An Improvement in Formal Verification

Gerard J. Holzmann
Doron Peled
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Critical safety and liveness properties of a concurrent system can often be proven with the help of a reachability analysis of a finite state model. This type of analysis is usually implemented as a depth−first search of the product state−space of all components in the system, with each (finite state) component modeling the behavior of one asynchronously executing process. Formal verification is achieved by coupling the depth−first search with a method for identifying those states or sequences of states that violate the correctness requirements.

It is well known, however, that an exhaustive depth−first search of this type performs redundant work. The redundancy is caused by the many possible interleavings of independent actions in a concurrent system. Few of these interleavings can alter the truth or falsity of the correctness properties being studied.

The standard depth−first search algorithm can be modified to track additional information about the interleavings that have already been inspected, and use this information to avoid the exploration of redundant interleavings. Care must be taken to perform the reductions in such a way that the capability to prove both safety and liveness properties is fully preserved. Not all known methods have this property. Another potential drawback of the existing methods is that the additional computations required to enforce a reduction during the search can introduce overhead that diminishes the benefits. In this paper we discuss a new reduction method that solves some of these problems.

Keywords: I.3, I.6, III.2, IV.5

Proceedings FORTE 1994 Conference, Bern, Switzerland, 4– 7 October 1994.

August 20, 1994

An Improvement in Formal Verification

Gerard J. Holzmann
Doron Peled
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

The depth-first search algorithm that is used for on-the-fly reachability analyses can explore many execution sequences that are not strictly required to prove the safety and liveness properties of a concurrent system. In the last few years, several proposals have been made for revised search algorithms that can avoid some or all of this redundancy, e.g., [V90], [GW91], [HGP92], [V93], [P93].

The methods that have been studied so far can be classified as 'dynamic reduction methods.' They attempt to compute mostly at runtime (i.e., during the search) which parts of the reachability analysis are redundant and can be skipped. Unavoidably, the additional computations also consume resources: they require memory to store additional data structures, and they require CPU time to discover the redundancies. This overhead reduces the amount of improvement that can be achieved. In some cases, the costs of improvement outweigh the gains, which means that the unoptimized full search can sometimes outperform the 'optimized' reduced search.

This paper considers the feasibility of performing the computations before the search, instead of during the search. We prove for one such reduction strategy that it preserves both the safety and the liveness properties of a concurrent system. The new reduction strategy is therefore generally usable for linear time temporal logic (LTL) model checking.

Section 2 discusses the main concepts needed for a definition of the static reduction method. Section 3 introduces the algorithm itself, and Section 4 contains its proof of correctness. Section 5 discusses an implementation of the static reduction method, as an experimental addition to the verification tool SPIN [H92]. Section 6 contains an evaluation of the performance of this implementation, and a comparison against both the classic search method and an existing dynamic reduction method, all implemented as part of the same verification system. Section 7 summarizes the results.

2. DEFINITIONS

We consider any verification problem that can be formalized as a reachability analysis problem in a finite labeled transition system (LTS). This specifically includes the problems of proving safety, liveness, and linear time temporal logic properties for any finite state concurrent system.

An **LTS** is defined as a triple $\{S, s_0, T\}$, where S is a finite set of states, s_0 is a distinguished initial state in S, and T is a finite set of transitions, with $T \subseteq (S \times S)$. In a simple form, an LTS can be used to formalize the behavior of a single sequential process. It can also formalize the combined behavior of a finite number of interacting and asynchronously executing sequential processes. Each transition of the LTS then corresponds to the execution of a specific atomic statement within one of the processes, in accordance with a standard interleaving semantics of concurrency. The LTS can be represented by a graph with nodes corresponding to the states in S and directed edges corresponding to the transitions in T. A connected path through this graph then defines the effects of a possible execution in the underlying concurrent system. There will be at least one path through the graph for every possible way in which the execution of process statements could be interleaved in time.

Given a transition $t \in T$ in an LTS, we will use the notation $Label(t)$ to refer to the process statement that is represented by transition t, and we will use $Pid(t)$ to refer to the sequential process that contains the statement $Label(t)$. Without loss of generality, we assume that the mapping from transitions to process statements is unique. The reverse mapping will, in general, not be unique.

The semantics of a statement $a = Label(t)$ are defined by two functions *Cond* and *Act*, where

$Cond(a)$ is the subset of S where a is enabled (or 'executable' [H92]), and
$Act(a,s)$ is that state of S that is reached when a is executed in a given $s \in Cond(a)$.

Normally, a statement in a sequential process is 'enabled' or 'executable' only if it is pointed to by the current program counter of the sequential process that contains that statement. In a concurrent system, however, we can define additional constraints on the enabledness or executability of statements. A message send operation, for instance, can be defined to be enabled only if also the destination message buffer is non-full, and a message receive operation can be defined to be enabled when also the source message buffer is non-empty.

