Trace theoretic verification of asynchronous circuits using unfoldings

K. L. McMillan

Cadence Berkeley Labs

Abstract. An approach is presented for hierarchical, trace-theoretic verification of speed-independent circuits based on Petri net unfolding. The purpose is to avoid the explosion of states that results from interleaving of concurrent transitions. The trace structures of the circuit components are represented by Petri nets. Conformance between implementation and specification is tested by composing the implementation with the mirror of the specification, unfolding the resulting product net into an occurrence net, and testing this net for failures. The latter problem is shown to be NP-complete, however a practical branch-and-bound algorithm is presented. In two examples of scalable asynchronous control circuits, the unfolding size is found to grow linearly with the circuit size, while the number of states grows exponentially. In one case, the unfolding method succeeds in verifying large configurations while BDD-based traversal techniques do not.

1 Introduction

In [McM92b] a structure called an occurrence net was suggested as an alternative representation to a state graph. An occurrence net (as defined by Nielsen, Plotkin and Winskel [NPW81]) is an acyclic net, obtained by unfolding a Petri net. The advantage of this representation from a verification point of view is that it avoids the explosion of states that can result from the many possible "interleavings" (*i.e.*, total orders) of essentially independent events (*i.e.*, signal transitions). In [McM92b] a practical method is described for constructing the unfolding and truncating it when it is sufficient to represent implicitly all of the reachable states. The benefit of not searching all interleavings was shown to be quite significant, for example, in the case of a scalable distributed mutual exclusion (DME) circuit. The number of states was found to be exponential in the circuit size (or number of clients of the DME), while the unfolding size was found to be quadratic.

The previous work treated only reachability analysis and deadlock – essentially it provided a way of testing whether a state satisfying a given conjunction of local conditions is reachable, or whether a state with no enabled transitions reachable. In this work, the notion of truncated unfolding is applied to a *verification* problem – that is, to determine whether a given structure correctly implements its specification. The notion of verification used here is that of Dill's trace theory for asynchronous circuits. A circuit or specification is represented by a trace structure – a set of sequences of transitions on input and output signals

that are labeled "successes" and a set that are labeled "failures". Verification is framed in terms of an asymmetric relation called "conformance" between two trace structures. If A conforms to B, then substituting A for B in any context will be safe, in the sense of not introducing any new failures. Because both implementation and specification are trace structures, this method lends itself to hierarchical verification.

For computational purposes, Dill represents trace structures by the regular languages associated with finite automata. Thus, each gate in a circuit is represented by a small automaton. The representation for the entire circuit is obtained by composing the automata for the individual gates. Conformance to a specification is tested by forming the "mirror" of the specification (*i.e.*, exchanging its inputs and outputs), composing this mirror with the implementation, and testing the resulting closed system for failures (a failure occurs when one component receives an input it is not enabled to receive). This test is accomplished by exhaustive state-space search. In this computation, all total orders of signal transitions are considered, even though some signal transitions may be "independent", in the sense that their total order is irrelevant, as no component of the system may distinguish this order.

In this work, we replace the composition of finite automata with a synchronous composition of Petri nets. This composition of nets may be unfolded into a single, acyclic net, much as described in [McM92b]. The unfolded net captures the partially ordered structure of events in the circuit. Once the unfolding is constructed, the problem of determining whether it is failure-free is NP-complete. The NP-completeness of this problem should be considered, however, in light of the fact that the overall conformance testing problem is PSPACE complete. In fact, a branch-and-bound algorithm described here appears to solve the problem effectively for some example conformance testing problems. In the cases of two scalable speed-independent arbiter designs, we find the size of the truncated unfolding generated in the verification process to scale *linearly* with the circuit size. In one case (that of a tree arbiter), the performance of the unfolding technique is significantly better than that of the BDD-based symbolic model checking technique [BCM+90].

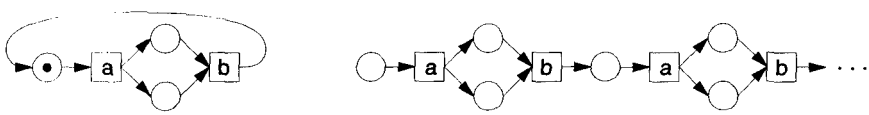
This paper begins with a brief and somewhat informal presentation of Petri nets and the unfolding algorithm in section 2. There follows a similarly brief treatment of trace theory for asynchronous circuit verification in section 3. Section 4 then covers the method of representing trace structures using Petri nets, and section 5 covers the algorithm for testing trace conformance using unfoldings. Finally, section 6 gives an example of how testing trace conformance using unfoldings can avoid the state explosion that would otherwise result from interleavings of concurrent signal transitions.

