Symbolic Model Checking of Software

Flavio Lerda, Nishant Sinha, Michael Theobald
School of Computer Science, Pittsburgh, PA 15213
{flerda,nishants,theobald}@cs.cmu.edu

Abstract

In hardware verification, the introduction of *symbolic* model checking has been considered a breakthrough, allowing to verify systems clearly out-of-reach of any explicit-state model checker. In contrast, in the verification of concurrent software, model checking is still predominantly performed by explicit-state model checkers such as SPIN. These methods gain much of their efficiency from state-reduction techniques such as partial-order reduction. To achieve a similar breakthrough for software as has been witnessed in moving to symbolic methods in hardware, efficient state-reduction techniques for symbolic methods must be developed. This paper introduces symbolic two-phase, a very promising technique for symbolic model checking of concurrent software. Experimental results show how this technique does not only improve the existing symbolic model checking approach, but it can be used to tackle problems that are not tractable using explicit-state methods.

1 Introduction

There has been a number of major recent initiatives toward making software verification more efficient [BR01] [VHB+03] [CGP02] [Hol02] [MPC+02] [HJMS02]. An important reason for this trend is that bugs in software systems can have dramatic consequences in safety critical applications [BOB92] [SMB+99] [Lio96]. Moreover, a recent study [New02] pointed out the major negative economic impact of buggy software.

Designing bug-free software is a very challenging problem. As the typical software design process uses tools that one may not be able to trust and errors also tend to be introduced by designers and programmer, software verification tools are indispensable. More and more software systems consist of concurrent-running components, and thus the verification of concurrent software is an increasingly important problem.

The current practice in the software design industry is to use testing to validate software. Testing has been used over the years both in hardware and software to validate designs. However testing is neither exhaustive nor very effective for software, especially concurrent software, which is much more complex than sequential software and where timing can play a very important role.

A very mature state-of-the-art tool for the verification of concurrent software is the SPIN model checker [Hol97]. Interestingly, SPIN is an *explicit-state* model checker. In hardware verification, the introduction of *symbolic* model checking is generally considered a breakthrough. Symbolic model checkers apply data structures, such as BDDs [Bry86] , that manipulate large number of objects simultaneously. In particular, symbolic model checkers allow to verify systems clearly out-of-reach of any explicit-state model checker [BCM+90] [EOH+93], with state spaces that go beyond 10^{20} states.

An important reason why software model checking is still predominantly performed using explicit-state model checkers such as SPIN, is that these methods gain much of their efficiency from state-reduction techniques such as partial-order reduction.

Partial-order reduction takes advantage of the independence of transitions executed by different processes. In an asynchronous model of composition all different interleavings of the transitions of the processes must be checked as different interleaving can lead to different states, and properties that are true for some interleavings may be false for a different interleaving of the same processes. However, under certain conditions (see Section 2.2), it is possible to visit only a representative set of interleavings with the guarantee that if a property is violated by the system, a violation will be present in the representative set of interleavings as well. The state space that needs to be visited can often be reduced dramatically.

Therefore, partial-order reduction is useful only when the system has an asynchronous model of composition. Most hardware designs are based on a clocked-approach and thus synchronous. Concurrent software, on the other hand, is asynchronous as the different components might be running on different processors or be interleaved by the scheduler.

Given the asynchronous nature of concurrent software and the dramatic reduction that can be achieved with partial-order reduction, to make a breakthrough for software similar to the one due to symbolic methods in hardware, efficient state-reduction techniques for symbolic methods must be developed.

Two approaches that apply partial-order methods in a symbolic setting are the approaches by Alur et al. [ABH+97] and Kurshan et al. [KLM+97]. However, both methods suffer from an inefficient in-stack check to guarantee that no transition gets indefinitely postponed: this check limits the effectiveness of the reduction.

This paper presents a symbolic partial-order reduction technique based on a different approach. This technique does not have the drawback of an inefficient reduction and thus is very promising for the verification of software.

The remainder of the paper is structured as follows. In Section 2 we present some background information that is used throughout the paper. Next we will discuss the proposed algorithm, and refer to the two-phase partial-order reduction this algorithm is inspired by. In Section 4 we will present related work together with the results obtained from a preliminary comparison with existing tools. Section 5 gives conclusions as well as directions for future work.

