Partial Order Reduction in Directed Model Checking

Alberto Lluch-Lafuente Stefan Edelkamp
Stefan Leue

Institut für Informatik
Albert-Ludwigs-Universität
Georges-Köhler-Allee
D-79110 Freiburg
email: {lafuente,edelkamp,leue}@informatik.uni-freiburg.de

Technical Report No.162

Report Date: October 19, 2001

Abstract

Partial order reduction is one of the most effective techniques for avoiding the state explosion problem that is inherent to explicit state model checking of asynchronous concurrent systems. It exploits the commutativity of concurrently executed transitions in interleaved system runs in order to reduce the size of the explored state space. Directed model checking on the other hand addresses the state explosion problem by using guided search techniques during state space exploration. As a consequence shorter errors trails are found and less search effort is required than when using standard depth-first or breadth-first search. We analyze how to combine directed model checking with partial order reduction methods and give experimental results on how the combination of both techniques perform.

1 Introduction

Model checking [3] is a formal analysis technique for the verification of hardware and software systems. Given a model of the system and a property specification, typically formulated in some temporal logic formalism, the state space of the model is analyzed to check whether the property is valid or not. The main limitation of this method is the size of the resulting state space, known as the *state explosion problem*. State space explosion occurs due to non-determinism in the model introduced by data or concurrency.

Different approaches have been proposed to tackle this problem. One of the most successful techniques is partial order reduction [19]. This method explores a reduced state space by exploiting the independence of concurrently executed events. Partial order reduction is particularly efficient in asynchronous systems, where many interleavings of events are equivalent with respect to a given property specification. Considering only one or a few representatives of one class of equivalent interleavings leads to drastic reductions in the size of the state space to be explored.

Another technique that has been suggested in dealing with the state space explosion problem is the use of heuristic search techniques. They apply state evaluation functions to rank the set of successor states and to decide where to continue the search. Applying such methods often allows to find errors at optimal or sub-optimal depths and to find errors in models in which "blind" search strategies like depth-first and breadth-first search exceed the available time and space resources. Optimal or near-to optimal solutions are particularly important for designers to understand the sequence of steps that lead to an error, since shorter trails are likely to be more comprehensible than longer ones. In protocol verification, heuristic search model checking has been shown to accelerate the search for finding errors [5] and to shorten already existing long trails [7].

In this paper we will focus on safety error detection in model checking and establish a hierarchy of reduction conditions for classifying which methods applies to which classes of heuristic search algorithm. Moreover, we prove a general correctness result for partial order reduction in checking safety properties. To the best of our knowledge, at the time of writing none of the steadily growing number of publications addressing heuristic search in model checking [5, 6, 7, 11, 12, 17, 22] has analyzed how to combine guided search with partial order reduction.

The paper is structured as follows. Section 2 gives some background on directed model checking. Section 3 discusses partial order reduction and a hierarchy of conditions for its application to different search algorithms. This section also addresses the problem of optimality in the length of the counterexamples, since partial order methods usually do not guarantee finding shortest error trails which is an objective in directed model checking. Section 4 presents experimental results showing that partial order reduction and directed model checking complement each other even better than expected. Section 5 summarizes the results and concludes the paper.

2 Directed Model Checking

During the verification process of a concurrent system, analysts have different expectations at different times [12]. In a first *exploratory* mode, one wishes to find errors fast. In a second *fault-finding* mode one expects meaningful error trails. Early approaches [17, 22] propose the use of best-first search in order to accelerate the search for error states. Further approaches [6, 5, 7, 12] propose the full spectrum of classical heuristic search strategies for the verification process in order to accelerate error detection and to provide optimal or near-to-optimal trails. Most of these techniques can be applied to the detection of safety properties only or for shortening given error traces corresponding to liveness violations [7].

Contrary to blind search algorithms like depth- and breadth-first search, heuristic search exploits information of the specific problem being solved in order to guide the search. Estimator functions approximate the distance from a given state to a set of goal states. The values provided by these functions decide in which direction the search will be continued. Two of the most frequently used heuristic search algorithms are A* and IDA*. In the following we briefly introduce both algorithms and consider different heuristic estimates to be applied in the context of directed model checking for error detection. For this setting, we interpret error states as goal nodes in an underlying graph representation of the state space with error trails corresponding to solution paths.