Two statements a and b are defined to be **independent** at state $s \in S$, written as $\{a,b\} \in Ind(s)$, if and only if the following five conditions are met:

(1) $s \in Cond(a)$, i.e., statement a is enabled in s,
(2) $s \in Cond(b)$, i.e., statement b is enabled in s,
(3) $Act(a,s) \in Cond(b)$, i.e., the execution of a cannot disable b,
(4) $Act(b,s) \in Cond(a)$, i.e., the execution of b cannot disable a,
(5) $Act(b,Act(a,s)) = Act(a,Act(b,s))$, i.e., the effect of executing a followed by b is indistinguishable from that of executing b followed by a.

Note that two statements from the same sequential process, i.e., with $Pid(a) = Pid(b)$, can not be independent. If the two statements are executed sequentially, they cannot be simultaneously enabled. If they appear together in a single selection, the execution of either one statement will disable the other. Two statements from distinct sequential processes can be independent under certain conditions. Two send operations on distinct message queues will in general be independent, but two send operations on the same message queue will not. The send operation that executes first may disable the second if its message fills the queue to capacity, which violates requirements (3) and (4). In addition, the order in which the two statements are executed can be distinguished by the order in which the messages appear in the destination queue, which violates requirement (5).

Statements a and b are defined to be **globally independent** if and only if they are independent in every possible state where they are simultaneously enabled:

(6) $s \in (Cond(a) \cap Cond(b)) \rightarrow \{a,b\} \in Ind(s)$.

Note that a and b are trivially globally independent when $Cond(a) \cap Cond(b) = \emptyset$. Two assignment statements from two distinct sequential processes, i.e., $Pid(a) \neq Pid(b)$, that access only local variables within each process, will in general also be globally independent.

Because it is known that both safety and liveness properties can be expressed by next-time-free linear-time temporal logic (LTL) formulae [W83], we will focus on a method for proving the satisfiability of LTL formulae (cf. [V90], [V93], [P93]). The **LTL formulae** we consider may contain boolean propositions on system-states, the boolean operators $\wedge$, $\vee$, ! (not), and the temporal operators $\Box$ (always), $\Diamond$ (eventually), and U (until), but not the temporal operator $\bigcirc$ (next-time).

Wolper [W83] showed that any next-time-free LTL formula can be formalized as a nondeterministic Büchi Automaton with a predefined initial state, and a finite set of acceptance states. The *transitions* in the Büchi Automaton carry predicate labels, each of which represents a boolean proposition. In our case, the boolean propositions can refer only to the (global) system-state of the labeled transition system for which the LTL formula formalizes a property. The Büchi Automaton itself can be represented by an LTS with predefined acceptance states. The satisfaction of an LTL formula can now be proven by detecting acceptance cycles in the synchronous product of two labeled transition systems: one representing the concurrent system and one representing the Büchi Automaton (e.g., [CVWY92]). The absence of acceptance cycles can similarly prove that the LTL formula cannot be satisfied [H92].

The **synchronous product** $F \times G$ of a labeled transition system F, representing a concurrent system, and a Büchi Automaton G, derived from a next-time-free LTL formula, is defined as follows. Let $F = (S_F, f_0, T_F)$ and $G = (S_G, g_0, T_G)$. Each state of the synchronous product $F \times G$ is a pair (f,g), with $f \in S_F$ and $g \in S_G$. Each transition, similarly, is a pair (v,w), with $v \in T_F$ and $w \in T_G$. We define the LTS for the synchronous product $F \times G$ recursively as follows. The initial state of $F \times G$ is (f_0, g_0). For each

state (f,g) there is a successor state (h,k), reachable via transition (v,w), if and only if:

(1) $v=(f,h)\in T_F$, i.e., h is a successor of f via v in F,
(2) $w=(g,k)\in T_G$, i.e., k is a successor of g via w in G, and
(3) The boolean proposition defined by $Label(w)$ is *true* in state $f\in S_F$.

A statement a in F is said to be **observable** by Büchi Automaton G if there exists a label in G for which the corresponding proposition can have a different truth-value in at least one system-state $s\in Cond(a)$ and in $Act(a,s)$. The statement a can now be said to be

- **Safe** if a is non-observable to G and globally independent from every b with $Pid(a)\neq Pid(b)$, and
- **Conditionally Safe** for condition $P(s)$, if a is safe in every state s where $P(s)$ holds.

The reduction algorithm that we will describe in the next section relies on the fact that the safety or conditional safety of statements can in many cases be determined statically.

3. REDUCTION ALGORITHM

We first consider the standard depth-first search algorithm that implements the generation of the labeled transition system F from a specification of a concurrent system. We then consider how this search can be extended to generate the synchronous product $F\times G$, where G is a Büchi Automaton that encodes an LTL formula, and to detect the existence of acceptance cycles in that product.

