Electronic Notes in Theoretical Computer Science 89 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 19 pages

Symbolic Model Checking of Software ⋆

Flavio Lerda [1], Nishant Sinha [2], Michael Theobald [3]

School of Computer Science
Carnegie Mellon University
Pittsburgh, USA

Abstract

Model checking is a popular formal verification technique for both software and hardware. The verification of concurrent software predominantly employs explicit-state model checkers, such as Spin, that use *partial-order reduction* as a main technique to deal with large state spaces efficiently. In the hardware domain, the introduction of *symbolic model checking* has been considered a breakthrough, allowing the verification of systems clearly out-of-reach of any explicit-state model checker.

This paper introduces ImProviso, a new algorithm for model checking of software that efficiently combines the advantages of partial-order reduction with symbolic exploration. ImProviso uses implicit BDD representations for both the state space and the transition relation together with a new implicit in-stack proviso for efficient partial-order reduction. The new approach is inspired by the Twophase partial-order reduction algorithm for explicit-state model checking.

Initial experimental results show that the proposed algorithm improves the existing symbolic model checking approach and can be used to tackle problems that are not tractable using explicit-state methods.

⋆ This research was supported by the National Science Foundation (NSF) under grants no. CCR-0121547 and CCR-0098072, by the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, by the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARO, ONR, NRL, NSF, SRC, the U.S. Government or any other entity.

[1] Email: flerda@cs.cmu.edu
[2] Email: nishants@cs.cmu.edu
[3] Email: theobald@cs.cmu.edu

1 Introduction

There have been a number of major recent initiatives toward making software verification more efficient [2,26,6,13,20,11]. An important reason for this trend is that bugs in software systems can have dramatic consequences in safety critical applications [3,24,17]. Moreover, a recent study [22] 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 may themselves contain bugs and therefore cannot be trusted, and since errors also tend to be introduced by designers and programmers, software verification tools are indispensable. More and more software systems include multi-threaded or distributed 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. However, testing is typically not exhaustive, and therefore is not sufficient to guarantee correctness.

Model checking is a popular formal verification technique for both software and hardware. A very mature state-of-the-art tool for the verification of concurrent software is the Spin model checker [12]. Spin is an explicit-state model checker that employs *partial-order reduction* as a main technique to efficiently deal with large state spaces.

Partial-order reduction takes advantage of the independence of the steps executed by concurrent processes. In an asynchronous model of computation, all different interleavings of the concurrent processes must be checked. Different interleavings 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 (cf. 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. Techniques that only visit a reduced set of interleavings are called partial-order reductions. The reduced state space can often be dramatically smaller than the full state space.

In the hardware domain, the introduction of *symbolic model checking* is generally considered a breakthrough. Symbolic model checkers use data structures, such as BDDs [4], that manipulate large number of objects simultaneously. In particular, symbolic model checkers allow the verification of systems clearly out-of-reach of any explicit-state model checker [5,8], with state spaces that go beyond 10^{20} states.

Partial-order reduction is useful only when the system has an asynchronous model of computation. While most hardware designs are based on a clocked approach and thus synchronous, concurrent software is asynchronous in nature. BDD-based symbolic model checking techniques, which provide efficient representation and manipulation of both the state space and the transition relation, are applicable to both hardware and software. Therefore, an efficient combination of *symbolic model checking* and *partial-order reduction* can be used to overcome some of the current limitations of software model checking. Two such approaches are the techniques by Alur et al. [1] and Kurshan et al. [15]. However, both techniques suffer from an inefficient *in-stack proviso* check that limits the effectiveness of the reduction. Partial-order reduction algorithms are based on the idea of *postponing* transitions without affecting the property to be checked, and *provisos* are used to guarantee that no transition is postponed indefinitely.

There are three main challenges for model checking software using a BDD-based symbolic approach. First, an effective way to extract a finite model from a software language is needed. Second, a model checker must effectively combine the symbolic approach with other optimization techniques, and in particular, with partial-order reduction. Third, new BDD algorithms optimized for the software domain must be developed. This paper focuses on the second point, however, the other two points will be briefly discussed in the results and conclusions sections.

This paper introduces ImProviso, a new algorithm for model checking of software that efficiently combines the advantages of partial-order reduction with symbolic exploration. ImProviso uses implicit BDD representations for both the state space and the transition relation together with a new implicit *in-stack proviso* for efficient partial-order reduction. The new approach is inspired by the Twophase partial-order reduction algorithm for explicit-state model checking. Our new technique introduces a much tighter over-approximation of the *in-stack proviso* than presented in previous work, 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 discuss the new proposed ImProviso algorithm is Section 3. In Section 4, we present 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 where each process maintains a set of local variables that cannot be accessed by other processes. The values of the local variables of a process form the *state of the process*. Each process includes a distinguished local variable called *program counter*.