2 Background

2.1 Definitions

We assume a process oriented modeling language with each process maintaining a set of local variables that only it can access. The values of the local variables of a process form the *state of the process*. Each process' local variables include a distinguished local variable called *program counter*.

A concurrent system ("system") consists of set of processes with local variables, a set of global variables which all processes can access, and a set of point-to-point channels of finite capacity used for communication. The *state of the system* consists of the states of all the processes, the values of all global variables, and the content of the channels in the system. The potential state space S is simply the cross product over the finite ranges thereof.

Each process is specified in terms of statements. For each value of the program counter one of a finite number of statements may execute. We allow the specification of multiple statements per program counter value to be able to describe non-determinism such as reading from several concurrent channels. Each statement also defines an enabling condition, which specifies for which states of the system the statement can be executed.

Each such statement is formalized as a transition function on the state space. The notion of a statement being enabled is captured by defining transitions over a subset of state space $enabled(t) \subseteq S$ for a transition t. We say that a transition t is enabled in state $s \in S$, $enabled_t(s)$, iff $s \in enabled(t)$. Let t be a transition enabled in $s \in S$. The state that is reached from s by executing t is denoted by $t(s)$.

Central to partial-order reduction are the concepts of *enabledness* and *commutativity*.

Given two enabled transitions t_1 and t_2 in a state s:

Enabledness Condition The execution of t_2 does not disable t_1, i.e. $enabled_{t_1}(t_2(s))$.

Commutativity Condition The execution of t_1 followed by t_2 leads to the same state as executing t_2 followed by t_1, i.e. $t_2(t_1(s)) = t_1(t_2(s))$.

2.2 Partial-Order Reduction

There are many different equivalent approaches on which partial-order reduction algorithms can be based, including stubborn sets [Val90], ample sets [Pel94, CGP99], and sleep sets [GW91].

The common idea among the different approaches is to define an equivalence relation over all the possible execution paths. At least one path from each equivalence class must be visited in order to verify the

correctness of the original system. All approaches thus define two types of "steps" when the system is in a state s:

- A **full expansion** corresponds to executing all enabled transitions of all processes at state s. This is the normal approach taken by model checkers without partial-order reduction.
- Under certain conditions, it is possible to expand only a subset of the enabled transitions, called a **partial expansion**. The conditions must guarantee to not eventually exclude any equivalence class.

A partial expansion essentially postpones the transitions which are not included. A similarity of all these approaches is the need to avoid postponing a transition *indefinitely* in some execution path. This can easily occur if there exists a loop in the global reachable state space which contains only partially expanded states. If this is the case all the transitions that are not taken in any of the partial expansions in the cycle will be postponed indefinitely.

A common way to avoid this problem is to add a so called *in-stack proviso*, which is checked at run-time. When the *in-stack proviso* is not satisfied a full expansion of the current state is necessary. The *in-stack proviso* makes sure that any loop will contain at least one full expansion, by forcing a full expansion whenever a cycle is detect.

Explicit-state model checkers perform a depth first search visit, therefore a cycle can be detect by checking if a newly visited state belongs to the current DFS stack, hence the name *in-stack proviso*.

A very efficient way to deal with checking the *in-stack proviso* (see Section 3.1) is based on the notion of a state being *deterministic*.

A state s is deterministic for a process P, iff

- only one of the process' transitions is enabled in s;
- the enabled transition commutes with transitions that can be executed at any given point forward by any other processes;
- executing the enabled transition does not disable transitions that can be executed at any given point forward by any other processes;
- executing a transition at any given time forward in another process cannot disable or enable any of P's. transitions.

Note, however, that a deterministic process may enable transitions in other processes.

2.3 Practical Partial-Order Reduction

Computing the set of conditions necessary and sufficient for partial-order reduction is typically too computationally expensive [CGP99]. Therefore, in practice partial-order reduction algorithms compute safe approximation, i.e. some steps taken may use sub-optimal (i.e. larger than necessary) partial expansions. Hence, the reduced state-space is guaranteed to conserve the desired properties, but not necessarily the minimum such state-space. In particular, a very simple heuristic is to use syntactic information about the transitions, e.g. which variables processes and transitions read or writes, and which channels are accessed.

We can safely say that a state s is deterministic for process P_i, if only one transition of P_i is enabled in s and the following conditions hold for every transition t, enabled or disabled, of process P_i:

- t does not access any global variable;
- One of the following holds:
  - t is not a channel operation;
  - t is a receive operation from a channel, and the channel is not empty in s;
  - t is a send operation on a channel, and the channel is not full in s.