The initialization of the search is illustrated in Figure 1a. First, the basic transition structure of the concurrent system is obtained and optimized. The optimization step, can, as we shall argue, also include a precomputation of independence relations, with a static identification of all safe and conditionally safe process-statements. Two sets of states are then initialized with the predefined initial system state s_0: the *Statespace* and the *Stack*. The search begins with a call of the depth-first search routine, *Dfs*( ), with parameter 1. The relevance of the parameter will become clear shortly.

```
1 start_search(s0)
2 { derive and optimize transition structures
3   enter s0 into Statespace;
4   push  s0 onto Stack;
5   Dfs(1);  /* see Figure 1c */
6 }
```

Figure 1a – Initialization

Figure 1b first shows the expansion step for process statements, in routine *dfs*( ) (note: not the routine from line 5). In the absence of a Büchi Automaton, the calls on lines 5 and 16 could both be implemented as calls on `dfs(N)`.

```
 7 dfs(N)
 8 { s = top(Stack);
 9   for each sequential process i
10   {    nxt = all transitions in F enabled in s with Pid(t)=i
11        for all t in nxt
12        {    s' = successor of s after t;
13             if {s',N} NOT in Statespace
14             {    enter {s',N} into Statespace;
15                  push s' onto Stack;
16                  Dfs(N);
17   }    }    }
18   pop s from Stack
19 }
```

Figure 1b – Expansion Step for the Sequential Processes

In the general version of the verification algorithm, however, the calls on lines 5 and 16 invoke the routine shown in Figure 1c, which implements the expansion step for the transitions in the Büchi Automaton. The state of the Büchi Automaton is part of compound system state s. Because the transitions in the Büchi Automaton represent boolean propositions from an underlying LTL formula, a transition $t\in T_G$ in Büchi Automaton G will only be enabled if and only if proposition $Label(t)$ holds. The synchronous coupling of

system *F* and Büchi Automaton *G* is achieved by alternating the calls to `Dfs(N)`, on line 16, and `dfs(N)` on line 28. Each pair of subsequent calls, explores one synchronous transition of $F \times G$.

```
20 Dfs(N)
21 { s = top(Stack);
22   nxt = all transitions in G enabled in s; /* the Büchi Automaton */
23   for all t in nxt
24   {    s' = successor of s after t;
25        if {s',N} NOT in Statespace
26        {    enter {s',N} into Statespace;
27             push s' onto Stack;
28             dfs(N);
29   }    }
30   pop s from Stack
31 }
```

Figure 1c – Interleaved Transitions of Büchi Automaton

Figure 1c shows only the basic expansion step without the extra hooks that are required to detect the presence of acceptance cycles in the synchronous product of concurrent system and Büchi Automaton. To enable also the detection of acceptance cycles, we can check for every reachable acceptance state in *G* if that state is also reachable from itself. We do so with a second depth-first search, in post-order, in a separate state space. Two separate values for parameter `N` serve to indicate in which part of the search the algorithm operates. To initiate the second search, we include four extra lines between lines 28 and 29 of Figure 1c, as illustrated in Figure 1d. If the seed state is reachable from itself this can be detected and reported at line 24, as illustrated by lines 24a-d in Figure 1d.

```
20 Dfs(N)
21 { s = top(Stack);
22   nxt = all transitions in G enabled in s; /* the Büchi Automaton */
23   for all t in nxt
24   {    s' = successor of s after t;
24a       if N == 2 and s' == seed
24b       {    report acceptance cycle
24c            return
24d       }
25        if {s',N} NOT in Statespace
26        {    enter {s',N} into Statespace;
27             push s' onto Stack;
28             dfs(N);
28a            if N == 1 and s is an accepting state in G
28b            {    seed = s
28c                 dfs(2)
28d            }
29   }    }
30   pop s from Stack
31 }
```

Figure 1d – Extension for Cycle Detection

A description, and correctness proof, for this method of cycle detection was given in [CVWY92]. The algorithm generates at least one example of an acceptance cycle, if one or more such cycles exist. It is not guaranteed to generate *all* such cycles. If, however, the Büchi Automaton is used to formalize an undesirable behavior, i.e., the violation of a correctness requirement, a proof of either the existence or the absence of acceptance cycles that satisfy the claim is always sufficient for a conclusive verification result.

Note that when the existence of an acceptance cycle is discovered, its complete traversal is contained in the *Stack*, and can be generated as a counter-example to the correctness claim.

Static Reduction

To implement a static reduction technique, it suffices to modify only the algorithm from Figure 1b, since the safety of transitions applies only to the transitions in the sequential processes, not to those of the Büchi Automaton. The change is illustrated in Figure 1e. The aim of the reduction method is to find the smallest set of transitions that will suffice to perform the expansion (given that we want to preserve both safety and liveness properties). Clearly, the expansion cannot be complete unless for every transition selected, we also select all those simultaneously enabled transitions that that are *not* independent from it. This means that if we select a, we must minimally also select all simultaneously enabled transitions b with $Pid(b)=Pid(a)$ (cf. line 10 in Figure 1e).

