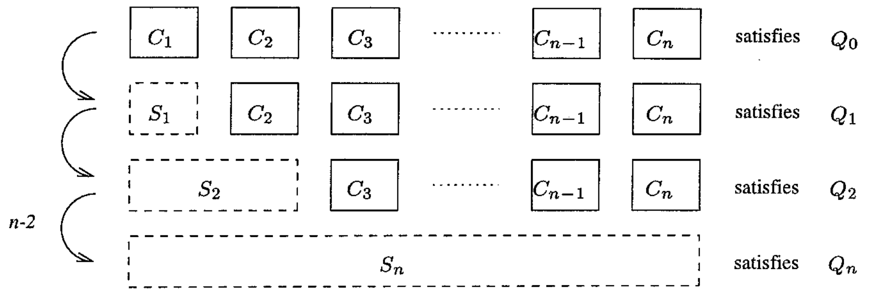


Fig. 1. Sketch of the quotient technique.

final implication is also computed as an operation on ROBDDs. ROBDDs are not guaranteed to avoid the combinatorial explosion — and on some real examples they fail to do so [4] — but they do on very many examples, providing one of the most successful heuristics currently known.

There are two important differences compared to the localized verification technique described in section 3. One is that the forward generation technique does not require any manual assistance from the designer in formulating auxiliary invariants needed. To verify the invariant $A(o3, i41)$ in the FIFO queue no additional effort is needed from the designer. The other significant difference is that neither the forward generating technique nor the ROBDDs make use of the modularity of the design. The next section presents an automatic technique where this is done.

5 The quotient technique

The third technique we shall present combines the ROBDD technique with a use of the modularization of the design and maintains the automation of the forward generation. Instead of computing the set of reachable states by a forwards iteration from the initial states, *the quotient technique* is a modified backwards iteration from the property to be verified towards the initial states. A backwards iteration utilizes knowledge of the property to be verified. If this property is simple, the hope is that the intermediate sets of states are also simple. The quotient technique with ROBDDs is a refinement and modification of this idea.

Figure 1 gives a sketch of the technique: Q_0 is the property to be verified; Q_1 is constructed from Q_0 by backwards iterating with the transitions from cell C_1 until a fixed point is reached; S_1 is constructed by restricting C_1 to the subset $Q_1 \times Q_1$, i.e., S_1 is a simplified version of C_1. As we proceed, Q_{i+1} is constructed from Q_i by backwards iterating with the transitions from cell C_{i+1} and the simplified representation S_i of the transitions of cell C_1 to C_i; S_{i+1} is constructed from the union of C_{i+1} and S_i by restricting to the subset found as Q_{i+1}. The verification is done by a final backwards iteration of S_n

from Q_n, followed by a check to decide whether this set contains the initial states.

More precisely, we take $S_0 = \emptyset$ and define for $i \in \{0, \ldots, n-1\}$:

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

where T^* is the transitive, reflexive closure of a transition relation T, $\times$ forms the Cartesian product of two sets, and $T \multimap Q$ is the set of states that through a transition in T only can lead to states in Q. The operation $T \multimap Q$ is defined by:

$$T \multimap Q \;=\; \{s \mid \forall s'.\ (s, s') \in T \;\Rightarrow\; s' \in Q\}.$$

(In program verification this is known as the weakest precondition.) The set $(S_i \cup C_{i+1})^* \multimap Q_i$ is found by a backwards fixed-point iteration.

The quotient technique "removes" the cells of the design one-by-one. The order in which this is done must be determined manually. Since the intermediate sets generated will vary with the order, the choice of order can influence the efficiency. For a design that has a linear topology there are two obvious choices: from right to left or from left to right. For other topologies like for instance binary trees the best choice is less obvious.

The quotient technique has two potential benefits over a direct backwards iteration [10]. Firstly, the full next-state relation which is a disjunction of all transitions of the design need not be computed – a computation that is often costly, as reported for instance in [10]. Secondly, the intermediate ROBDDs constructed during the iteration tend to be simpler than the intermediate ROBDDs in the simple backwards iteration.

For details and more experimental evidence on the quotient technique, see [1] (in this paper the technique is called partial model checking) and [2].

6 Quantitative comparison

This section presents a number of quantitative comparisons of the verification techniques presented in sections 3 to 5. The comparisons are based on three examples: The FIFO queue, a Modulo-N counter, and a tree arbiter. All three are rather simple to describe as modular designs where the size can be varied in order to analyze how the verification effort grows with the size of the problem.

All experiments were carried out on a Sparc 20 with 96 MB of memory using an ROBDD package written in Standard ML of New Jersey, version 0.93.[1] The package was written with the purpose of ease of use and no special attention was drawn to optimizing efficiency. Thus less emphasis should be put on the absolute running times than the *relationship* between the results.

[1] The package is freely available via Internet: `http://www.it.dtu.dk/~hra`.