2.1 A*

A* [9] is depicted in Figure 1. The state space is divided into three sets: the set *Open* of visited but not yet expanded states, the set *Closed* of visited and expanded states and the set $S \setminus (\mathit{Closed} \cup \mathit{Open})$ of not yet visited states. The algorithm performs the search by expanding states of the *Open* set and moving them to the *Closed* set. A successor of an expanded state is either new, in which case it is added to the *Open* set, or old. In this case and if the new path to the state is shorter than the previous one, the stored state is moved into the *Open* list or updated with the new path information. Therefore, in contrast to Dijkstra's shortest path algorithm, states are possibly *re-opened* during the search.

The algorithm selects the next state u to be expanded according to the minimal value of a cost function $f(u)$ applied to all possible successor states. $f(u)$ is computed as the sum of the length of the path from the start state $g(u)$ and the estimated distance to a goal state $h(u)$. If h is a lower bound of the distance to a goal state, then A* finds the shortest path to the goal state.

A* can be generalised to a weighted version if the cost function is weighted as follows: $f(u) = w * g(u) + (1-w) * h(u)$, $0 \leq w \leq 1$. The resulting algorithm is called weighted A* (WA*). If $w = 1$ or $h \equiv 0$ WA* performs breadth-first search, if $w = 0$ WA* is called best-first search and if $w = 0.5$ we have pure A*.

2.2 Iterative-deepening A*

Iterative-deepening A* [15], IDA* for short, is a refinement of the brute-force depth-first iterative deepening search (DFID). While DFID performs successive iterations with increasing search depth, in IDA* increasing cost bounds are used to limit search iterations. The cost bound f of a state is the same as in A*. Similar to A*, IDA* guarantees optimality if the estimator is a lower bound.

IDA* can be enhanced by the use of various data structures to store states. The simplest version of IDA* depicted in Figure 2 does not record visited states

```
procedure A*
begin
  $\underline{Open \leftarrow \{(start\_state, h(start\_state))\};}$ $Closed \leftarrow \emptyset$;
  while $(Open \neq \emptyset)$ do
    $u \leftarrow deletemin(Open)$; $insert(Closed, u)$;
    if $error(u)$ then
      exit $ErrorFound$;
    end if;
    for each successor $v$ of $u$ do
      $f'(v) \leftarrow f(u) + 1\underline{+h(v) - h(u);}$
      $\underline{\textbf{if } search(Open, v) \textbf{ then}}$
        if $(f'(v) < f(v))$ then
          $DecreaseKey(Open, (v, f'(v))$;
        end if;
      $\underline{\textbf{else if } search(Closed, v) \textbf{ then}}$
        $\underline{\textbf{if } (f'(v) < f(v) \textbf{ then}}$
          $\underline{delete(Closed, v);}$
          $\underline{insert(Open, (v, f'(v));}$
        $\underline{\textbf{end if};}$
      else $insert(Open, (v, f'(v))$;
      end if;
    end do;
  end do;
end procedure;
```

Figure 1: The A* Algorithm.

at all. At the expense of an increase in node expansions, space consumption remains linear in the search depth since duplicates cannot be identified. Due to the typically large number of duplicates this approach is not suitable for the domain of model checking.

A more useful state storage scheme for IDA* is that of transposition tables [23] to store already visited states. This approach requires to additionally maintain the smallest depth for each state and to enforce revisiting of states when encountered on shorter generating paths.

2.3 Heuristic Estimates

The above presented search algorithms require suitable estimator functions. In model checking, such functions approximate the number of transitions for the system to reach a target state from a given state. In our setup we will henceforth

```
**procedure IDA***
**begin**
  $threshold \leftarrow h(start\_state)$;
  **do**
    $new\_threshold \leftarrow \infty$;
    search($start\_state$);
    $threshold \leftarrow new\_threshold$;
  **end do;**
**end procedure;**

**procedure search**($s$,$threshold$)
**begin**
  **if** $error(s)$ **then**
    **exit** $ErrorFound$;
  **end if;**
  **for** each successor $s'$ of $s$ **do**
    **if** $g(s') + h(s') \leq threshold$ **then**
      search($s'$,$threshold$);
    **else if** $g(s') + h(s') < new\_threshold$ **then**
      $new\_threshold \leftarrow g(s') + h(s')$;
    **end if;**
  **end do;**
**end procedure;**
```

Figure 2: Simple IDA*

assume that the target state corresponds to an error in the model. During the model checking process, however, an explicit error state is not always available. In fact, in many cases we do not know if there is an error in the model at all. We distinguish the cases when errors are unknown and when error states are explicit.