In the static reduction method we try to identify at least one process that can execute only safe, or conditionally safe, transitions. Such a process can be found by a prescan of the processes. In Figure 1e, this critical step is performed on line 8a and is used to re-order the processes in such a way that processes that perform only (conditionally) safe transitions can be selected first for the expansion step on line 9. If the expansion succeeded (more about this below) we can ignore the (independent) transitions from all other processes by breaking out of the loop over processes on line 16f. The ordering step itself introduces virtually no runtime overhead. In the implementation discussed in Section 5, for instance, it is implemented by a table-lookup for unconditionally safe transitions, and by the evaluation of a precomputed boolean condition for conditionally safe transitions.

A check is added on lines 16a-b, to see if the last transition explored returned the search to a state s' that is already contained in the search stack or not. If there is at least one such transition, the value of a local boolean variable *NotInStack* is set to *false*. Once all transitions of the process have been explored, the values of *NotInStack* and *AtLeastOneSuccessor* are inspected. The reduction attempt fails unless all transitions explored for the current process have produced successor states that are currently *not* contained in *Stack*. If this requirement is not met, the algorithm will try to make another selection of transitions, by moving to the next process in the outer for-loop. In the worst case, this will mean that the reduced expansion step will explore all enabled transitions, just as it did in Figure 1b.

The condition on line 16e that has to be fulfilled for the reduction attempt to be considered successful is known as the **reduction proviso**. The need for such a proviso was first recognized by Valmari in [V90]. The version of the proviso used here was first proposed in [P93]. A weaker version of the same test, for the preservation of safety properties only, was discussed in [HGP92]. In the next section we will show that the stronger proviso from [P93] guarantees the preservation of both safety and liveness properties.

```
 7 dfs(N)
 8 { s = top(Stack);
 8a  order processes; /* using safety as ordering principle - see text */
 9   for each sequential process i
 9a  {    boolean NotInStack = true
 9b       boolean AtLeastOneSuccessor = false
10       nxt = all transitions t in F enabled in s with Pid(t)=i
11       for all t in nxt
12       {    s' = successor of s after t;
13            if {s',N} NOT in Statespace
14            {    enter {s',N} into Statespace;
15                 push s' onto Stack;
16                 Dfs(N);
16a           } else if s' in Stack  /* reduction proviso */
16b                NotInStack = false
16c           AtLeastOneSuccessor = true
16d      }
16e      if AtLeastOneSuccessor ∧ NotInStack
16f           break /* from the loop over processes */
17   }
18   pop s from Stack
19 }
```

Figure 1e – Reduced Expansion Step

The tests on line 16a and 16e introduce virtually no overhead to the algorithm.

4. PROOF OF CORRECTNESS [skip on first reading]

We will give the main proof argument that supports the correctness of the reduction algorithm. The remaining steps that are required for a rigorous proof are only briefly indicated.

An *execution sequence* σ of an LTS can be defined either as a sequence of transitions or as the sequence of states that is traversed by these transitions. Let $Eq(\sigma)$ be the set of all execution sequences that can be obtained from σ by zero or more permutations of adjacent, globally independent, transitions. For each sequence in this set we can define the **distance** to σ as the smallest number of permutations that must be performed to retrieve σ. (This distance can be eithe finite or infinite.)

Any sequence ρ that equals a finite prefix of at least one sequence in $Eq(\sigma)$ is called a **permuted prefix** of σ. Let $PP(\rho,\sigma)$ be the set of sequences in $Eq(\sigma)$ that contain ρ as a prefix. Let $PP'(\rho,\sigma)$ further be the subset of those sequences in $PP(\rho,\sigma)$ that have the shortest distance to σ. Note that the sequences in this set differ from σ in at most a prefix of finite length. For each such sequence, therefore, we can define a finite prefix ρ', such that the remainder of the sequence (after the deletion of ρ') equals σ. This prefix, which can be longer than ρ, is called the **minimal stable extension** of ρ in σ.

A **generalized permuted prefix** ρ of an execution sequence σ is finite execution sequence that can be transformed into a permuted prefix of σ by omitting zero or more non-observable transitions.

To prove the correctness of the reduced search algorithm, we first prove the following Lemma.

Lemma – At each state that is reached during the search, the reduced search algorithm generates at least one generalized permuted prefix ρ for every execution sequence σ that can start from that state. □

Proof – The proof is by induction on the order in which states are removed from the depth-first stack in the reduced search algorithm.

[1.] For the induction basis, consider the first state that is removed from the stack in the reduced search algorithm. There are two cases to consider, depending on the number of enabled transitions in that state.

[1.1.] The state has no enabled transitions, and thus no successor states. In this case there exist no further executions from this state, and the Lemma holds.

[1.2.] The state has enabled transitions. All these transitions must have returned the search to previously visited states: they cannot be new states because such states would have been removed from the stack before the current one. Since no states were previously removed from the stack, all previously visited states are still contained in the stack. The reduction proviso from the reduced search algorithm will in this case force a complete exploration of all enabled transitions from this state (line 16a, Figure 1e). This set includes the first transition a from σ. This transition a is a generalized permuted prefix of length one. The Lemma therefore holds for this case.