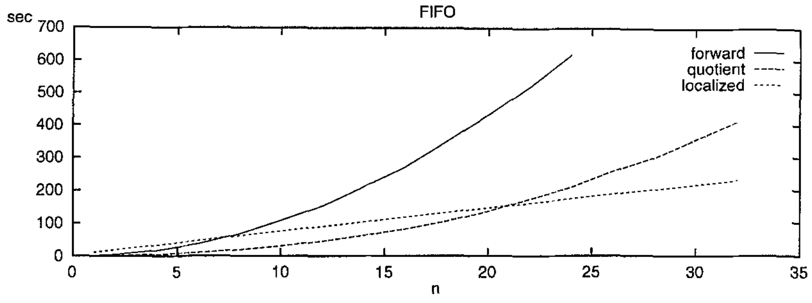


Fig. 2. Running times for the FIFO with $m = 6$

6.1 The FIFO queue

The first series of experiments were carried out with the FIFO queue. It was verified that the alternation invariant holds for the last two elements. The design is parameterized in m, the size of each segment, and n, the number of segments. With localized verification a set of implications are extracted and these are shown to hold by the ROBDD package. As described in section 3 the number of implications grows linearly with n (and m). The manual effort required is to state that the alternation invariant holds everywhere in the FIFO.

The forwards iteration requires no manual effort. All the work is done by the ROBDD package. The quotient technique requires choosing an ordering of the cells. Since the modular structure is a linear sequence there are two obvious choices. The one used is from output-to-input. Figure 2 shows the running times for $m = 6$ as a function of n. For the forward and quotient techniques these are third degree polynomials. However, the polynomial for the quotient has much smaller constant factors, resulting in better performance. We tried increasing m and observed that the difference between the two also increases. This seems to confirm the assumption that the quotient technique can benefit from the cells containing much local state. In fact, the quotient technique is so efficient that the state space must be of considerable size before the localized verification is advantageous. (For $n = 20$ the state space is of size 2^{120}.) But from that point on, nothing seems to compete with the linear growth of the localized verification.

6.2 Modulo-N counter

The modulo-N counter with constant response time is a simple example of a speed-independent design [8]. To simplify the presentation, it is assumed that N is a power of two, and therefore 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 on a are acknowledged by up-going changes on p and the last, 2^n-th, by an up-going change on q. The same with down-going changes.

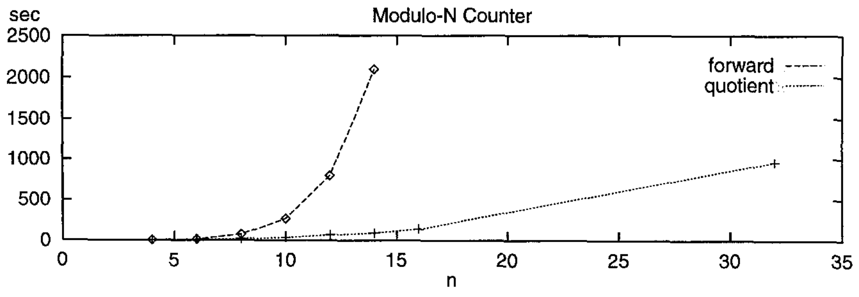


Fig. 3. Running times for the modulo 2^n-counter.

We verified the property that at each point in time only one of p and q holds the value 'one' at the output of the counter. The running times are shown in figure 3. The forwards iteration and quotient techniques again behave as third degree polynomials. The quotient, however, has dramatically better running time than the forwards iteration. We anticipate that the main reason is that the quotient technique can fully benefit from the original property being simple and the local state relatively large (each cell contains 7 boolean variables).

The design of the counter can be given as a recursive description, which means that the number of implications coming from the localized verification is *constant*, independent of n. The running time is therefore a horizontal line very low in the figure (not drawn). The price here is, however, that a relatively strong invariant must be supplied by the designer.

6.3 A tree 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 nodes (including the root and the leaves) 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.

Each node of the tree has three pairs of connections (see figure 4), one for its parent and one for each of its children. A connection pair consists of two state variables, *req* and *gr*, standing for request and grant. Such a pair is used according to the following four-phase protocol: A node requests the token by setting *req* to true. When *gr* becomes true, the node has the token and may pass it down the tree. The token is handed back by setting *req* to false. When *gr* becomes false, a new request can be made. Figure 4 shows a few nodes and their interconnections. The complete design description is shown in [14].