A *system* consists of a set of concurrent processes with local variables, a set of global variables that all processes can access, and a set of point-to-point channels [4] of finite capacity used for communication. The *state of the system*

[4] For simplicity we only consider point-to-point channels, but this is not a necessary restriction of the presented approach.

consists of the states of all the processes, the values of all global variables, and the content of the channels. The potential state space S is simply the cross product over the finite ranges thereof.

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

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

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

Enabledness Condition: The execution of t_2 does not disable t_1, i.e. $t_2(s) \in enabled(t_1)$.

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. $t_2(t_1(s)) = t_1(t_2(s))$.

The concepts of *enabledness* and *commutativity* are central to partial-order reduction.

2.2 Partial-Order Reduction

Partial-order reduction is a technique to reduce the state space that needs to be visited to model check a system. The basic idea 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 system.

There are many different approaches for partial-order reduction, including stubborn sets [25], ample sets [23,9], and sleep sets [10]. All approaches define two types of "steps" at a given state:

- A **full expansion** generates the next states for all enabled transitions. 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 visit a path from every equivalence class.

A partial expansion essentially *postpones* the transitions which are not included. A postponed transition will be enabled in all the next states. In order to ensure correctness a transition must not be *indefinitely postponed* along any execution path. This can occur if there exists a loop in the global reach-

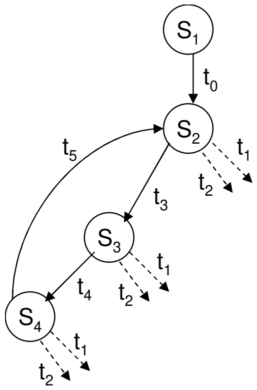

Fig. 1. The dotted transitions are indefinitely postponed.

able state space which contains only partial expansions and some transition is postponed by all of the partial expansions in the cycle (transitions t_1 and t_2 in Figure 1).

In partial-order reduction, therefore, two main issues need to be considered: *(i)* which subsets of transitions to choose for partial expansions, and *(ii)* how to avoid that transitions are postponed indefinitely. The former leads to the notion of *deterministic states* and the latter leads to the notion of *proviso*, discussed below.

Deterministic States

An effective way to choose subsets of transitions for partial expansions is based on the notion of a state being *deterministic*. A state s is deterministic for a process P iff

- only one transition of the process P is enabled in s;
- the enabled transition of P commutes with transitions that can be executed by another process at any point in the future (Commutativity Condition; cf. Section 2.1);
- executing the enabled transition of P does not disable transitions that can be executed by another process at any point in the future (Enabledness Condition; cf. Section 2.1);
- a transition executed by another process at any point in the future cannot disable or enable any of the transitions of P defined at the program counter location of P in s (Enabledness Condition and its dual).

Note, however, that the definition of deterministic does not preclude the enabled transition of P from enabling transitions in another process.

In a deterministic state, the only enabled transition can be taken, i.e. used for a partial expansion, without affecting the safety property to be checked.

Proviso

A common way to avoid that a transition is postponed indefinitely is to check a so called *in-stack proviso* at run-time. Explicit-state model checkers perform a depth-first search, therefore a cycle can be detected by checking if a newly visited state belongs to the current depth-first search stack, hence the name *in-stack proviso*. When the *in-stack proviso* is not satisfied a full expansion of the current state is performed. 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 detected.

2.3 Practical Partial-Order Reduction

Computing the set of conditions necessary and sufficient for partial-order reduction is typically computationally too expensive [9]. Therefore, in practice, partial-order reduction algorithms compute safe approximations, i.e. some steps may be larger than necessary partial expansions. Hence, the reduced state space is guaranteed to preserve the desired properties, but may not necessarily be the minimum such state space.

A simple but effective heuristic is based on syntactic information from the specification of the processes, i.e. which variables and channels are accessed by each statement. A safe approximation for a state s to be deterministic for process P is that only one transition of P is enabled in s and the following conditions hold for every enabled or disabled transition t of process P defined at the program counter of P in s:

- 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.

These conditions imply that state s is deterministic for process P.

3 New Algorithm: ImProviso

This section introduces a novel partial-order reduction algorithm for symbolic model checking called ImProviso. Before explaining the new algorithm, we briefly review the Twophase algorithm for *explicit-state model checking*, developed by Nalumasu and Gopalakrishnan [21], which forms a basis of our approach.