The formula-based heuristic [5] constructs an error function that describes the error in order to derive an estimate for the distance to an error state. For example, the error formula for an invariant is defined as the negation of the invariant. Given an error formula f and starting from state s, a heuristic function $h_f(s)$ is constructed for estimating the number of transitions needed until a state s' is reached, where $f(s')$ holds. Constructing an error formula for deadlocks is not trivial. In [5] we discuss various options for determining heuristic estimates, including formula based approaches, an estimate based on the number of active processes, and an estimate derived from user-provided characterisations of local control states as deadlock-prone. If an explicit error state is given, refined estimates that exploit the information of this state can be devised, for instance the

Hamming distance and the FSM distance [7]. The Hamming distance defined as the number of bits that differ from a given state to the error provides an estimate that is not a lower bound and, therefore, does not guarantee optimality. To the contrary, the FSM distance heuristic is a lower bound. It is defined as the sum of the local distances in the state transition graph of the different processes of the system. Let $D_i(u, v)$ be the distance from state u to state v in the state transition graph of process i, $pc_i(s)$ be the local state of process i in global state s, p the number of process in the system and s' the error state. The FSM distance is defined as $h_{fsm} = \sum_{i=1}^{p} D_i(pc_i(s), pc_i(s'))$. This function is not only a lower bound, but may also lead to equivalent error states at smaller search depths, while the Hamming distance estimate guides the search precisely into the targeted error state [7].

3 Partial Order Reduction

Partial order reduction methods exploit the commutativity of asynchronous systems in order to reduce the size of the state space. The resulting state space is constructed in such a manner that it is equivalent to the original one with respect to the specification. Several partial order approaches have been proposed, namely those based on "stubborn" sets [21], "persistent" sets [8] and "ample" sets [20]. Although they differ in detail, they are based on similar ideas. Due to its popularity, in this paper we mainly follow the ample set approach. Nonetheless, most of the reasoning presented in this paper can easily be adjusted to any of the other approaches.

3.1 Stuttering Equivalence of Labeled Transition Systems

Our approach is mainly focused to the verification of asynchronous systems where the global system is constructed as the asynchronous product of a set of local component processes following the interleaving model of execution. Such systems can be modeled by labeled transitions systems.

A labeled finite transition system is a tuple $\langle S, S_0, T, AP, L\rangle$ where S is a finite set of states, S_0 is the set of initial states, T is a finite set of transitions such that each transition $\alpha \in T$ is a partial function $\alpha : S \to S$, AP is a finite set of propositions and L is a labeling function $S \to 2^{AP}$.

An execution of a transition system is defined as a sequence of states interleaved by transitions, i.e. a sequence $s_0 \alpha_0 s_1 \ldots$, such that state s_0 must be a state of S_0 and for each $i \geq 0, s_{i+1} = \alpha(s_i)$.

The algorithm for generating a reduced state space is very simple. It tries to explore only some of the successors of a state. A transition α is *enabled* in a state s if $\alpha(s)$ is defined. The set of enabled transitions from a state s is usually called the *enabled set* and denoted as *enabled*(s). The algorithm selects and follows only a subset of this set called the *ample set* and denoted as *ample*(s).

Partial order reduction techniques are based on the observation that the order in which some transitions are executed is not relevant. This leads to the concept of independence between transitions. Two transitions $\alpha, \beta \in T$ are independent if for each state $s \in S$ the following two properties hold:

1. *Enabledness* is preserved: $\alpha \in \mathit{enabled}(\beta(s))$ and $\beta \in \mathit{enabled}(\alpha(s))$, i.e. α and β do not disable each other.

2. α and β are *conmutative*: $\alpha(\beta(s)) = \beta(\alpha(s))$, i.e. executed in any order α and β lead to the same global state.

A further fundamental concept is the fact that some transitions are *invisible* with respect to the property specification. A transition α is invisible with respect to a set of propositions P if for each state $s, s' \in S$ such that $s' = \alpha(s)$, $L(s) \cap P = L(s') \cap P$.