3 Symbolic Two-Phase

This section introduces a novel algorithm for symbolic partial-order reduction called *symbolic two-phase*. We first present some detail on the two-phase partial-order reduction for explicit-state model checking [NG02] – we show how this approach offers great advantages in the context of symbolic methods. We extend this approach to exploit the power of symbolic exploration in several ways, including the introduction of an added fix-point computation step, and the removal of the requirement to restrict the applicability to deterministic transitions.

This paper focuses on safety properties, and reachability will be the only property we are concerned with. Extensions to liveness properties and temporal logics are possible but are left for future work.

3.1 Two-Phase Partial-Order Reduction

Our approach is inspired by the two-phase explicit-state partial-order reduction algorithm presented in [NG02].

This algorithm works in two separate phases:

- Phase one expands only deterministic states.
- Phase two performs a full expansion of the given state, executing every transition enabled at that state.

Phase one considers each process in a fixed order: as long as such process is deterministic, the only enabled transition for that process is executed; otherwise the phase one algorithm moves on to the next process. After expanding all processes the last reached state is passed to phase two. Phase two then performs a full expansion of such state and calls phase one on each of the newly reached states.

In order to avoid ignoring a transition indefinitely, it is necessary to perform a full expansion for at least one state of every cycle. To ensure that, every time a cycle is detected during phase one, control is passed to the next process, or to phase two if expanding the last process. Cycles that go beyond a single 'phase one' instance, must contain a fully expanded state and therefore do not need to be taken care of.

3.2 Symbolic Algorithm

While checking the in-stack proviso is quite simple for explicit-state model checkers, it is not as easy to do so in symbolic model checkers. Symbolic model checkers usually perform a breadth first search, therefore they don't maintain a stack and checking for cycles is not directly possible.

The great potential the two-phase algorithm offers is that the in-stack proviso needs to be checked only for states belonging to phase one, and only against the stack internal to the current phase one.

Other approaches to symbolic partial-order reduction (see Section 4.2) in most cases can only perform a limited reduction because of the way the in-stack proviso is implemented symbolically: no depth first stack is available and therefore a coarse conservative approximation is computed, limiting the effectiveness of the reduction.

On the other hand the approach taken by two-phase reduces the number of cases in which a symbolic algorithm will mistakenly assume a state belongs to the stack. During a breadth first visit the stack is not available, so any state that has already been visited is assumed to be on the stack. With two-phase the state needs to be check only against the stack local to phase one, so during a breadth first visit it can be approximated using only the state visited during since the last phase one start, instead of all the visited states.

3.3 Improvements

3.3.1 Defining the Transition Relations

We define two transition relations: one contains all the transition from states that are deterministic (as defined above) and another one corresponding to the whole system.

The first transition relation is partitioned over the process that is executing, to form a set of transition relations that will be used during phase one. Phase two will correspond to one step of image computation using the transition relation of the whole system.

3.3.2 Dropping the Determinism

The condition required by the two-phase algorithm that the transition expanded during phase one is the only one enabled for such process is not necessary for the correctness of the algorithm but was introduced to avoid keeping a full fledged stack and avoid backtracking during phase one. Since our algorithm does not actually maintain a stack, we can relax such condition and formulate a new version of the algorithm. This also simplifies the condition that need to be posed on transitions to decide if they belong to phase one or not.

3.3.3 Fix-Point Computation During Phase One

During phase one each process is executed in a fixed order and after reaching a state during the expansion of the last process that is not deterministic, such state is passed directly to phase two.

However, the execution of another process might have caused a previous process that was not in a deterministic state to reach a deterministic one.

This can happen for instance when a process P_i is not in a deterministic state as the only available transition is an exclusive read from an empty channel. However, a successive process might execute an exclusive write to that same channel, making the channel not empty, and enabling the transition and making process P_i deterministic.

One improvement of the two-phase algorithm is therefore to re-execute each process until none of the processes was able to perform an operation. Furthermore, this guarantees that none of the transitions that are present in the transition relations for phase one will be enabled in phase two, making it possible to build a simpler transition relation which contains all but the transitions included in the phase one transition relations.

3.3.4 Main Algorithm

The algorithm is illustrated by the pseudocode in figure 1.

The algorithm will have two phases that are interleaved.

- Phase one will apply the phase one transition relations, one at a time until we reach a fix point.