3.1 Twophase *Algorithm*

Our approach is inspired by the Twophase explicit-state partial-order reduction algorithm presented in [21]. The Twophase algorithm consists of two alternating phases:

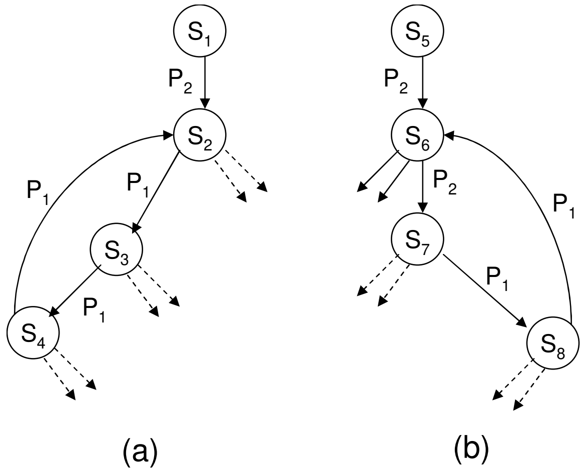

Fig. 2. Back edge: (a) local to phase one; (b) beyond current phase one.

- *Phase 1* expands only deterministic states (cf. Section 2.2), considering each process at a time, in a fixed order. As long as a process is deterministic, the single transition that is enabled for that process is executed. Otherwise, the algorithm moves on to the next process. After expanding all processes, the last reached state is passed on to *phase 2*.
- *Phase 2* performs a full expansion of the given state, executing every transition enabled at that state. Then, *phase 1* is invoked for each of the newly visited states, one at a time.

In order to avoid ignoring a transition indefinitely, it is sufficient to perform a full expansion for at least one state of every cycle. To ensure that, every time a cycle is detected within the current *phase 1* (e.g. states S_2, S_3, and S_4 in Figure 2-a), therefore the control is passed on to the next process or to *phase 2* if expanding the last process. Cycles that go beyond a single *phase 1* instance (e.g. states S_6, S_7, and S_8 in Figure 2-b) do not pose a problem since they contain at least one fully expanded state (e.g. state S_6 in the figure). The reason for the latter is that *phase 2* always performs a full expansion and the two phases are alternated.

3.2 Overview of IMPROVISO

The main challenge in designing a symbolic algorithm for partial-order reduction is to check the *in-stack proviso* efficiently. Since symbolic model checkers perform a breadth-first search, it is not as easy to identify cycles as with explicit-state model checkers (cf. Section 2.2). One possible approach [1] is to over-approximate the set of predecessors with the set of previously visited states. This guarantees that all cycles are correctly identified. However, this over-approximation leads to a limited reduction, since in many cases a full expansion is performed when a partial expansion would be sufficient. In fact, for a given state the set of its predecessors is typically much smaller than the set of all previously visited states.

An important aspect of improved partial-order reduction techniques for symbolic model checking is to better distinguish between the sets of all previously visited states and the set of predecessors. We recognized that the Twophase algorithm has a great potential for application to *symbolic* model checking: the *in-stack proviso* is checked only during *phase 1* and only against states belonging to the stack of the current *phase 1*. Therefore, it is possible to over-approximate the *phase 1* stack with the set of states visited during the current *phase 1*, instead of all previously visited states.

The major ideas of our approach are presented in the following subsections. First, we explain how the transition relations for the different phases are constructed. Then, two improvements over the Twophase algorithm are described: *(i)* the removal of the restriction of *deterministic* transitions in *phase 1*; *(ii)* the addition of a *fixpoint* operation to *phase 1*. Finally, the new ImProviso algorithm is described, followed by a brief comparison with related work.

3.3 Defining the Transition Relation

Symbolic model checking uses an implicit transition relation to perform efficient image computation. Normally, a single transition relation (possibly partitioned) represents all the transitions. In contrast, our approach defines multiple transition relations because it employs two phases and different sets of transitions are executed in each phase.

Phase 1 expands at each step only the transitions of states that are deterministic for a given process. Therefore, a separate transition relation is defined for each process. This transition relation includes only the transitions of a process from states that are deterministic. To build the transition relation of a process, each of its transitions are considered. For each transition, we compute the deterministic states out of all the states at which the transition is enabled. Let us consider three examples:

- An assignment to local variable `v` (line 4 in Figure 3-a) is deterministic for every state at which it is enabled.
- A non-deterministic choice with mutually exclusive conditions (lines 5-8 in Figure 3-a) is also always deterministic.
- A non-deterministic choice with overlapping conditions (lines 9-12 in Figure 3-a) is deterministic only at states where exactly one of the two options is enabled ($w \neq 0$ in the example).

Then, each transition can be restricted to the states that are deterministic for the process it belongs to. The restricted transitions of a process are merged to obtain a transition relation for that process. Such transition relations will be used during *phase 1*.

```
active proctype a()
{
    int v, w;
    v = 1;
    if
    :: v == 0; w = 0;
    :: v == 1; w = 5;
    fi;
    if
    :: w >= 0; w = 1;
    :: w <= 0; w = -1;
    fi
}
```

(a)

```
chan q = [1] of {int};
int x;
active proctype receiver()
{
    int a;
    q ? a;
}
active proctype sender()
{
    q ! 1;
}
active proctype another1()
{
    x = 1;
}
active proctype another2()
{
    x = 2;
}
```

(b)

Fig. 3. Simple Promela examples.