[2.] Next, we must show that if the Lemma holds for the first N states that are removed from the stack, it necessarily also holds for the $(N+1)$-th state. Let s be that state. There are again two cases to consider.

[2.1.] The set of enabled transitions in s does *not* contain a true subset of (conditionally) safe transitions that includes all the enabled transitions for one sequential process, and none of which leads to a successor state on the stack. In this case, the reduced search algorithm explores all enabled transitions from s and the Lemma holds by the same construction as was used in the proof of step [1.2].

[2.2.] The set of enabled transitions in s *does* contain a true subset of (conditionally) safe transitions that includes all enabled transitions for one sequential process, and none of which leads to a successor state on the stack. Call that subset x, and call the (non-empty) set of all remaining transitions y. The reduced search algorithm explores only the sequences that start with a transition from x.

First note that any transition in x forms a generalized permuted prefix of length one for σ. That is: each such transition either appears in σ after a finite number of globally independent transitions, or it does not appear in σ and is globally independent of all transitions that do appear.

There are two cases to consider.

[2.2.1.] If σ starts with a transition from x, the Lemma again holds.

[2.2.2.] Next, consider the case where σ starts from state *s* with a transition from *y* and reaches successor state *s′*. We distinguish two further sub-cases.

[2.2.2.1.] First, consider the case where σ starts with a transition from *y* and where that transition is globally independent of all transitions in σ. In this case, none of the transitions in σ can have been disabled by the execution of the globally independent transition from *y*, and the transition itself forms a generalized permuted prefix of length one. The transition from *y* is now itself a non-observable transition that could be deleted from generalized permuted prefix to obtain the (empty) permuted prefix of σ. The Lemma therefore holds for this case.

[2.2.2.2.] Next, consider the case where σ starts with a transition from *y* and where that transition is *not* globally independent of all transitions in σ. Let *a* be the first transition in σ, and *b* a transition from the chosen set *x* that appears also in σ. (The case where *b* does not appear in σ was already covered in the second half of proof step [2.2.].) Call *s′* the state that is reached after the execution of *b*. We can now find a minimal stable extension of *b* in σ, as defined above, which includes all the occurrences of transitions in the prefix of σ that ends at the first occurrence of *b*. Call that prefix ρ. Further, call σ′ the suffix of σ that follows first occurrence of *b*. Then the sequence ρ.σ′ is equal to a copy of σ from which this first occurrence of *b* is deleted. The prefix ρ is then a generalized permuted prefix of the sequence ρ.σ′ that starts at state *s′*. But then, the prefix *b*.ρ must be a generalized permuted prefix of σ, which starts at state *s*, which means that the Lemma also holds for this case. This completes the proof of the Lemma. □

The Lemma can be shown to imply that for every execution sequence σ, the reduced search algorithm explores at least one execution sequence that becomes equivalent to σ when a finite number of non-observable transitions are deleted from it. (This proof step is not detailed here.) Next we must show that this property is sufficient for the completeness of the search itself. To do this, we must take a closer look at the synchronous product of a concurrent system and a Büchi Automaton.

Given a concurrent system *C* and a Büchi Automaton *M*, we can construct an ordered set of predicates $P(M)$ with one predicate for each boolean proposition on the states of *C* that appears in *M*. For each reachable system state of *C*, each predicate in $P(M)$ then uniquely defines a boolean value, and the set $P(M)$ similarly defines a unique vector of boolean values. For given $P(M)$, an execution sequence of *C* corresponds to a sequence of boolean value vectors. Call that sequence 'the vector-sequence induced by *M*.'

We define two execution sequences to be ***M*-equivalent**, for given Büchi Automaton *M*, if and only if the corresponding vector-sequences induced by *M* are equal up to stuttering, i.e., if the two sequences are equal when each series of two or more consecutive occurrences of the same value vector *v* is replaced by a single occurrence of *v*.

The Lemma implies that the reduced search algorithm generates at least one *M*-equivalent sequence for each execution sequence of the concurrent system. The intuition for this is that all non-observable transitions correspond to stuttering steps. (This proof step is not further detailed here.) The correctness of the reduced search algorithm can now be formalized in the following theorem.

Theorem – If there exist acceptance cycles in the synchronous product of a Büchi Automaton and a concurrent system, the reduced search algorithm will detect at least one of these cycles. □

Proof – by the Lemma and the fact that the set of sequences satisfying a next-time-free LTL formula is closed under stuttering [L83]. The reduced search generates at least one *M*-equivalent sequence for each complete sequence that satisfies the LTL formula. All sequences that satisfy the LTL formula are detected in the non-reduced depth-first search as acceptance cycles in the synchronous product of the corresponding Büchi Automaton and the concurrent system (e.g., [W83][CVWY92][H92]). Therefore, if at least one *M*-equivalent sequence for such a satisfying sequence is generated in the reduced search, at least one acceptance cycle is necessarily detected. □

5. IMPLEMENTATION

For a sample implementation of the static reduction technique in the verification system SPIN and its specification language PROMELA [H92], we identified five types of statements that can be marked statically as unconditionally safe when they appear separately, and conditionally safe when they appear as guards in selection structures.