  An improvement on the two-phase algorithm is to restart from the first process after reaching a fix point on the last one, until a fix point for every transition relation is reached.

  Another difference is that instead of applying this approach to only one state at a time, the transition relation is applied to a set of states at once: this has the advantage of being able to explore multiple paths at once, however the breadth first nature of such a visit makes it impossible to accurately detect cycles.

  If we reach a state that we already visited since the current phase one started, we consider it to be either due to stuttering or the beginning of a cycle and therefore we added it to the set of states that will become the initial frontier for the next process.

  The last frontier generated by phase one is passed to phase two.

- Phase two applies the phase two transition relation to the frontier received from phase one and the generated frontier, from which the already visited states are removed, is passed to the next instance of phase one.

The algorithm terminates when the frontier generated by either phase is empty.

3.4 Advantages

One of the main advantage of using a symbolic approach is the ability to visit sets of reachable states that go beyond the possibility of any explicit-state algorithm [BCM+90]. Moreover the application of a transition relation that is encoded as a BDD provides the possibility to visit many different transitions in one step: during phase one this can only occur if a process has more than one transition enabled (the process is non-deterministic). This can occur in the modeling of software either because of abstractions or because of the

```
PROCEDURE phase1()
BEGIN
  LOCAL Stack := Empty;
  LOCAL Moved := True;
  WHILE Moved
  BEGIN
    Moved := False;
    FOR EVERY Process I
    BEGIN
      LOCAL LoopFrontier := Empty;
      WHILE Frontier IS NOT EMPTY
      BEGIN
        LOCAL Image := apply(
            TransitionRelation[I],
            Frontier);
        Image := Image - Visited;
        LoopFrontier := LoopFrontier
            UNION
            (Image INTERSECTION Stack);
        Frontier := Image - Stack;
        Stack := Stack UNION Frontier;
        IF Frontier IS NOT EMPTY
        BEGIN
          Moved := True;
        END;
      END;
      Frontier := LoopFrontier;
    END;
  END;
  Visited := Visited UNION Stack;
END
```

```
PROCEDURE phase2()
BEGIN
  LOCAL Image := apply(
      TransitonRelationPhaseTwo,
      Frontier);
  Frontier := Image - Visited;
  Visited := Visited UNION Frontier;
end

PROCEDURE mc()
BEGIN
  Visited := Init;
  Frontier := Init;
  WHILE Frontier IS NOT Empty
  BEGIN
    phase1();
    phase2();
  END;
END
```

Figure 1: Symbolic Two-Phase Algorithm Pseudocode

modeling of the environment; multiple enabled transitions are more common during phase two where the transition relation represents the whole system, and different transition may be enabled in different processes.

4 Experimental Results

We have implemented the symbolic two-phase algorithm as part of the NuSMV symbolic model checker. NuSMV [CCG+02] is the evolution of SMV, the first symbolic model checker developed by Ken McMillan at Carnegie Mellon University.

To verify the effectiveness of our approach we decided to run a series of tests to compare our tool with these two: first of all, we compare it with NuSMV without partial-order reduction, to verify how much of an improvement our tool gives; since our algorithm was inspired by the two-phase algorithm, we compare it to the Utah Verifier (UV from now on) [NG02], which implements the two-phase algorithm; last we compare it to SPIN [Hol97], the leading tool for verification of concurrent systems.

To make our comparison fair, we decided to use the same models for each of the tools. Since our tool is able to accept a broader range of input programs, we translated the models (which were originally written in Promela, the input language of both SPIN and UV) into a formalism that can be accepted by our tool. This conversion has been done automatically using a translator currently under development within our group [Ler03]: this approach avoids giving any advantage to one of the tools, except perhaps SPIN and UV for which Promela is the natural input notation.

Non-PO				PO		
states	**time**	**mem.**		**states**	**time**	**mem.**
4864210	3217.69	63.6	**Migratory Protocol (2)**	155040	108.63	56.3
1270	0.87	6.2	**Stable Marriages (2)**	710	0.84	7.3
3107	4.26	10.3	**Stable Marriages (3)**	1275	2.72	10.4
71495	112.25	24.7	**Stable Marriages (5)**	10351	31.56	30
2187	0.08	0.7	**Best (7)**	15	0.06	0.7
3486780000	0.56	5.7	**Best (20)**	41	0.34	5.7
27	0.04	0.3	**Worst (3)**	15	0.04	0.3
3486780000	0.46	5	**Worst (20)**	2097150	0.36	5
N/A[1]			**Worst (100)**	2.5353E+30	14.34	14.6

Table 1: Comparing NuSMV with and without partial order reduction

The results have been obtained using similar options for the tools where available to compare only the main algorithms. The tools have been run on a dual-processor 1.4 AMD Athlon giving each tool up to 1GB of memory.