Phase 2, on the other hand, expands all transitions. Therefore, its transition relation consists of all the processes and transitions and can be computed in the usual way.

3.4 Dropping the Determinism

Phase 1 considers one process P at a time. One of the requirements for the current state to be expanded is that only one transition of P is enabled. As a consequence, only one single path is generated during *phase 1*. Therefore, no backtracking is necessary, simplifying the implementation of an explicit-state algorithm.

This requirement is not necessary for the correctness of the algorithm. It is possible to remove such requirement while still preserving safety properties. Our new algorithm, ImProviso, hence takes advantage of multiple enabled transitions in a single process.

In particular, the states that can be expanded during *phase 1* only need to satisfy the following three conditions:

- the enabled transitions of P commute with transitions that can be executed by another process at any point in the future;
- executing any of the enabled transitions of P does not disable transitions that can be executed by another process at any point in the future;
- a transition executed by another process at any point in the future cannot disable or enable any of the transitions of P defined at the program counter location of P in s.

Therefore, in practice, it is sufficient to check that every transition t of P

defined at the program counter location of P in s satisfies the following:

- t does not access any global variable;
- one of 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 to a channel and the channel is not full in s;

As a consequence of this new definition, the construction of the transition relations for *phase 1* is simplified because it is no longer necessary to determine which states have only one transition enabled for each process. At the same time, we are able to apply the reduction in more cases. In particular, we are able to perform as well a reduction as Spin on an example that the Twophase authors used to show how their tool performs a worse reduction than Spin.

3.5 Fixpoint Computation During Phase 1

The transitions expanded during *phase 1* of Twophase do not affect each other and all interleavings thereof are equivalent. This phase considers the processes in a *fixed order* and therefore, only one of all possible interleavings is explored. This is the key of the reduction.

Transitions of one process are expanded as long as the current state is deterministic for the process: the algorithm generates the next state by expanding the only transition that is enabled. When the current state is not deterministic for the process, the next process is considered. When the last process cannot be expanded any further, the current state is passed on to *phase 2* for full expansion. However, the state that is passed on to *phase 2* by this algorithm might still be deterministic for one of the previous processes.

As an example, consider the four processes in Figure 3-b: the first two processes are a receiver and a sender which communicate using a channel, the other two processes set the value of a global variable. *Phase 1* will consider the processes in a fixed order, for instance, the one given by the ordering of the processes in the source. In the initial state the queue is empty and the value of `x` is zero. The initial state is not deterministic for the first process (*receiver*) because no transition is enabled (the queue is empty, so the read operation is disabled). The next process (*sender*) is able to take a step and a new state is reached in which the queue contains the value 1. This new state is not deterministic for process *sender* as no transition is enabled at this point. Moving on to the next process (*another1*), no transition is taken because the only enabled transition refers to a global variable and therefore cannot be safely executed during *phase 1*. The same situation occurs when considering the last process (*another2*). At this point *phase 1* terminates.

However, the current state (in which the queue contains the value 1) is deterministic for process *receiver*. The Twophase algorithm does not consider processes more than once during the same *phase 1*, and a full expansion is

```
1:  global Frontier;
2:  global VisitedStates;
3:
4:  procedure Phase1()
5:  begin
6:    local Moved := true;
7:    local Stack := Frontier;
8:
9:    while Moved do
10:   begin
11:     Moved := false;
12:     for each process I
13:     begin
14:       local NextFrontier := empty;
15:
16:       while Frontier is not empty
17:       begin
18:         local Image := apply(
              TransitionRelationPhase1[I],
              Frontier);
19:         NextFrontier := NextFrontier ∪
              (Image ∩ Stack);
20:         Frontier := Image − Stack;
21:         Stack := Stack ∪ Frontier;
22:
23:         if Frontier is not empty then
24:         begin
25:           Moved := true;
26:         end;
27:       end;
28:
29:       Frontier := NextFrontier;
30:     end;
31:   end;
32:   Frontier := Frontier − Visited;
33:   Visited := Visited ∪ Stack;
34: end;
35:
36: procedure Phase2()
37: begin
38:   local Image := apply(
        TransitionRelationPhase2, Frontier);
39:
40:   Frontier := Image − Visited;
41:   Visited := Visited ∪ Image;
42: end;
43:
44: procedure ImProviso(Init)
45: begin
46:   Frontier := Init;
47:   Visited := Init;
48:
49:   while Frontier is not empty do
50:   begin
51:     Phase1();
52:     Phase2();
53:   end;
54: end;
```

Fig. 4. Pseudo-code of the ImProviso algorithm.

performed at this point: this full expansion causes the exploration of multiple interleavings of the receive operation with the other two processes, which is not necessary.