We verified that no two children could be granted access at the same time (*mutual exclusion*). Contrary to the two previous examples, the natural modularization of the design is a tree and not a linear sequence. When applying

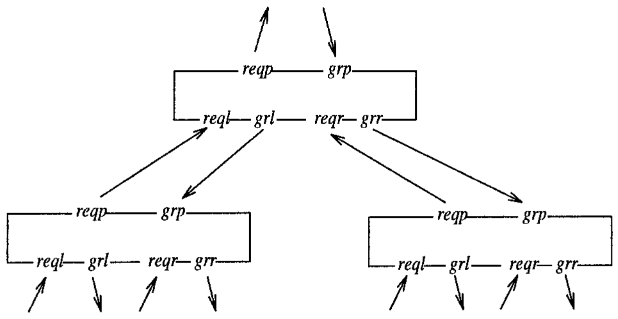


Fig. 4. Two levels of arbiter tree

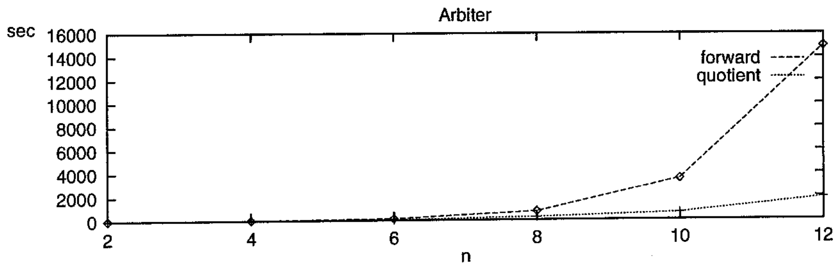


Fig. 5. Running times for the arbiter when verifying mutual exclusion between all pairs of children.

the quotient technique a sequencing of the cells must be chosen. We decided to first divide out the leaves and then move upwards in the tree. Again, we observed that the quotient technique performs better than the forward iteration. This time each cell has 4 variables but the property was not as simple as in the previous two cases. It expressed that no pair of children could be granted access at the same time: a conjunction of the order of n^2 conjuncts. Still the quotient technique seemed to provide an advantage. The order of performing the quotienting was significant: starting from the root and moving down results in considerably worse performance.

The arbiter can easily be described recursively, yielding only a constant number of implications to be verified when using localized verification. However, this time considerable human effort is involved. It is necessary to formalize the four phase protocol described above in order to actually prove the property. This has been done (see [14]), but it requires manual effort to do.

7 Related work

The closest related work seems to be Burch et al [5]. They also try to avoid building the complete transition relation $t = t_1 \vee t_2 \vee \cdots \vee t_n$ (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 C_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 cell at a time. The examples shown in this paper show that this can reduce the verification effort significantly. We would expect this to be the case for most examples of practical relevance.

8 Conclusion

This paper has presented and compared three mechanized techniques for verifying safety properties of state transition systems. The comparisons have focused on the quantitative properties of the three techniques. The forward generation of the reachable states puts the minimal demand on the designer, all that is necessary is to formalize the property to be verified. However, the automation is computationally expensive which limits the size of the designs that can be verified using this technique. In contrast to this, the verification effort needed by the localized technique is much smaller. Unfortunately, the price payed for this is an extra effort needed by the designer to formulate predicates characterizing the interfaces between the modules of a design.

The third alternative, the quotient technique, can be viewed as a compromise between the other two. Some manual assistance is needed by the designer to identify the module structure of a design, but once this has been done the technique proceeds completely automatically.

These claims are supported by the three examples presented in section 6. The examples are relatively simple, but all three techniques have been successfully used on a number of other examples. The localized technique is supported by tools for generating and proving the required verification conditions [14][2].

Acknowledgements

Thanks to Henrik Hulgaard for commenting on an earlier draft and to the anonymous referees for constructive comments.

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] A package is freely available via Internet: `ftp://ftp.it.dtu.dk/pub/ST`

2. Henrik R. Andersen, Niels Maretti, and Jørgen Staunstrup. Partial model checking with ROBDDs. To appear in Proceedings of TACAS'97. LNCS.
3. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Transactions on Computers*, 8(C-35):677–691, 1986.
4. R.E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *Computing Surveys*, 24(3):293–318, September 1992.
5. 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.
6. K. Mani Chandy and Jajadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
7. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.
8. 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.
9. R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Proceedings of the Symposium in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.
10. Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of the 4th Workshop on Computer Aided Verification, CAV'92, June 29 - July 1, 1992, Montreal, Quebec, Canada*, volume 663 of *LNCS*, pages 82–95. Springer Verlag, 1992.
11. Jeffrey J. Joyce. Generic specification of digital hardware. In *Designing Correct Circuits, Oxford 1990*, pages 68–91. Springer-Verlag, 1991.
12. J.P. Billon and J.C. Madre. Original concepts of PRIAM, an industrial tool for efficient formal verification of combinational circuits. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 487–501, Glasgow, Scotland, 1988. IFIP WG 10.2, North-Holland. IFIP Transactions.
13. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993.
14. Jørgen Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.
15. Jørgen Staunstrup and Niels Mellergaard. Localized verification of modular designs. *Formal Methods in System Design*, 6(3):295–320, June 1995.