2 Petri nets and unfoldings

Petri nets are well suited to modeling asynchronous circuit elements and control flows in asynchronous circuits. An introduction to nets can be found in [Pet77]. A *net* consists of a set of *places* P, a set of *transitions* T and an *incidence relation* F, where $F \subseteq (P \times T) \cup (T \times P)$. That is, F is a set of arcs from places

to transitions and transitions to places. The set of predecessors of any given transition t is called its *preset* and is denoted ${}^\bullet t$. Similarly the set of successors of a given transition is called its *postset* and denoted $t^\bullet$. A *marking* of a net is an assignment of tokens to places. In any given marking, a transition is said to be *enabled* if every place in its preset contains a token. The firing of an enabled transition produces a new marking by removing a token from each place in the preset and placing a token on each place in the postset. Any marked net N defines a set of firing sequences $\mathcal{F}(N)$, which are all the sequences of transitions that can legally fire starting with the given initial marking. A firing sequence π has a postset $\pi^\bullet$ which is the set of tokens remaining after the last transition in the sequence has fired.

An *occurrence net* [NPW81] is a specialized form of net having the following additional properties: first, the incidence relation is *well founded*, meaning that there is no infinite path running backward from any node (and therefore that the net is acyclic); second, it has no *forward conflict*. A forward conflict occurs when a given place has more than one predecessor (and similarly a backward conflict occurs when a place has more than one successor, but this is allowed in an occurrence net). If N is a net, then an N-labeled occurrence net is a net N' with a labeling function L' that maps every place (transition) of N' to a place (transition) of N. Any marked net N can be *unfolded* to create an N-labeled occurrence net N'. As an example, figure 1 shows a simple marked net and its unfolding.



a) Marked net b) Unfolded occurrence net

Fig. 1. Example of unfolding

The exact definition of the unfolding, and a procedure for constructing it can be found in [McM93, McM95]. Here we will discuss only some key properties of unfoldings that are relevant to the algorithm for testing trace conformance.

The first key concept is that of a *configuration*. This is any set of transitions in an occurrence net that is backward closed and free of backward conflict. A set of transitions is backward closed if any predecessor of an element is also an element. A backward conflict occurs when two transitions in the set have a common place in their presets. The transitions in a configuration are partially ordered by F'^*, the transitive closure of the incidence relation. The configuration represents a partially ordered run of the net, in the sense that any total order on these transitions consistent with the partial order is a legal firing sequence. Every configuration has a postset – the set of places marked after all the transitions in the configuration have fired. This set is independent of the total order in which

the transitions fire. Thus, while a configuration may correspond to many firing sequences of the original net, it has a unique final state.

Another important concept is that of a *local configuration*. If t' is a transition in the unfolding, the local configuration of t', denoted $\lceil t' \rceil$, is the minimal set of transitions necessary to fire t'. The local configuration $\lceil t' \rceil$ is by definition, the least backward closed set of transitions containing t', (provided this is conflict-free). Similarly, if S is a set of places, we can define its local configuration $\lceil S \rceil$ as the least set of transitions necessary to mark all the places in S. The local configuration of a set of places S is the least backward closed set of transitions containing all predecessors of S, provided this is conflict free and does not remove any tokens from S. A set of places S in an occurrence net is *concurrent*, in the sense that all can be marked simultaneously, exactly when it has a local configuration.

The local configuration of S defines, in a sense, its past. Similarly we can identify the future of this set as the set of transitions which may be fired starting from a state with tokens only on S. We will denote this set $\lfloor S \rfloor$. Thus, a set of concurrent places, which we will refer to as a *slice*, defines two subnets of the unfolding, as depicted in figure 2. In implementing algorithms on unfoldings, it is convenient to represent subnets implicitly as either the past or future of a given slice.

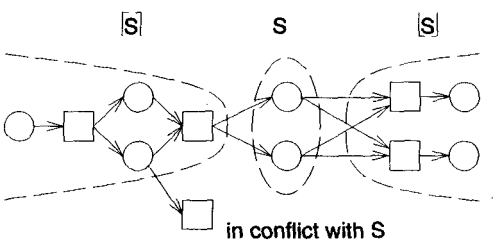


Fig. 2. A slice S, its past $\lceil S \rceil$ and future $\lfloor S \rfloor$.

In [McM92b] an algorithm is described for building the unfolding N' of a marked net N. This is done essentially by starting with a set of places corresponding to the initial marking of N, then proceeding to enumerate all the concurrent sets of places in N' that might match the preset of some transition in N. Each time a matching set of places is found, a new transition is added to N' and places corresponding to its postset are generated. A method is described to terminate this unfolding in such a way that every reachable marking of the original net is represented by the postset of some configuration in the resulting (truncated) unfolding. From the truncated unfolding it is a straightforward matter to test whether a given *conjunctive* condition is reachable in the original net (that is, whether a certain collection a places can be simultaneously marked). It suffices to find a matching set of concurrent places in the unfolding. The local configuration of this set can be linearized to produce a firing sequence leading to

the desired condition. We will need an additional algorithm, however, to verify circuits in the trace theoretic framework, since the condition for failure in this case is not conjunctive.