ImProviso includes an *additional fixpoint computation* in *phase 1*, which guarantees that a state is passed on from *phase 1* to *phase 2*, only if it is not deterministic for all of the processes. At the same time, processes are still considered in a fixed order during each iteration, therefore exploring only one of the possible interleavings.

Although fixpoint computations are natural in symbolic model checking, the proposed improvement could have been added to the Twophase explicit-state algorithm as well. In addition, this improvement of the algorithm guarantees that no transition belonging to *phase 1* will be executed during *phase 2*, making the generation of the transition relation for *phase 2* simpler by including only those transitions which are not included in the *phase 1* transition relations.

3.6 The ImProviso *Algorithm*

The pseudo-code for the ImProviso algorithm for reachability is presented in Figure 4.

The main procedure ImProviso (lines 44-54) initializes the frontier and the set of already visited states to the set of initial states (lines 46-47), and then

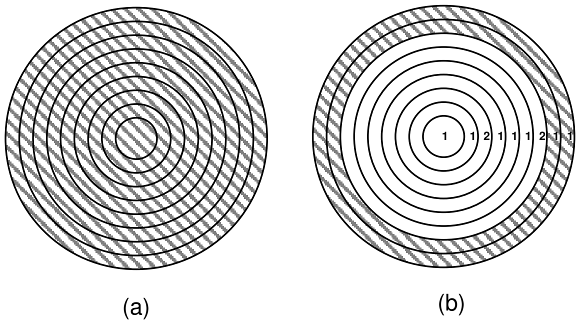

Fig. 5. Part of the visited state space which is used in cycle detection: (a) Alur et al.; (b) IMPROVISO.

the two phases are called alternately until the frontier becomes empty (lines 49-53).

Phase 1 performs a partial expansion of the deterministic states for each process (lines 12-27), by using a transition relation which only includes transitions of deterministic states for process i (*TransitionRelationPhase1[I]*).

If any of the newly reached states (*Image*) has already been visited during the current *phase 1* (i.e. belongs to *Stack*), then such state needs to be expanded by the next process and therefore it is added to *NextFrontier* (line 19). This represents the case where one of the successors belongs to the set of previously visited states and may be part of a cycle. Adding this state to the next frontier guarantees that at least one other transition at that state will be expanded.

The outer most loop (lines 9-31) is repeated as long as at least one of the processes made a step, i.e. generated a new, unvisited state (line 25).

Phase 2 performs a full expansion (lines 39-41) in the classical way. The transition relation used for this step is *TransitionRelationPhase2*, which contains all transition.

3.7 Related Work

The idea of combining partial-order reduction and symbolic model checking is not new, however, our approach has a significant advantage in the way the in-stack proviso needs to be checked. Other approaches presented before include the work by Alur et al. [1] and Kurshan et al. [15]. We briefly review their approaches and relate them to our work. However, a detailed experimental comparison with these approaches was not possible because neither of the tools was freely available.

Alur et al. [1] adapt a partial-order reduction algorithm used in explicit-state model checking to the symbolic case. They expand an *ample* sets of transitions at each step. However, they assume that the entire set of previously visited states in the breadth-first symbolic exploration (Fig. 5-a) corresponds to the stack in the explicit case: revisiting any state is considered a possible cycle, even though a previously visited state may not be a predecessor to the current state along any execution path. Thus, they might detect spurious cycles in the state space and visit an over-approximation of the actual partial-order reduced state space. In contrast, our approach checks for possible cycles only with respect to states visited in the current *phase 1* (Fig. 5-b).

Kurshan et al. [15] propose a completely *static* technique based on pre-processing the model. They statically transform the model into one with a reduced state space by adding information to the system description before it is model checked. Their technique avoids ignoring a transition indefinitely during model checking by fully expanding at least one state from each local cycle, where a local cycle is a cycle in the control flow graph of a single process. However, not every local cycle of transitions may correspond to a cycle in the global state space and thus results in a limited reduction.

4 Experimental Results

We have implemented the ImProviso algorithm in the NuSMV symbolic model checking framework and called the tool NuSMV-ImProviso. NuSMV[7] is the evolution of SMV, the first symbolic model checker developed by Ken McMillan at Carnegie Mellon[19].

In order to show the effectiveness of our approach, we report comparisons with three other tools: *(i)* we compare NuSMV-ImProviso with NuSMV without partial-order reduction to show the significant benefits of adding ImProviso to NuSMV; *(ii)* since ImProviso was inspired by the Twophase algorithm, we compare NuSMV-ImProviso with the *Protocol Verifier* (PV) [21], which implements Twophase; *(iii)* we compare NuSMV-ImProviso with Spin [12], which is one of the leading tools for verification of concurrent systems.