(1) Any access to exclusively local variables. Any atomic process-statement that reads or writes exclusively objects that are non-observable to other processes, is also non-observable to the PROMELA *never* claim (which formalizes the Büchi Automaton).

(2) Any receive operation on a message queue q, provided that no more than one process can either receive messages from q or test the contents or length of q. We mark such a queue with a special status: *exclusive receive-access*. Exclusive receive-access implies that a *never* claim contains no propositions on the contents of q.

(3) Any send operation on a message queue q, provided that no more than one process can send messages to q, or test the contents or length of q. We say that such a queue has *exclusive send-access*. Exclusive send-access implies that a *never* claim contains no propositions on the contents of q.

(4) The boolean test *nfull*(q), that returns *true* when message queue q is currently non-full, and false otherwise, provided that the statement is performed by a process that has exclusive send-access to that queue.

(5) The boolean test *nempty*(q), that returns *true* when message queue q is currently non-empty, and false otherwise, provided that the statement is performed by a process that has exclusive receive-access from that queue.

The statements of types (1)-(5) are *conditionally safe* if they do appear as guards in selection structures. The condition for the conditionally safe statements is defined as the logical *and* combination of the following clauses for each type of guard: (1) *true* (i.e., these statements contribute no additional constraints), (2) and (5) *nempty*(q), and (3) and (4) *nfull*(q). Note that statements of type (2-5) can only contribute constraints of two statically determined types.

We extended the PROMELA grammar with the two new primitives *nfull*( ) and *nempty*( ), referred to in (4) and (5). A simple grammar rule in the parser prevents attempts to include negations of these two tests.

The observability of the effect of statements to the propositions of the Büchi Automaton (i.e., the PROMELA *never* claim) is already guaranteed by the scope rules of PROMELA: in the absence of remote referencing, the *never* claim can only refer to global objects in the specification. All safe and conditionally safe operations are therefore necessarily non-observable to the claim. Any reference to a queue, for instance, breaks the exclusive access status of that queue, and automatically marks the send or receive operations as observable, and therefore non-safe.

Because PROMELA allows the dynamic creation of a finite number of processes, it is not always possible to determine a priori which processes will be able to access which queues. Exclusive send and receive access, in our implementation, is therefore entered into the PROMELA specification as a logical assertion, which can be checked at runtime. The Appendix shows an example of a complete PROMELA specification for a leader election protocol from [DKR82], with the exclusive send and receive assertions added. It can easily be shown that the validity of an assertion of this type can be proven by both the non-reduced and the reduced search, even when the reduction is based on an invalid assertion of this type. The intuition behind this is that the reduced search can only *permute* globally independent statements, it cannot prevent their execution alltogether. Therefore, at least one send or receive operation that violates an exclusive access assertion will eventually be executed in the reduced search, though perhaps at a different place then in the non-reduced search. There is, of course, also the possibility that the reduced search is stopped on the detection of an acceptance cycle *before* the violation of an exclusive access assertion can be demonstrated. In that case, however, the search has already reached its goal: it has detected the existence of at least one error (i.e., an acceptance cycle). If the violation of an exclusive access assertion can be demonstrated first, our implementation also reports an error (i.e., an assertion violation), which in that case means that the reduction itself was invalid.

For the correctness of the reduction algorithm itself it must be demonstrated that if there exist one or more acceptance cycles in $F \times G$, the reduced search algorithm will always report at least one of them. The proof of this property is given in [P94]. Note that it is not guaranteed, neither for the reduced nor for the standard algorithm, that *all* acceptance cycles will be reported.

6. PERFORMANCE

We have measured the performance of the new reduction algorithm on five sample protocols, including a best-case example, a worst-case example, and two average protocol applications produced independently by users of SPIN [H92]. The performance is compared both to the non-reduced 'classic' verification algorithm, and to an existing dynamic reduction method based on Godefroid's sleep-set method [GW91], [HGP92], which is publicly available (in binary form) via anonymous FTP from the University of Liege as and extension to SPIN. In the comparisons, it should be observed that the dynamic reduction method preserves only safety properties, while the other two methods preserve both safety and liveness properties, and thus provide a stronger and more general model checking system. There exist other reduction methods that preserve both safety and liveness properties (e.g., [V90], [V93]), but at the time of writing no implementation of these methods was available for these comparisons.