3 Trace structures and conformance

In the trace theoretic framework [Dil88], a circuit is represented by a set of *traces*. Each trace is a sequence of transitions on the inputs and outputs of the circuit. The traces are divided into a *success set* and a *failure set*. Failures generally represent a situation where the circuit has received an input it was not ready to receive. As an example, consider the simplest possible gate, a non-inverting buffer, with input a and output b. The success set of this gate would be any sequence of transitions of the form $ababab\ldots$. That is, each input transition is followed by an output transition (note, no distinction is made between rising and falling signal transitions, since these always alternate). A failure trace for this circuit would be aa, since after receiving an input transition, the gate is not ready to receive another transition until it has produced an output (such a situation is a *hazard* and may result in a nondigital pulse or "glitch" on the output of the gate).

The trace framework provides an operation for composing circuits that has the effect of synchronizing inputs and outputs. The composition of x and y is denoted $x|y$. Two trace structures may be composed if they have no outputs in common. Outputs of either component become outputs of the composition. For example, suppose we compose two simple buffers, one with input a and output b, and the other with input b and output c. A trace is found in the composition if it is found in both components when projected onto their respective alphabets (sets of input and output signals). The trace is a failure if either component categorizes it as a failure. Thus, for example, in the composition of two buffers, abc is a success, since ab is a success for the first buffer, while bc is a success for the second buffer. On the other hand, $abab$ is a failure, since bb is a failure for the second buffer.

The notion of verification in this framework is called *conformance*. If trace structure x conforms to trace structure y, then x may be substituted for y in any failure-free context, and the result will still be failure-free. With respect to conformance, every trace structure is equivalent to a *simple* trace structure. A simple trace structure is specified only by its set of inputs and outputs and its set of successes. Implicitly, a failure is any trace of the form σi, where σ is a success, i is an input, and σi is not a success.

Every trace structure y has a *maximal environment*, also known as the *mirror* of y and denoted y^M. A trace structure x conforms to y exactly when $x|y^M$ is failure-free. If y is a simple trace structure, then y^M is simply y with inputs and outputs exchanged. Thus, the problem of testing conformance reduces to testing whether a composition of simple trace structures is failure-free. The rest of this paper is concerned with solving this problem.

4 Net representation of trace structures

In this section, we will see that simple trace structures can be represented by labeled Petri nets, that a composition of nets can be defined that corresponds to trace structure composition, and that conformance testing can be accomplished by composing the implementation with the mirror of its specification, unfolding the resulting composed net and testing it for failures. A failure in this case is a state (marking) in which some output transition is enabled to occur, but no corresponding input transition is enabled. The purpose of using net models and unfoldings is to avoid enumerating all reachable states of the composed system.

4.1 Modeling trace structures using nets

The first problem that arises is how to represent a simple trace structure using a Petri net. We will use a net to represent the success set of the trace structure (the failure set being implicit). Each transition of the net is labeled with an input signal or an output signal. Then each firing sequence contributes a trace to the success set of the trace structure. As an example, consider modeling a "C" element – a circuit which produces a transition on its output when transitions have occurred on all of its inputs. If the inputs of this circuit are a and b, and the output is c, then the success set can be represented by the net of figure 3.

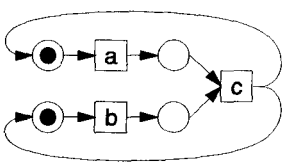


Fig. 3. Net representing success set of "C" element.

Definition 1. A *process* M is a tuple (I, O, N, L), where I, O are distinct sets of signals, $N = (P, T, F)$ is a Petri net, and L is a function $T \rightarrow (I \cup O)$. $\mathcal{T}(M)$ is a simple trace structure with inputs I and outputs O, where:

- the success set S is $\{L(\sigma) \mid \sigma \in \mathcal{F}(N)\}$ and
- (implicitly) the failure set is $\{\sigma i \mid \sigma \in S,\ i \in I,\ \sigma i \notin S\}$.

4.2 Product nets

We can determine whether a composition of processes is failure-free by constructing a product of the corresponding nets, and identifying states in this product at which failures occur.