The tools have been run on a 1.4MHz AMD Athlon dual-processor workstation giving each tool up to 1GB of memory.

For our comparisons, we use models written in Promela, which is the input language of Spin and PV. For NuSMV and NuSMV-ImProviso we translate the Promela models into an SMV specification. This conversion has been done automatically using a translator currently under development within our group[16]. The results presented in this section have been obtained using similar options in each of the tools, with the goal of focusing on a comparison of the main algorithms.

Although we attempted to make our comparisons as fair as possible, some differences could not be resolved, giving a slight advantage to Spin and PV over our tool. The reported memory for the NuSMV runs includes the storage of the transition relation while for both Spin and PV the transition relation is encoded as part of the generated verifier and therefore is not included in the memory reported; the run-time for the NuSMV runs includes the time necessary to parse the input and generate the transition relation, while this operation is performed by the verification program generator and the compiler for both

NuSMV				NuSMV-ImProviso		
#states	time	memory		#states	time	memory
4864210	3217.69s	63.6 MB	**Migratory Protocol (2)**	155040	108.63s	56.3 MB
1270	0.87s	6.2 MB	**Stable Marriage (2)**	710	0.84s	7.3 MB
3107	4.26s	10.3 MB	**Stable Marriage (3)**	1275	2.72s	10.4 MB
71495	112.25s	24.7 MB	**Stable Marriage (5)**	10351	31.56s	30.0 MB
2187	0.08s	0.7 MB	**Best (7)**	15	0.06s	0.7 MB
3486780000	0.56s	5.7 MB	**Best (20)**	41	0.34s	5.7 MB
27	0.04s	0.3 MB	**Worst (3)**	15	0.04s	0.3 MB
3486780000	0.46s	5.0 MB	**Worst (20)**	2097150	0.36s	5.0 MB
N/A[1]			**Worst (100)**	2.54E+30	14.34s	14.6 MB

Table 1
Comparison of NuSMV and NuSMV-ImProviso.

Spin and PV and is not included in the their run-times. It should be noted that NuSMV-ImProviso is currently only a prototype and not as optimized as Spin and PV. Many improvements are possible that will the reduce memory requirements and the run-time once implemented.

4.1 NuSMV *vs.* NuSMV-ImProviso

The first set of results (Table 1) is meant to compare NuSMV with our version of the tool NuSMV-ImProviso.

The examples presented in the table belong to the official distribution of PV or have been taken from published work.

- The "Migratory Protocol" example, from the distribution of PV, describes the migratory cache coherency protocol of the Avalanche system.
- The "Stable Marriage" example implements a distributed algorithm for the solution of the stable marriage problem[18]. This example is parameterized by the number of couples.
- The "Best" example is presented in [21] and represents a case where the PV tool gives a better reduction than Spin. This example is parameterized by the number of processes in the system.
- The "Worst" example is presented in [21] and represents a case where Spin performs better reduction than the PV tool . This example is parameterized by the number of processes in the system.

The results show that in many cases the reduction of the state space obtained by using the ImProviso algorithm causes a reduction in the number of states and a significant decrease in run-time. For example, for the "Migratory Protocol", both the states and the run-time are reduced by a factor of 30. The slight increase of memory requirements for some of the examples can be

[1] Results are not available because the model checker ran out of memory or did not complete within 24 hours. The memory limit was set of 1GB.

	NuSMV-ImProviso			PV			SPIN		
	#states	time	memory	#states	time	memory	#states	time	memory
Migratory Protocol (2)	155040	108.63s	56.3 MB	86246	1.00s	4.3 MB	435456	2.34s	42.8 MB
Stable Marriage (2)	710	0.84s	7.3 MB	595	<0.01s	2.2 MB	568	<0.01s	1.5 MB
Stable Marriage (3)	1275	2.72s	10.4 MB	1135	<0.01s	2.2 MB	945	<0.01s	1.5 MB
Stable Marriage (5)	10351	31.56s	30.0 MB	9063	0.14s	2.6 MB	8421	0.03s	2.1 MB
Best (7)	15	0.06s	0.7 MB	15	<0.01s	2.2 MB	2187	0.03s	1.5 MB
Best (20)	41	0.34s	5.7 MB	41	<0.01s	2.2 MB	N/A[1]		
Worst (3)	15	0.04s	0.3 MB	27	<0.01s	2.1 MB	15	<0.01s	1.5 MB
Worst (20)	2097150	0.36s	5.0 MB	N/A[1]			2097150	15.03s	110.6 MB
Worst (100)	2.54E+30	14.34s	14.6 MB	N/A[1]			N/A[1]		

Table 2
Comparison of NuSMV, NuSMV-ImProviso, Spin and PV.

explained by the fact that a BDD representing a smaller set of states is not necessarily smaller than one representing a larger set.