Even if we tried to make all tools equal some differences could not be leveled: in the memory requirements it has to be noted that for NuSMV the transition relation is stored in dynamic memory (and therefore counted as part of the memory requirements) while for both explicit state model checkers, the transition relation is encoded as part of the generated verifier program and therefore it is not accounted for in the memory requirements; in the running time for NuSMV the time necessary to parse the input and generate the transition relation is included in the presented results, while this operation is performed by the verification program generator and the compiler and therefore not taken into account in the case of both explicit-state model checkers.

Moreover the version of the tool used for computing the result is still a prototype and many improvements still need to be implemented which would reduce the memory requirements and improve the running time.

4.1 Improvement over NuSMV

The first set of results (Table 1) is meant to compare NuSMV with our version of the tool which uses the symbolic two-phase partial-order reduction algorithm.

The examples presented in this table mostly belong to the official distributions of UV, or have been taken from published work.

The "Migratory Protocol" example, from the distribution of UV, describes the migratory cache coherency protocol of the Avalanche system.

The "Stable Marriages" example implements a distributed algorithm for the solution of the stable marriages problem . This example is parameterized on the number of couples.

The "Best" example is an example presented in [NG02] which represents a case were the UV tool gives a reduction while SPIN is not able to perform any reduction at all. This example is parameterized on the number of processes that compose the system.

The "Worst" example is an example presented in [NG02] which represents a case where the UV tool gives no reduction while SPIN is able to perform some reduction. This example is parameterized on the number of processes that compose the system.

These examples show that in many cases the reduction of the state space causes a reduction in the memory usage or a reduction in the run-time. However if the state-reduction is too small, we might have an increase of both. The increase of memory requirements can be easily explained by the fact that a BDD representing a smaller set of states is not necessarily smaller than one representing a bigger set. The increase in run-time can be explained by the fact that the running time of most BDD operation is proportional to the size of the BDDs involved in the operation and therefore bigger BDDs lead to a longer running time.

[1] Results are not available because the model checker ran out of memory. The memory limit was set of 1GB.

	NuSMV			PV			SPIN		
	states	**time**	**mem.**	**states**	**time**	**mem.**	**states**	**time**	**mem.**
Migratory Protocol (2)	155040	108.63	56.3	86246	1	4.3	435456	2.34	42.8
Stable Marriages (2)	710	0.84	7.3	595	0.01	2.2	568	0.01	1.5
Stable Marriages (3)	1275	2.72	10.4	1135	0.01	2.2	945	0.01	1.5
Stable Marriages (5)	10351	31.56	30.0	9063	0.14	2.6	8421	0.03	2.1
Best (7)	15	0.06	0.7	15	0.01	2.2	2187	0.03	1.5
Best (20)	41	0.34	5.7	41	0.01	2.2	**N/A**[1]		
Worst (3)	15	0.04	0.3	27	0.01	2.1	15	0.01	1.5
Worst (20)	2097150	0.36	5.0	**N/A**[1]			2097150	15.03	110.6
Worst (100)	2.53e+30	14.34	14.6	**N/A**[1]			**N/A**[1]		

Table 2: Comparing NuSMV with SPIN and PV

When the reduction in the number of states is more than marginal, the reduction in the memory requirements and running times is also present, even making, for the last entry in the table, an example too big to be verified without partial-order reduction, verifiable.

4.2 Other Symbolic Approaches to Partial-Order Reduction

The effort to combine partial-order reduction and symbolic model checking is not new. Other approaches by Alur et al. [ABH+97] and Kurshan et al. [KLM+97] have been presented before and we would like to give a quick introduction to such approaches here and compare the different algorithm. We would have liked to have an experimental comparison with these approaches as well, but neither of the tools is freely available.