Definition 2. Let M_i be a process for $i = 1 \ldots k$, such that $O_1 \ldots O_k$ are disjoint. The *product* $M = M_1 \times \cdots \times M_k$ is a process such that:

- $O = \cup O_i$,
- $I = (\cup I_i) \setminus O$.
- P (the set of places) is the disjoint union of $P_1 \ldots P_k$
- there is a transition $t \in T$ with a given ${}^\bullet t$, $t^\bullet$ and $L(t)$ exactly when, for all i s.t. $L(t) \in (O_i \cup I_i)$, there exists a $t_i \in T_i$ s.t.:
  - $L(t) = L_i(t_i)$,
  - ${}^\bullet t \cap P_i = {}^\bullet t_i$
  - $t^\bullet \cap P_i = t_i^\bullet$

In other words, every transition in the product has the effect of simultaneously firing a transition from each component process that inputs or outputs the given signal. From this definition, it is not difficult to show that the success set of the composition of the corresponding trace structures is equal to the success set of the product of the processes:

Theorem 3. *Let M_i be a process for $i = 1 \ldots k$, such that $O_1 \ldots O_k$ are disjoint. The success set of $\mathcal{T}(M_1)|\cdots|\mathcal{T}(M_k)$ equals the success set of $\mathcal{T}(M_1 \times \cdots \times M_k)$.*

4.3 Failures states

Constructing the failure set is a bit more subtle, however. To be a failure of a composition of simple trace structures $x_1|\cdots|x_k$, a trace σi must satisfy several conditions: first, σ must be a success of the composition; second, if i is not an input of x_j, then σi must be a success of x_j; third, there must be some component x_j for which i is an input, and for which σi is *not* a success. Such a sequence is called a *choke*. We can associate chokes with markings of the product net, that we will call *failure states*.

Definition 4. Let $M_1 \ldots M_k$ be processes with disjoint output sets, and let $M = M_1 \times \cdots \times M_k$. A *failure state* of $M_1 \ldots M_k$ is any reachable marking S of N (the net component of M) such that for some signal i,

- if $i \in O_j$ for some j, then there exists some $t_j \in T_j$ such that $L_j(t_j) = i$ and ${}^\bullet t_j \subseteq S$,
- for some j, $i \in I_j$ and there is *no* transition $t_j \in T_j$ such that $L_j(t_j) = i$ and ${}^\bullet t_j \subseteq S$.

A *failure sequence* of $M_1 \ldots M_k$ is any firing sequence π of N such that $\pi^\bullet$ is a failure state.

In other words, a failure state of the product net is any reachable marking where a given signal is enabled to be output (either by some process or by the environment), but some process that inputs that signal is not enabled to receive that signal. From any trace in the failure set of the composed trace structure, it is straightforward to construct a corresponding failure sequence of the product net. However, the existence of a failure sequence of the net does not imply the existence of a choke. This is because the net may be nondeterministic, in the

sense that there may be more than one firing sequence corresponding to any given trace. Thus, while one firing sequence may lead to a failure state, a different firing sequence yielding the same trace σi may lead to a success. Thus, by definition, the sequence σi is in the success set of the given process, and hence σi is not a choke. As an example, consider the net of figure 4. The trace ab is not a choke for this net, even thought we may reach a failure state after executing a, in which b is not enabled.

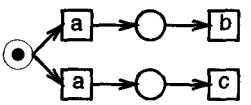


Fig. 4. A nondeterministic net.

For this reason, we will restrict ourselves to deterministic nets, such that, in any reachable marking there is at most one enabled transition labeled with any given signal. In this case, a given trace corresponds to at most one firing sequence, and hence the existence of a failure sequence in the product implies the existence of a choke.

Theorem 5. *Let* $M_1 \ldots M_k$ *be* deterministic *processes with disjoint output sets, and let* $M = M_1 \times \cdots \times M_k$. *The trace structure* $\mathcal{T}(M_1)| \cdots |\mathcal{T}(M_k)$ *is failure-free exactly when there is no failure state of* $M_1 \ldots M_k$.

Although it would be possible to test each process using reachability analysis to make sure it is deterministic, it is perhaps more practical to allow nondeterministic processes and accept the fact that some "false negatives" will be produced, in the sense that the product net may have failure sequences that do not correspond to an actual choke of the composed trace structure. In the sequel, we will allow nondeterministic nets, but will restrict our attention to closed systems, where the composition has no inputs. Composing the mirror of a specification with an implementation results in a closed system. Thus, we don't need to consider chokes caused by inputs from the environment.

5 Conformance testing using unfoldings

Testing for failure states could be accomplished by simply searching all of the reachable markings of the product net. The purpose here, however, is to use the unfolding method. Thus, we propose to construct the (truncated) unfolding of the product net, and to test it for the existence of failure states. It is not necessary in fact to explicitly construct the product net. Instead, we can construct the transitions of the product net "on-the-fly". Recall that the construction of the unfolding is a process of enumerating sets of concurrent places that might form the preset of some transition, and adding a new transition to the unfolding whenever a match is found. For each set of places generated in this way, we examine the set of processes to determine if some product transition can be

constructed from the given set of places (and also to determine what places might be added to the given set in order to match the presets of some transition). Except for this more general transition matching process, the construction of the unfolding is the same as described in [McM92b].