In cases where the reduction in the number of states is more than marginal, a reduction in the memory requirements is typically also present. In fact, the last entry in the table is only verifiable after we introduce partial-order reduction.

4.2 Comparison with Explicit-State Model Checkers

The next table (Table 2) compares NuSMV-ImProviso with Spin and PV using the same examples presented above.

In almost all the cases, the number of states in the reduced state space of NuSMV-ImProviso is close to the number of states explored by the explicit-state model checkers: this shows that the over-approximation of the proviso condition in NuSMV-ImProviso is indeed tight.

NuSMV-ImProviso is able to verify the largest example presented while both explicit-state tools run out of memory. This shows how the use of symbolic representations for the state space can be crucial for the verification of large examples.

However, for examples that can be handled by both the symbolic and explicit-state tools, the explicit state tools almost always use less memory and less time. This is in part due to the fact that both Spin and PV generate a customized verifier for each model that needs to be verified, thus generating a very optimized verifier for the model. We believe that further optimizations of NuSMV-ImProviso will make it more competitive in these cases.

It is also relevant to notice that in both the “Best” and “Worst” examples, NuSMV-ImProviso always matches the better of the the two algorithms (Spin or PV) in terms of state space reduction: we believe that this is due to the removal of the deterministic constraint on the transitions. This allows for ample sets of size larger than one, but at the same time conserves the concept of Twophase.

	Non-PO								
	NuSMV			PV			SPIN		
#	#states	time	memory	#states	time	memory	#states	time	memory
2	70	0.11s	1.1 MB	70	<0.01s	2.1 MB	70	<0.01s	1.5 MB
3	488	0.57s	4.6 MB	488	0.03s	2.2 MB	488	<0.01s	1.5 MB
4	3576	6.77s	10.6 MB	3576	0.38s	2.5 MB	3576	0.10s	2.3 MB
8	N/A[1]			N/A[1]			N/A[1]		

	PO								
	NuSMV-ImProviso			PV			SPIN		
#	#states	time	memory	#states	time	memory	#states	time	memory
2	48	0.10s	1.0 MB	48	0.04s	2.1 MB	48	0.02s	1.5 MB
3	209	0.31s	3.0 MB	209	<0.01s	2.2 MB	209	<0.01s	1.5 MB
4	922	1.77s	10.4 MB	922	0.04s	2.2 MB	922	<0.01s	1.7 MB
8	306903	3553.86s	381.8 MB	306903	28.62s	60.4 MB	306903	11.82s	232.8 MB

Table 3
The leader election protocol with NuSMV, NuSMV-ImProviso, PV, and Spin.

4.3 The Leader Election Protocol

The next table (Table 3) shows the details 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 on partial-order reduction [1,15,14], both in the explicit-state and the symbolic model checking domain. The example models a distributed algorithm used to determine which node in an unidirectional ring has the highest identifier. The example is parameterized by the number of nodes that belong to the ring.

It can be noted from Tables 3 and 4 ("Non-PO"), that without partial-order reduction, our translation produces equivalent models with the same number of states as the other tools. Interestingly, the number of states after the reduction are the same, showing that the partial-order reduction of all the three algorithms are equally effective on this example.

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. However, this is no longer true when we consider a slightly modified and more difficult example.

Table 4 shows the "Leader Election" example with a modification: as pointed out in [1], the example presented as before and in [14], only represents a particular case for a fixed assignment of identifiers to nodes. Hence, it is not representative of all the different cases that the algorithm was meant to deal with. In order to verify all possible assignments of identifiers to nodes, it is sufficient to consider all possible assignments of identifiers in the range between 0 and $n-1$, where n is the number of nodes.

As can be seen from the Table 4, this change to the model causes the number of states to grow exponentially as the number of nodes increases even after

	Non-PO								
	NuSMV			PV			SPIN		
#	#states	time	memory	#states	time	memory	#states	time	memory
2	187	0.17s	3.0 MB	187	<0.01s	2.1 MB	187	<0.01s	1.5 MB
3	5602	5.61s	12.5 MB	5602	0.32s	2.6 MB	5602	0.07s	2.4 MB
4	473173	650.25s	62.9 MB	473173	46.62s	49.1 MB	473173	13.58s	119.7 MB
5	N/A[1]			N/A[1]			N/A[1]		

	PO								
	NuSMV-ImProviso			PV			SPIN		
#	#states	time	memory	#states	time	memory	#states	time	memory
2	119	0.17s	3.3 MB	139	<0.01s	2.1 MB	119	<0.01s	1.5 MB
3	2566	2.14s	11.7 MB	3298	0.12s	2.4 MB	2566	0.07s	1.9 MB
4	135173	133.69s	37.6 MB	167173	6.99s	18.9 MB	135173	1.81s	34.3 MB
5	7699370	11635.00s	829.2 MB	N/A[1]			N/A[1]		

Table 4
The leader election protocol with non-deterministic initial state.

applying partial-order reduction. Even for just 5 nodes, it is already impossible for both Spin and PV to verify the system, and a more efficient representation like the one used by the symbolic model checker becomes necessary.