Other approaches to combining symbolic model checking and partial-order reduction failed to obtain a reduction comparable with explicit methods in order to guarantee the absence of transitions indefinitely postponed. Alur et al. [ABH+97]'s approach is to dynamically check for possible cycles: however, given that the breath first does not maintain the full stack, this might not perform some reductions which would be valid, because it is not able to determine the difference between a real cycle and visiting an earlier visited state. Kurshan et al. [KLM+97] instead avoids ignoring a transition indefinitely by fully expanding at least one state from each local cycle of safe transitions, where a local cycle is a cycle in the control flow graph of a local process: however not every local cycle of safe transitions correspond to a global cycle.

4.3 Comparison with Explicit-State Model Checkers

The next table (Table 2) compares NuSMV with partial-order reduction with explicit-state tools using the same examples presented before.

First of all it has to be noted that whenever the same example can be handled by both the symbolic and explicit-state tools, the explicit state tools are almost always using less memory and less time. This is in part due to the fact that both SPIN and UV generate a specific verifier for each model that needs to be verified, generating this way the most optimized verifier.

The only cases where NuSMV with partial-order reduction is able to outperform the explicit-state model checkers is when they are not able to represent explicitly the great number of states even after reduction, while the symbolic representation used by NuSMV can more easily handle the state space.

Another interesting point is that the reduced state space is in almost all cases as small as the number of states returned by the explicit-state model checkers: that is the approximation of the proviso condition is not really affecting the reduction.

It is also relevant to notice that in both the Best and Worst examples, symbolic two-phase performs as the best of the two algorithms (SPIN and UV): we believe this is due to the fact that we lifted the constraint about deterministic transitions from the two-phase algorithm allowing for ample sets of size bigger than one, but at the same time conserving the two phase approach.

	NuSMV					
	Non-PO			PO		
#	states	time	mem.	states	time	mem.
2	70	0.11	1.1	48	0.10	1.0
3	488	0.57	4.6	209	0.31	3.0
4	3576	6.77	10.6	922	1.77	10.4
8	N/A[1]			306903	3553.86	381.8

	NuSMV					
	Non-PO			PO		
#	states	time	mem.	states	time	mem.
2	187	0.17	3.0	119	0.17	3.3
3	5602	5.61	12.5	2566	2.14	11.7
4	473173	650.25	62.9	135173	133.69	37.6
5	N/A[1]			7699370	11635.00	829.2

	PV					
	Non-PO			PO		
#	states	time	mem.	states	time	mem.
2	70	0.01	2.1	48	0.04	2.1
3	488	0.03	2.2	209	0.01	2.2
4	3576	0.38	2.5	922	0.04	2.2
8	N/A[1]			306903	28.62	60.4

	PV					
	Non-PO			PO		
#	states	time	mem.	states	time	mem.
2	187	0.01	2.1	139	0.01	2.1
3	5602	0.32	2.6	3298	0.12	2.4
4	473173	46.62	49.1	167173	6.99	18.9
5	N/A[1]			N/A[1]		

	SPIN					
	Non-PO			PO		
#	states	time	mem.	states	time	mem.
2	70	0.01	1.5	48	0.02	1.5
3	488	0.01	1.5	209	0.01	1.5
4	3576	0.1	2.3	922	0.01	1.7
8	N/A[1]			306903	11.82	232.8

Table 3. The Leader Election Protocol with NuSMV, PV, and SPIN

	SPIN					
	Non-PO			PO		
#	states	time	mem.	states	time	mem.
2	187	0.01	1.5	119	0.01	1.5
3	5602	0.07	2.4	2566	0.07	1.9
4	473173	13.58	119.7	135173	1.81	34.3
5	N/A[1]			N/A[1]		

Table 4. The Leader Election Protocol with Non-Deterministic Initial State

4.4 The Leader Election Protocol

The next table (Table 3) shows the detail of the comparison of the three tools, with and without partial-order reduction, on a set of instances of the leader election protocol.

The "Leader Election" protocol has been used in many papers about partial-order reduction [ABH+97] [KLM+97] [HP94], both in the explicit-state and in the symbolic model checking domain, to compare results. These examples models the leader election protocol on a unidirectional ring used to determine which node has the highest identifier using a distributed algorithm. The example is parameterized on the number of nodes that belong to the ring.

As it can be noted, the example has been encoded in exactly the same way in all the tools, and the results obtained (in terms of number of states) are the same for all tool, with and without partial-order reduction respectively.

In all the shown examples the size of the state space after reduction is small enough for the explicit-state model checkers to outperform the symbolic model checker.