Given the truncated unfolding of the product net, the next task is to determine whether there are any failure states. Recall that for every reachable marking of the net, there is a configuration of the unfolding whose postset corresponds to that marking. A failure thus corresponds to a configuration in whose postset the output of a given signal x is enabled, while the input of x is not enabled.

5.1 Complexity of failure testing

Unfortunately, the condition for a failure to occur in a given configuration is not a strictly conjunctive one. A failure occurs when some set of places enabling output of x is present, while *no set* of places enabling the input of x is enabled. It is not difficult to show that testing the existence of a configuration satisfying such a condition is NP-complete.

The proof can be accomplished by reduction from 3-SAT. Suppose we have a set of positive literals $l_1, l_2, \ldots$ and negative literals $\bar{l}_1, \bar{l}_2, \ldots$, and we have a 3-CNF formula f of the form

$$(x_1 + y_1 + z_1)(x_2 + y_2 + z_2) \cdots (x_n + y_n + z_n)$$

Construct a labeled net with input a and output b defined as follows:

- There are places l_i and $\bar{l}_i$ corresponding to each positive and negative literal.
- There is initially a token on each $\bar{l}_i$.
- There is a transition from each $\bar{l}_i$ to l_i, outputing some unique signal.
- For each term $(x_i + y_i + z_i)$ in f, there is a transition inputing a with $\bar{x}_i$, $\bar{y}_i$ and $\bar{z}_i$ in its preset and empty postset. (Thus, this transition is enabled when the corresponding term is false).

If we view each marking of the net as a truth assignment to the literals, then a given marking is enabled to input a exactly when the formula f is *not* satisfied (*i.e.*, when some term is false). Thus, if we compose this process with a process that simply outputs a single a, there is a failure exactly when the formula is satisfiable. The resulting unfolding is linear size in n. Thus, we have a polynomial reduction from 3-SAT to failure detection in unfoldings. Further, the problem is clearly in NP, since we have only to guess a configuration and confirm that it has a failure. Hence the problem is NP-complete.

The NP-completeness of testing for failures in unfoldings is unfortunate, however it should be considered in light of the fact that overall problem being addressed (testing conformance of a composition of processes to a specification process) is PSPACE-complete, as are most problems involving reachability analysis of compositions of finite state processes. In practice, a branch-and-bound approach to the problem appears to be effective.

5.2 An algorithm for failure testing

The first step of the algorithm is to identify a set of concurrent places S in the unfolding that enable some transition that outputs some signal a. These sets can be found using the same procedure that is used to identify presets of transitions when constructing the unfolding. We then restrict our attention to the subnet of the unfolding that is *concurrent* with S, since we are interested only in configurations whose postsets contain S. A branch-and-bound procedure is then used to find a failed configuration within this subnet, if one exists (that is, a configuration whose postset does not enable any transition that inputs a).

Letting C initially be $\lceil S \rceil$, the local configuration of S:

1. If $C^\bullet$ enables no transition that inputs a, report failure.
2. Else, let $S' \subset C^\bullet$ be a set of places enabling some transition t that inputs a.
3. For every t' in the (restricted) unfolding whose preset contains some place in S':
   (a) add t' to C (let $C = C \cup \lceil t' \rceil$)
   (b) eliminate those net elements in conflict with t'
   (c) recurse

The situation when the above procedure is started is depicted in figure 5. If there is no transition labeled a in the given state, then clearly we have a failure. Otherwise, there is some set of places S' that enables the input of a. In order to obtain a failure (as a superset of C), we must use a transition that removes at least one place from S' (otherwise, the input of a remains enabled). Step three iterates through all the possible cases. With each recursion, the solution space is bounded by introducing a new transition to the configuration, and removing any parts of the unfolding that are in conflict with this transition. (Note – we maintain the invariant that no element in the restricted unfolding is in conflict with C. Thus, in step 3.1, $C \cup \lceil t' \rceil$ is guaranteed to be a legal configuration.)

As an example, consider the net of figure 6. Suppose we identify the place labeled S as enabling the output of a. The subnet between the dashed lines is concurrent with S. We restrict our attention to this subnet. Starting with the configuration $C = \lceil S \rceil$, we find that $C^\bullet$ contains a place labeled S' that, together with S, enables an a transition. Thus, this configuration is not a failure. However, we find that the place S' is in the preset of a transition labeled t'. Thus, we add t' to our configuration and recurse. (If there had been more that one t', we would have had to recurse once for each such transition. If there had been more than one S', however, we would need to consider only one of them.) After adding t' to C, we find that $C^\bullet$ contains a place S'' which also enables an a transition. Thus, this configuration is not a failure. Since no transitions (in the restricted net) remove S'', we are done. On the other hand, if S'' had not enabled an a to be input, this configuration would have represented a failure.