Table I – Measurements

Protocol	Algorithm	States	Transitions	Time(sec.)	Memory (Mb)
Best-Case	Non-Reduced	100,001	450,002	13.2	4.3
	Static Reduction	47	47	(<0.1)	1.0
	Dynamic Reduction	47	47	0.1	1.4
Worst-Case	Non-Reduced	100,001	450,002	14.5	5.0
	Static Reduction	100,001	450,002	16.7	5.1
	Dynamic Reduction	100,001	450,002	84.5	5.3
Tpc	Non-Reduced	3,918,286	11,762,426	630.6	268.4
	Static Reduction	391,534	466,753	30.6	26.2
	Dynamic Reduction	267,204	295,395	131.4	18.9
Snoopy	Non-Reduced	91,920	305,460	14.4	11.5
	Static Reduction	16,279	23,532	1.7	3.2
	Dynamic Reduction	7,158	8,459	6.8	2.6
Pftp	Non-Reduced	417,321	1,244,865	73.2	62.3
	Static Reduction	53,244	67,901	6.8	9.3
	Dynamic Reduction	125,718	163,459	105.5	20.6
Leader	Non-Reduced	45,885	185,032	8.1	9.6
	Static Reduction	79	79	0.1	1.1
	Dynamic Reduction	79	79	0.2	1.4

Not all types of reduction can be computed with a static algorithm. For instance, if multiple sequential processes share access to the elements of a data array, the precise values of the array indices may only be known dynamically, which makes it impossible for a static algorithm to determine if the access of any given element is safe. Because every reduction that can be made with a static algorithm can also be made with a dynamic algorithm, we should expect that dynamic algorithms can achieve greater reductions than static ones in terms of the numbers of states and transitions explored. If the bottom-line commodities of run-time and memory used are measured, however, the results are less predictable, and it should be possible for a static algorithm, with lower overhead, to defeat a dynamic one.

The best and the worst case examples are artificial cyclic models, containing only local and only global statements, respectively. *Tpc* is a model of a telephone switch, specified in 279 lines of PROMELA. *Snoopy* is a model of a cache coherence protocol specified in 255 lines of PROMELA. *Pftp* is a version of the file transfer protocol of 204 lines from [H92]. *Leader* is the leader election protocol with 5 processes, as shown in the Appendix. All measurements were made on a Sparc-10 workstation with 128Mbyte of RAM. The runtimes are the sum of system-time and user-time.

The static reduction method gave the shortest run-time in all cases tested, despite the fact that it sometimes searched a larger statespace compared to the dynamic method. In one case (Pftp) the static reduction method even completes its search in a smaller search space than the dynamic method, but presumably, this

is a deficiency in the dynamic method that could be remedied. The amount of memory used by the static and the dynamic reduction method is comparable, and not decisively in favor of either method. For both the dynamic and the static reduction method, the amount of memory used is significantly lower than in a non-reduced search.

In the worst case, the performance of the reduced search is not significantly different from that of a non-reduced search, which is not true for the dynamic method. The best-case application, as expected, shows a (literally) exponential reduction of both runtime and memory requirements.

Exploiting Structure

The reduction rules for the static reduction algorithm also have an unexpected, and sometimes quite dramatic, positive effect on our capability to prove essential properties of large protocol specifications. Because it can lower the complexity of the search if we can mark queues as having exclusive send-access or exclusive receive-access, the user can consciously avoid using queues that do not have this property. It is often possible to rewrite a specification in this way, without changing its functionality. As an example, consider the system from Figure 2a, which corresponds to a validation model that we built in 1988 for the validation of the IEEE 802.2 Logical Link Control Protocol [IEEE84].

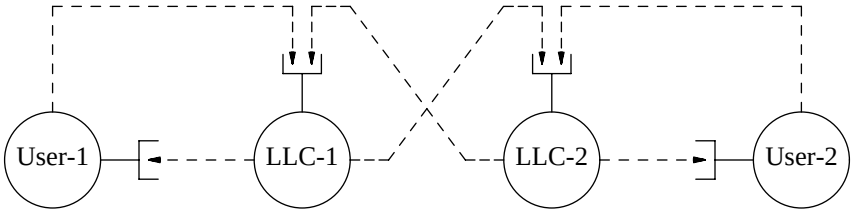

Figure 2a – Model of IEEE LLC 802.2

An exhaustive search for this protocol generates 1,851,049 reachable states. The reduced search brings this down 111,159 reachable states. Notice, however, that the input queues of the LLC processes are non-exclusive. By splitting these two queues, as illustrated in Figure 2b, using a separate queue for each source of information, all queues in the revised model obtain both exclusive receive- and exclusive send-access.

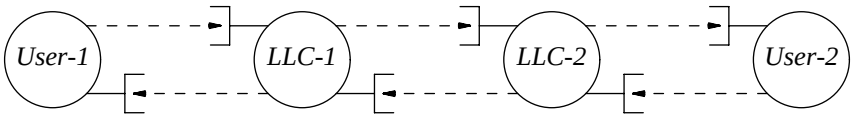

Figure 2b – Revised Model

The reduction algorithm will be able to do a better job in this case, but the revision also has an effect on the complexity of the *non-reduced* search, by avoiding the generation of all non-deterministic interleavings of messages from different sources in the shared input queues of the LLC processes. The search space reduces from 1,851,049 reachable states to 19,407 reachable states, for the non-reduced algorithm. The reduction algorithm reduces this further to 2,000 reachable states. The reduction method in this case successfully guides us to a solution that requires three orders of magnitude less resources than before.