The last table (Table 4) shows the Leader Election example with a modification. As noted in [ABH+97], the example presented before and used with many explicit-state model checkers, only represent a particular case for a fixed assignment of identifiers to nodes. This obviously is not representative of all the different cases that the algorithm was meant to deal with. We need to consider all the possible assignment of identifiers, in particular it suffice to consider identifiers in the range from 0 to $n-1$, where n is the number of nodes.

As can be seem from the example this causes an exponential blow up of the number of states, that partial-order reduction can only partially mitigate. With only 5 nodes neither of the explicit-state model checker is able to verify the system, and a more efficient representation like the one used by a symbolic model checker becomes necessary.

In specific all the different assignments of identifiers to the nodes generate a set of initial states: the explicit-state model checkers need to visit all states reachable from these initial states individually. One advantage of a symbolic approach (besides a more compact representation) is the ability to execute at the same time all transitions enabled at the initial states. This is therefore a case in which the symbolic model checker can be much more effective.

5 Conclusions and Future Work

In this paper we propose a new symbolic method that exploits partial-order reduction to extend the applicability of symbolic model checking to concurrent and distributed software.

The effectiveness of this method is based on a more efficient way of checking the *in-stack proviso*. This approach was inspired by the two-phase partial-order reduction algorithm used by the explicit-state model checker UV, but this algorithm has been revisited to better exploit the potential of symbolic exploration. We extended the approach to exploit the power of symbolic exploration in several ways, including the introduction of an added fix-point computation step, and the removal of the requirement to restrict the applicability to deterministic transitions.

We implemented this algorithm within the framework for verification offered by NuSMV and we presented some preliminary comparison with two leading tools in the explicit-domain. We have been able to show that particularly big examples can be verified by this new tool, but not by either an explicit-state model checker, or a symbolic model checker without partial order reduction.

As for the future, we are planning on extending our algorithm to liveness properties as well as going to work on improving the core algorithm to enhance its performance.

We believe that the presented algorithm can be used as a core to build an efficient tool for model checking of concurrent software. Additional challenges that lie ahead are to make some of the NuSMV algorithms for symbolic computation more efficient for the case of software as they are currently optimized for hardware.

References

[ABH+97] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351, 1997.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[BOB92] Michael Blair, Sally Obenski, and Paula Bridickas. Patriot missile software problem. Technical Report GAO/IMTEC-92-26, General Accounting Office, 1992.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057, 2001.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CCG+02] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In E. Brinksma and K. Guldstrang Larsen, editors, *Computer Aided Verification - Proc. of the 14th International Conference*, volume 2404 of *LNCS*. Springer, Copenhagen, Denmark, July 2002.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CGP02] Satish Chandra, Patrice Godefroid, and Chris Palm. Software model checking in practice: An industrial case study. In *Proc. of 24th International Conference on Software Engineering*, pages 431–441, 2002.

[EOH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *The Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, pages 5–20, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[GW91] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. In *Logic in Computer Science*, pages 406–415, 1991.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[Hol02] Gerard J. Holzmann. Software analysis and model checking. In *Proc. of 14th International Conference on Computer Aided Verification*, 2002.

[HP94] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. FORTE 1994 Conference*, 1994.

[KLM+97] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenign. Verifying hardware in its software context. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 742–749. IEEE Computer Society, 1997.

[Ler03] Flavio Lerda. Translating Promela into SMV for partial-order reduction. Technical Report to be published, Carnegie Mellon University, 2003.

[Lio96] J. L. Lions. Arianne 5: Flight 501 failure. Technical report, European Space Agency, 1996.

[MPC+02] Madanlal Musuvathi, David Y.W. Park, Andy Chow, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *Proc. of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[New02] Michael Newman. Software errors cost U.S. economy $59.5 billion annualy. Technical Report NIST 2002-10, National Institute of Standards and Technology, 2002.

[NG02] Ratan Nalumasu and Ganesh Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, 20(3):231–247, May 2002.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of CAV'94*, pages 377–390. Springer Verlag, LNCS 818, 1994.

[SMB+99] A. Stephensons, D. Mulville, F. Bauer, G. Dukeman, P. Norvig, L. LaPiana, P. Rutledge, D. Folta, and R. Sackheim. Mars climate orbiter mishap investigation board phase I report. Technical report, National Aeronautics and Space Administration, 1999.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Proceedings of Computer Aided Verification*, pages 25–42, 1990.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.