The worst-case complexity of this algorithm is exponential. However, exponential branching only occurs when an transition corresponding to an input of some process may be in conflict with two or more other transitions (and hence

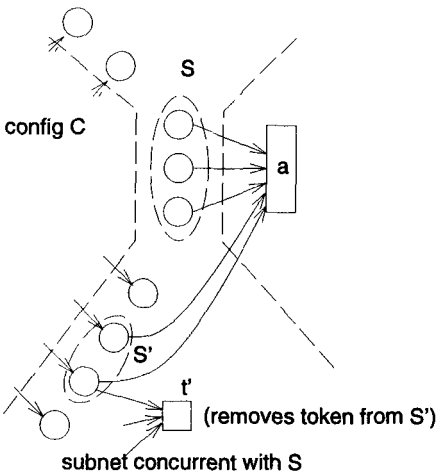


Fig. 5. Illustration of failure testing algorithm.

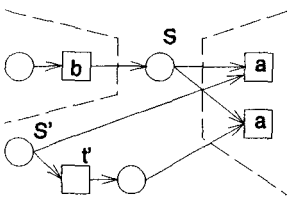


Fig. 6. Branch-and-bound example.

cause multiple recursions at step 3). In speed-independent circuits, such a conflict corresponds to a "hazard" – a situation that may result non-digital behavior or the given circuit element. Since a hazard usually indicates a design error, we may expect the case of *multiple* hazards to be fairly rare, hence we should only rarely observe exponential branching in the algorithm.

5.3 Implementation of the failure testing algorithm

As mentioned previously, every slice (concurrent set of places) S implicitly represents both a configuration $\lceil S \rceil$ and an open-ended subnet $\lfloor S \rfloor$ (see figure 2). When implementing algorithms on occurrence nets, it is convenient to represent an occurrence net implicitly by $\lfloor N \rfloor$, where N is the initial set of places, and to represent a configuration as $\lceil C \rceil$, where C is the postset of the configuration. Figure 7 shows an implementation of the failure testing algorithm in an ML-like notation, using this representation. This code uses a routine walk_net(N,f) that applies a function f to every transition in net N; a routine local(S,N) that returns the local configuration of a set of places S in net N; and a routine walk_presets(N,T,f) that applies function f to every (S, e) such that S is a

concurrent set of places in N matching the preset of some transition in T with label e. This routine is used to enumerate all the cases where an output of some process is enabled.

```
fun test_failure(N) =
    let fun recurse(S, e, C) =
        if some t ∈ T' s.t. •t ⊆ C and L(t) = e then
            let fun recurse_case(t') =
                if •t ∩ •t' ≠ ∅ then
                    recurse(S,e,local(t'•, C))
                else ()
            in walk_net(C − S,recurse_case)
        else raise failure(C)
    in walk_presets(N,output_transitions,fn S, e ⇒ recurse(S,e,local(S,N)))
```

Fig. 7. Implementation of failure testing algorithm.

The implementation used for the examples in the following section tests for failures "on-the-fly", as the unfolding is being built, rather than first constructing the unfolding and then testing for failures. This is done because, after a failure has occurred, the behavior of the circuit may become greatly more complex, resulting in a much larger unfolding than would be the case for a correct circuit. Although testing on-the-fly does not add greatly to the complexity of the implementation, the details are omitted here.

6 An example

We now consider two verification examples that show that using unfoldings can be much more effective than state space search, and in one case even more effective than BDD based methods [McM92a]. The first example is a speed-independent [Sei80] circuit designed to implement a distributed mutual exclusion (DME) protocol. The design is due to Martin [Mar85] and was corrected and verified by Dill [Dil88] using state space search methods.

The circuit consists of a ring of DME cells, as depicted in figure 8. Each cell communicates with one "user" via a request and an acknowledge signal (*ur* and *ua* respectively). These signals follow a four-phase protocol, in the sequence *ur*; *ua*; *ur*; *ua* (where implicitly, the first two transitions are rising and the second two falling). While the acknowledge signal is high, the user is enabled to access some resource. The purpose of the DME ring is to guarantee that only one user is enabled at a time. This is done by passing a token around the ring. When a given cell wants the token, it obtains it from its neighbor on the right by means of a four-phase handshake. Only the cell with the token may enable its user.