We now present the concept of *stuttering equivalence.* A *block* is defined as a finite execution of identically labeled states. Intuitively, two executions are stuttering equivalent if they can be defined as a concatenation of blocks such that the i-th block in one of the executions has the same label as the i-th block in the other execution, for each $i > 0$. Figure 3 depicts two stuttering equivalent paths.

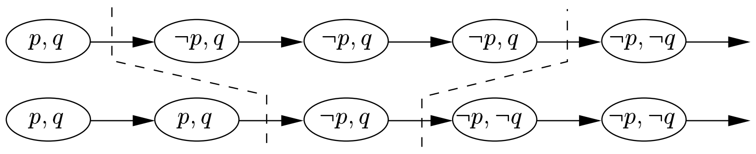

Figure 3: Stuttering equivalent executions.

Two transition systems are stuttering equivalent if and only if they have the same set of initial states and for each execution in one of the systems starting from an initial state there exists a stuttering equivalent execution in the other system starting from the same initial state. It can be shown that LTL$_{-X}$[1] formulae cannot distinguish between stuttering equivalent transition systems [3]. In other words, if M and N are two stuttering equivalent transition systems, then M satisfies a given LTL$_{-X}$ specification if and only if N also does.

3.2 Ample Set Construction for LTL$_{-X}$

The main goal of the ample set construction is to produce an ample set such that the reduced state space is stuttering equivalent to the full state space. Significant reductions can be expected from this reduction without requiring a high computational overhead. For a given property specification P the following four conditions on a set of successor states of a given state s are necessary and sufficient for performing an ample set construction on this set.

[1]LTL$_{-X}$ is the linear time temporal logic without the next-time operator X.

Condition C0: The ample set of s is void exactly when the enabled set of s is void.

Condition C1: Along every path in the full state space that starts at s, a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occurring first.

Condition C2: If a state s is not fully expanded, then each transition α in the ample set must be invisible with regard to P.

Condition C3: If for each state of a cycle, a transition α is always enabled, then α must be selected in the ample set of some of the states of the cycle.

As shown in [3] conditions **C0** and **C2** are easy to check and do not depend on the search algorithm[2]. Condition **C1** is also independent from the search algorithm, but more complicated to verify[3]. In the next we focus on condition **C3**. We will see that it is dependent on the search algorithm. **C3** is commonly over-approximated using a condition $\mathbf{C3}_{cycle}$ that states that each cycle must contain at least one state that is fully expanded:

$\mathbf{C3}_{cycle}$: Every cycle in the reduced state space contains at least one state that is fully expanded.

$\mathbf{C3}_{cycle}$ reduces the verification problem for **C3** to detecting cycles during the search. Cycles can easily be established in depth-first search: Every cycle contains a *backward edge*, i.e. an edge that links back to a state that is stored on the search stack. Consequently, avoiding backward edges in the ample set ensures satisfaction of $\mathbf{C3}_{cycle}$ when using depth-first search or iterative deepening A* (IDA*), since both methods perform a depth-first traversal. The resulting stack-based characterization $\mathbf{C3}_{stack}$ can be stated as follows:

$\mathbf{C3}_{stack}$: If a state s is not fully expanded, then no transition in $ample(s)$ leads to a state on the search stack.

The example on the left of Figure 4 illustrates how $\mathbf{C3}_{stack}$ is used. The set of enabled transitions in state s is $\{\alpha_1, \ldots, \alpha_n\}$. Transition α_1 closes a cycle on the stack and cannot be included in any ample set candidate, except when the state is fully expanded. Therefore, the set of transitions $\{\alpha_2\}$ is a valid candidate, while $\{\alpha_1, \alpha_2\}$ and $\{\alpha_1\}$ are examples of invalid ample sets.

Following [2] and [1], enforcing the cycle is based on the observation that for a cycle to exist, it is necessary to reach an already visited state. Using this idea will lead to weaker reductions, since it is known that the state spaces of concurrent systems usually have a high density of duplicate states. The resulting condition is defined as:

[2]We say that a condition is dependent on a search algorithm if the complexity of checking the condition depends on the algorithm

[3]In fact it has been shown to be at least as hard as checking reachability for the full state space. The commonly used method [3] for constructing the ample set satisfies the **C1** using a sub-optimal approximation.