In particular, all the different assignments of identifiers to the nodes generate a set of initial states: the explicit-state model checkers need to visit all the states reachable from these initial states individually. One advantage of a symbolic approach (besides a more compact representation) is the ability to execute all transitions enabled in the initial states at the same time. 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 algorithm called ImProviso 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*. ImProviso was inspired by the Twophase partial-order reduction algorithm used by the explicit-state model checker PV, but this algorithm has been revisited to better exploit the potential of symbolic exploration. We extended the approach in several ways, including the introduction of an added fixpoint computation step, and the removal of the requirement to restrict the applicability to deterministic transitions.

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

In the future, we plan to extend the ImProviso algorithm to verify properties in temporal logics and intend to further improve 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 include making the NuSMV-ImProviso algorithms for symbolic computation more efficient for the case of software as they are currently optimized for hardware.

References

[1] Alur, R., R. K. Brayton, T. A. Henzinger, S. Qadeer and S. K. Rajamani, *Partial-order reduction in symbolic state space exploration*, in: *Computer Aided Verification*, 1997, pp. 340–351.

[2] Ball, T. and S. K. Rajamani, *Automatically validating temporal safety properties of interfaces*, Lecture Notes in Computer Science **2057** (2001).

[3] Blair, M., S. Obenski and P. Bridickas, *Patriot missile software problem*, Technical Report GAO/IMTEC-92-26 (1992).

[4] Bryant, R. E., *Graph-based algorithms for Boolean function manipulation*, IEEE Transaction on Computers **C-35** (1986), pp. 677–691.

[5] Burch, J. R., 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* (1990), pp. 1–33.

[6] Chandra, S., P. Godefroid and C. Palm, *Software model checking in practice: An industrial case study*, in: *Proc. of 24th International Conference on Software Engineering*, 2002, pp. 431–441.

[7] Cimatti, A., 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. G. Larsen, editors, *Computer Aided Verification - Proc. of the 14th International Conference*, LNCS **2404**, Springer, Copenhagen, Denmark, 2002 .

[8] Clarke, E. M., 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* (1993), pp. 5–20.

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

[10] Godefroid, P. and P. Wolper, *A partial approach to model checking*, in: *Logic in Computer Science*, 1991, pp. 406–415.

[11] Henzinger, T. A., R. Jhala, R. Majumdar and G. Sutre, *Lazy abstraction*, in: *Symposium on Principles of Programming Languages*, 2002, pp. 58–70.

[12] Holzmann, G. J., *The model checker SPIN*, Software Engineering **23** (1997), pp. 279–295.

[13] Holzmann, G. J., *Software analysis and model checking*, in: *Proc. of 14th International Conference on Computer Aided Verification*, 2002.

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

[15] Kurshan, R., V. Levin, M. Minea, D. Peled and H. Yenigün, *Combining software and hardware verification techniques*, Formal Methods in System Design **21** (2002), pp. 251–280.

[16] Lerda, F., *Translating Promela into SMV for partial-order reduction*, Technical Report To be published, SCS - Carnegie Mellon (2003).

[17] Lions, J. L., *Arianne 5: Flight 501 failure*, Technical report, European Space Agency (1996).

[18] Lluch-Lafuente, A., L. Edelkamp and S. Leue, *Partial order reduction in directed model checking*, in: *SPIN*, 2002, pp. 112–127.

[19] McMillan, K. L., "Symbolic Model Checking - An Approach to the State Explosion Problem," Ph.D. thesis, SCS - Carnegie Mellon (1992).

[20] Musuvathi, M., D. Y. Park, A. Chow, D. R. Engler and D. L. Dill, *CMC: A pragmatic approach to model checking real code.*

[21] Nalumasu, R. and G. Gopalakrishnan, *An efficient partial order reduction algorithm with an alternative proviso implementation*, Formal Methods in System Design **20** (2002), pp. 231–247.

[22] Newman, M., *Software errors cost U.S. economy $59.5 billion annualy*, Technical Report NIST 2002-10, National Institute of Standards and Technology (2002).

[23] Peled, D., *Combining partial order reductions with on-the-fly model-checking*, in: *Proceedings of CAV'94* (1994), pp. 377–390.

[24] Stephensons, A., 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).

[25] Valmari, A., *A stubborn attack on state explosion*, in: *Proceedings of Computer Aided Verification*, 1990, pp. 25–42.

[26] Visser, W., K. Havelund, G. Brat, S. Park and F. Lerda, *Model checking programs*, Automated Software Engineering **10** (2003), pp. 203–232.