As in [Dil88], we will verify the DME ring by first implementing the DME cell as an abstract process (in this case a Petri net) that characterizes its in-

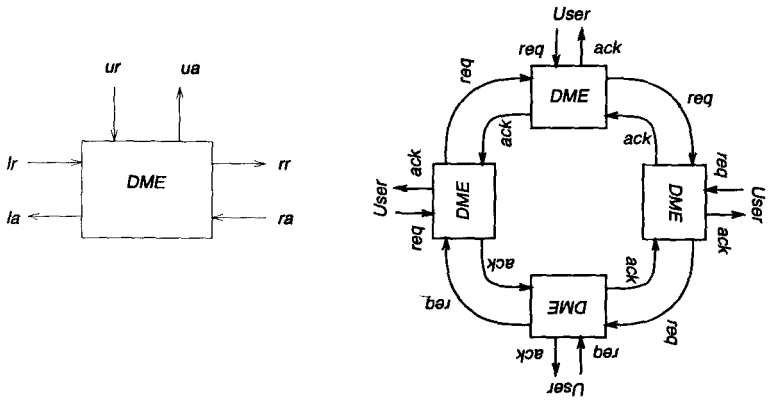


Fig. 8. DME cell and ring arbiter.

put/output behavior. We verify that a ring of these cells conforms to a specification that defines only the behavior at the user interfaces (and is also given by a Petri net). In the next step, the DME cell becomes the specification, and the implementation is given as a composition of small nets representing individual gates. This hierarchical approach is possible because in the trace theoretic framework, both implementation and specification are trace structures. We can infer that a ring of gate-level cells also implements the high level specification.

Figure 9a shows the Petri net that defines the high level specification, for the case of two users. It is easily extrapolated to any number of users. The net that models a single DME cell (at the abstract level) is shown in figure 9b. The unfolding method, as outlined in the previous section, was used to verify that a ring of n cells conforms to the high level specification for n users. It was found experimentally that the size of the unfolding of the product net is exactly $9n-2$ transitions and $25n-3$ places. That is, the amount of space used for the verification is *linear* in the number of cells. On the other hand, the number of reachable states is on the order of 10^n. For this reason, Dill only verified a ring of three cells (with 2,496 states). The unfolding size and total verification run time for DME rings of size 2-10 is shown in table 1. Run times are for an implementation in standard ML of New Jersey, running on a SPARCstation 5. Clearly in this case the ability to avoid exploring all interleavings of concurrent transitions is of substantial benefit. The branch-and-bound algorithm for failure testing, though exponential in the worst case, does not show exponential behavior here. For the case of 10 cells, the run time of about 7 seconds contrasts with a state space search of approximately 10^{10} states, which would clearly be impractical. On the other hand, verifying that the gate level cell implements the abstract cell can be easily accomplished using either unfolding or state space search.

The second example is a tree structured arbiter circuit (also found in [Dil88]). This arbiter consists of a tree of cells. Each cell obtains access to the resource from its parent and grants access to its two children via a four-phase handshake

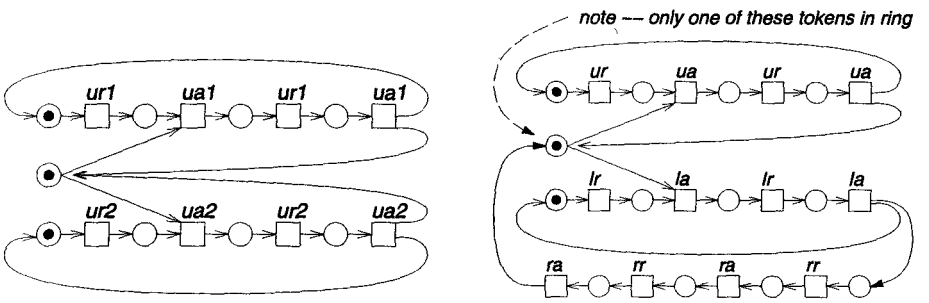


Fig. 9. High level specification for 2-user arbiter and abstract DME cell.

Table 1. DME ring results

n	transitions	places	time (s)
2	16	47	0.22
4	34	97	0.88
6	52	147	2.08
8	70	197	4.11
10	88	247	6.92

Table 2. Tree arbiter results

n	transitions	places	time (s)
2	12	36	0.15
4	28	84	0.66
6	38	114	1.09
8	60	180	3.34
10	70	210	4.22

protocol. Users obtain access from the leaves of the tree. At the root of the tree is a single "server" cell, that grants global access. The unfolding method was used to verify that a tree of such cells composed with a server cell conforms to the same mutual exclusion specification as in the previous example. In this case, we find that the size of the unfolding of the product net is not as simple a function of n, but is bounded from above by $11n$ transitions and $33n$ places. As before, the amount of space used is linear, although the number of reachable states is exponential. The space and time requirements for the verification are shown in table 2. Again, while the run time is superlinear, it does not show the worst case exponential increase. By contrast, in this case, not even a BDD based approach to state space search can verify large tree arbiters [McM94]. This is because the tree structure of the circuit does not allow for any linear variable order that keeps parents and children together. As a result the reached state set cannot be efficiently represented as a BDD. In [McM94], this problem is solved by using a more general representation called BDD trees. The current method has the advantage, however, that it does not require any structural information to be extracted from the circuit (in the form of a variable order).