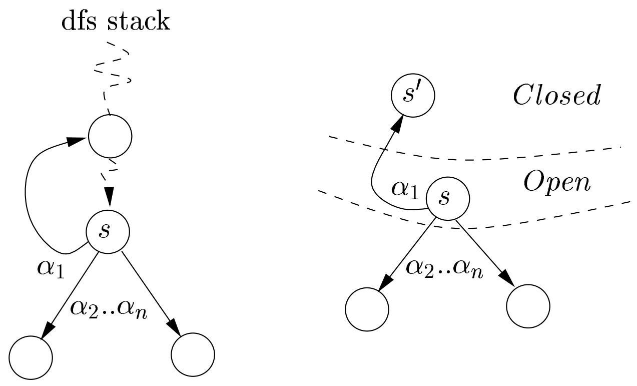

Figure 4: Example for depth-first search (left) and A* (right).

C3$_{duplicate}$: If a state s is not fully expanded, then no transition in *ample*(s) leads to an already visited state.

We use the example on the right of Figure 4 to illustrate this condition. Transition α_1 leads to a state s' that lies below the search horizon defined by the *Open* set, i.e., s' has already been visited when state s is expanded. Condition **C3**$_{duplicate}$ forbids s' in any ample set if s is not fully expanded. Hence, $\{\alpha_1\}$ and $\{\alpha_1, \alpha_2\}$ are examples of not valid ample set. On the other hand, the set $\{\alpha_2\}$ is not refuted.

3.3 Ample Set Construction for Safety Properties

The authors of [10] propose an approximation of the **C3** condition that can be applied when checking safety properties. This condition is defined as follows:

C3$^-_{stack}$: If a state s is not fully expanded, then at *least one* transition in *ample*(s) does not lead to a state on the search stack.

Consider again the example on the left Figure 4. Condition **C3**$^-_{stack}$ does not characterize the set $\{\alpha_1\}$ as a valid candidate for the ample set. Contrary to **C3**$_{stack}$, condition **C3**$^-_{stack}$ accepts $\{\alpha_1, \alpha_2\}$ as a valid ample set, since at least one transition (α_2) of the set leads to a state that is not on the search stack of the depth-first search. Hence, **C3**$^-_{stack}$ is not sufficient to guarantee **C3** which is necessary for checking liveness properties correctly.

We now define a modification of **C3** condition that together with **C0**-**C3** is sufficient and necessary to guarantee a correct reduction for safety properties.

C3$^-$: If a transition α is enabled in every state, then α must be selected in the ample set of some of the states of the *state space*.

This condition is implicitly defined in [10]. It is a relaxation of **C3** that is only correctly applicable to the verification of safety properties, which is the focus of our approach.

Condition $\mathbf{C3}^{-}_{stack}$ cannot be used with A*, since cycles cannot be efficiently detected with this algorithm. Therefore, we propose an alternative condition in order to enforce $\mathbf{C3}^{-}$. It is based on the same idea as $\mathbf{C3}_{duplicate}$.

$\mathbf{C3}^{-}_{duplicate}$ If a state s is not fully expanded, then at *least one* transition in $ample(s)$ does not lead to an already visited state.

Similar to the comparison of conditions $\mathbf{C3}^{-}_{stack}$ and $\mathbf{C3}_{stack}$, Figure 4 illustrates that the set of transitions $\{\alpha_1, \alpha_2\}$ is rejected as ample set by condition $\mathbf{C3}_{duplicate}$, but not by $\mathbf{C3}^{-}_{duplicate}$.

A proof of sufficiency of condition $\mathbf{C3}^{-}_{stack}$ for depth-first search is given in [10]. The proof of sufficiency of condition $\mathbf{C3}^{-}_{duplicate}$ when combined with a depth-first search is given by the fact that $\mathbf{C3}^{-}_{duplicate}$ implies $\mathbf{C3}^{-}_{stack}$. The correctness of $\mathbf{C3}^{-}_{duplicate}$ when combined with A* remains to be proven. We define and proof the following lemma. Applying the lemma to all states in S implies $\mathbf{C3}^{-}$. The proof is done via induction over the order in which states are expanded.