7. CONCLUSION

We have described a new static reduction algorithm that preserves the capability of a depth-first search to prove both safety and liveness properties of concurrent systems. We have shown that this static reduction method performs considerably better than a non-reduced search. It also approaches the reduction achieved by a dynamic method closely, while providing a greater correctness proving power. The static reduction method has no significant worst-case behavior.

Because it is known which types of statements can be exploited by the algorithm, with a static reduction method the designer of a protocol can reduce the complexity of a validation *a priori*, be choosing a specific structure for the model. In certain cases, it may thus become possible to perform true ‘structured design validation’ for concurrent systems.

8. REFERENCES

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, ‘‘Memory efficient algorithms for the verification of temporal properties,’’ Formal Methods in Systems Design I, 1992, pp. 275-288.

[DKR82] D. Dolev, M. Klawe, and M. Rodeh, ‘‘An O(n log n) unidirectional distributed algorithm for extrema finding in a circle,’’ Journal of Algorithms, Vol 3. (1982), pp. 245-260.

[GW91], P. Godefroid, P. Wolper, ‘‘A partial approach to model checking,’’ 6th LICS, 1991, Amsterdam, pp. 406-415.

[H92] G.J. Holzmann, ‘‘Design and Validation of Computer Protocols,’’ Prentice Hall, 1992.

[HGP92] G.J. Holzmann, P. Godefroid, and D. Pirottin, ‘‘Coverage preserving reduction strategies for reachability analysis,’’ Proc. IFIP, Symp. on Protocols Specification, Testing, and Verification, June 1992, Orlando, Fl. pp. 349-364.

[IEEE84] ANSI/IEEE Standard 8802/2, Logical Link Control, ISBN 0-471-82748-7, New York, 1984.

[L83] L. Lamport, ‘‘What good is temporal logic?,’’ Information Processing 83: Proc. of the 9th IFIP World Computer Congress. Ed. R.E.A. Mason, Elsevier Publ., pp. 657-668.

[P93] D. Peled, ‘‘All from one, one for all – on model checking using representatives,’’ 5th Int. Conf. on Computer Aided Verification, Greece, 1993, LNCS 697, Springer Verlag, pp. 409-423.

[P94] D. Peled, ‘‘Combining Partial Order Reductions with On-the-fly Model Checking,’’ 6th Int. Conf. on Computer Aided Verification, Stanford, Ca., June 1994.

[V90] A. Valmari, ‘‘A stubborn attack on state explosion,’’ 2nd Int. Conf. on Computer Aided Verification, Rutgers University 1990, Dimacs Series, Vol 3, pp. 25-42.

[V93] A. Valmari, ‘‘On-the-fly verification of stubborn sets,’’ 5th Int. Conf. on Computer Aided Verification, 1993, LNCS 697, Springer Verlag, pp. 397-408.

[W83] P. Wolper, M.Y. Vardi, and A.P. Sistla, ‘‘Reasoning about infinite computation paths,’’ Proceedings of 24-th IEEE symposium on the foundations of computer science, Tuscan, 1983, pp. 185-194.

APPENDIX

Complete PROMELA Specification of a leader election protocol in a unidirectional ring from [DKR82].

```
#define I 3    /* The node to have smallest identifier */
#define N 5    /* number of processes */
#define L 10   /* size of message buffer */

mtype    = { first, second };

chan q[N] = [L] of {byte,byte};      /* global FIFO buffer */

proctype node(chan in, out; byte id)
{    byte number, maxi=id, neighbourR;    /* local */
     bit active=1;              /* local */

     xr in;   /* assert exclusive receive access to chan in */
     xs out;  /* assert exclusive send access to chan out   */

     out!first(id);
end: do            /* repetition : valid end-state */
     :: in?first(number) ->
          if       /* selection */
          :: active ->
               if   /* selection */
               :: number != maxi ->
                    out!second(number);
                              neighbourR = number
               :: number == maxi ->
                    assert maxi == N
               fi
          :: !active -> out!first(number)
          fi
     :: in?second(number) ->
          if       /* selection */
          :: active ->
               if   /* selection */
               ::   neighbourR > number && neighbourR > maxi ->
                    maxi = neighbourR;
                    out!first(neighbourR)
               :: !(neighbourR > number && neighbourR > maxi) ->
                    active = 0
               fi
          :: !active -> out!second(number)
          fi
     od
}

init {
     byte n=1;     /* a local variable */
     atomic {      /* non-interleaved */
          do       /* repetition */
          :: n <= N ->
               run node(q[n-1], q[n%N], (N+I-n)%N+1);
               n = n+1
          :: n > N ->
               break    /* end repetition */
          od
}    }
```