7 Conclusions

An approach has been outlined for the hierarchical verification of speed-independent circuits that attempts to avoid the interleaving problem by modeling circuit components as Petri nets and using the technique of unfolding rather than state space search. Toward this end a means of representing trace structures by nets

was devised. Existence of a failure in the composition of such processes was cast as a condition on the state space of a product net. Testing this condition on the unfolding was found to be an NP-complete problem, however a practical branch-and-bound solution was found that appears to be well suited to the verification of speed-independent circuits. The technique was found to be more effective than state space search methods in two examples, in the sense of being able to verify significantly larger versions of scalable designs. In one case, verification using unfoldings was found to be scalable, whereas using BDD's to represent the reached state set it is not.

The closest technique described in the literature to the unfolding method is the "behavior structures" method of Probst [PL90, PL91]. In this case, the system behavior is represented by a tree of partially ordered sets of events. Verification of the DME ring was carried out using this method, and was found to use both linear time and linear space. However, this method is less automatic in the sense that it requires the user to construct a "behavior structure" describing the entire system, whereas in the present case the system is modeled directly in terms of its components. The user provides only the circuit and its specification. Esparza describes the use of unfoldings for model checking a modal logic of nets [Esp93]. This work does not address hierarchical verification or asynchronous circuits specifically. It should also be noted that, for the DME example, the hierachical verification method presented here is considerably more efficient than the original method of [McM92b], which unfolds a net corresponding to a gate level model of the entire circuit.

Other methods exist to avoid exploring all interleavings in state space search by pruning the state graph [Val89, Val90, God90, GW91, YTK91]. Although these methods have not been shown to be effective in verifying asynchronous circuits, it is possible that they could also be applied to the problems addressed here. These methods require the identification of a "persistent set" of transitions in each state, where every run from that state is equivalent to a run begining with a transition in the persistent set. Such sets are difficult to identify in speed-independent circuits, because it is difficult to rule out the possibility of any given enabled transition associated with a gate output being disabled by a signal transition at the gate's inputs. The unfolding method is particularly suited to the verification of speed-independent circuits because it does not require the identification of persistent sets.

The unfolding method is clearly limited in several respects. First, it almost certainly can be applied only to control circuits, and not to data circuits. Unfoldings address only the problem of interleavings, and not other sources of the state explosion problem. Second, most asynchronous designs are not entirely speed-independent – they rely on some known relationships between the delays of various elements. To apply unfoldings to such circuits, a version of the technique based on a timed trace theory, with some form of time-constrained nets as the model, would be required. This is an area for future study.

References

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth LICS*, June 1990.

[Dil88] D. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. Technical Report 88-119, CMU, Comp. Sci. Dept., 1988.

[Esp93] J. Esparza. Model checking using net unfoldings. In *TAPSOFT '93*, Orsay, France, April 1993.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Workshop on Computer Aided Verification*, 1990.

[GW91] P. Godefroid and P. Wolper. A partial approach to model checking. In *LICS*, 1991.

[Mar85] A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In *1985 Chapel Hill Conf. on VLSI*, pages 245–260, 1985.

[McM92a] K. L. McMillan. Symbolic model checking: an approach to the state explosion problem. Technical Report 92-131, CMU, Comp. Sci. Dept., 1992.

[McM92b] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Fourth CAV*, 1992.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[McM94] K. L. McMillan. Hierarchical representations of discrete functions, with application to model checking. In *Sixth CAV*, Stanford, CA, 1994.

[McM95] K. L. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. *Formal Methods in System Design*, 1995. to appear.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.

[Pet77] J. L. Peterson. Petri nets. *Computing Surveys*, 9(3):223–248, 1977.

[PL90] D. K. Probst and H. F. Li. Using partial order semantics to avoid the state explosion problem in asynchronous systems. In *CAV*, 1990.

[PL91] D. K. Probst and H. F. Li. Partial order model checking: A guide for the perplexed. In *Third CAV*, pages 405–416, July 1991.

[Sei80] C. L. Seitz. System timing. In Carver Mead and Lynn Conway, editors, *Introduction to VLSI Systems*, pages 218–262. Addison-Wesley, 1980.

[Val89] A. Valmari. Stubborn sets for reduced state space generation. In *10th Int. Conf. on Application and Theory of Petri Nets*, 1989.

[Val90] A. Valmari. A stubborn attack on the state explosion problem. In *Workshop on Computer Aided Verification*, 1990.

[YTK91] Tomohiro Yoneda, Yoshihiro Tohma, and Yutaka Kondo. Acceleration of timing verification method based on time Petri nets. *Systems and Computers in Japan*, 22(12):37–52, 1991.