Lemma 1 *For a state $s \in S$ the following is true: when the search terminates each transition $\alpha \in enabled(s)$ have been selected either in $ample(s)$ or in a state s' such that s' has been expanded after s.*

Proof: Let s be the last expanded state. Every transition $\alpha \in enabled(s)$ leads to an already expanded state, otherwise the search would have been continued. Condition $\mathbf{C3}^{-}_{duplicate}$ enforces therefore that state s is fully expanded and the lemma trivially holds for s.

Now suppose that the lemma is valid for those states which expansion order is greater than n. Let s be the n-th expanded state. If s is fully expanded, the lemma trivially holds for s. Otherwise we have that $ample(s) \subset enabled(s)$. Transitions in $ample(s)$ are selected in s. Since $ample(s)$ is accepted by condition $\mathbf{C3}^{-}_{duplicate}$ there is a transition $\alpha \in ample(s)$ such that $\alpha(s)$ leads to a state that has not been previously visited nor expanded. Evidently the expansion order of $\alpha(s)$ is higher than n. Since **C1** implies that the transitions in $\alpha \in ample(s)$ are all independent from those in $ample(s)$ [3], transitions in $ample(s) \setminus enabled(s)$ are still enabled in $\alpha(s)$ and, therefore, contained in $enabled(\alpha(s))$. We have supposed that the lemma holds for $\alpha(s)$ and, therefore, transitions in $enabled(s) \setminus ample(s)$ are executed in $\alpha(s)$ or in a state that is expanded after it. Hence the lemma also holds for s. $\blacksquare$

3.4 Static Reduction in the Ample Set Construction

In contrast to the previous approaches this ample set construction method explicitly exploits the structure of the underlying interleaving system. Recall that the global system is constructed as the asynchronous composition of several components. The authors of [16] present a static partial order reduction method based on the following observation. Any cycle in the global state space is composed of a local cycle, which may be of length zero, in the state transition graph of each component process. Breaking every local cycle breaks every global cycle. The state transition graph of processes of the system are statically analyzed before the global state space generation begins. Therefore, the method is independent from the search algorithm to be used during the verification.

A *sticky* transition is defined as a transition that enforces full expansion of a state. Marking at least one transition in each local cycle as *sticky* guarantees that at least one state in each global cycle is fully expanded, satisfying the cycle condition $\mathbf{C3}_{cycle}$. The resulting $\mathbf{C3}_{static}$ condition is defined as follows:

$\mathbf{C3}_{static}$: If a state s is not fully expanded then no transition in $ample(s)$ is sticky.

Selecting one sticky transition for each local cycle is a simple approach that can be improved. The effect of local cycles on the set of variables of the system can be analyzed in order to establish certain dependencies between local cycles. For example, if a local cycle C_1 has an overall incrementing effect on a variable v, for a global cycle to exist, it is necessary (but not sufficient) to execute C_1 in combination with a local cycle C_2 that has an overall decrementing effect on v. In this case one can select only a sticky transition for this pair of local cycles.

3.5 Hierarchy of C3 Conditions

Figure 5 depicts a diagram with all the presented **C3** conditions. Arrows indicate logical implication. In the rest of the paper we will say that a condition A is stronger than a condition B if A implies B. For example, if the search guarantees $\mathbf{C3}_{stack}$ then $\mathbf{C3}_{cycle}$ is also guaranteed, and we say that $\mathbf{C3}_{stack}$ is stronger than $\mathbf{C3}_{cycle}$. The dashed region contains the conditions that can be correctly used with A*, while the dotted region includes those for IDA*. For a given algorithm, the arrows also denote that a condition will produce better or equal reduction. For example, IDA* in combination with $\mathbf{C3}_{stack}$ will provide better or equal reductions than in combination with $\mathbf{C3}_{duplicate}$.

When the search is performed by A*, only $\mathbf{C3}_{static}$, $\mathbf{C3}_{duplicate}$ and $\mathbf{C3}^{-}_{duplicate}$ can be used for reduction as marked in the dashed region of the figure. Condition $\mathbf{C3}^{-}_{duplicate}$ is preferable to $\mathbf{C3}_{duplicate}$ since it will produce better reductions. Only experimental results can help us to decide whether $\mathbf{C3}_{static}$ is superior to $\mathbf{C3}^{-}_{duplicate}$ or not.

Since IDA*'s traversal order is depth-first, it can be combined with all cycle conditions contained in the dotted region of the Figure.

3.6 Optimality and Partial Order

One of the goals of directed model checking is to find shortest paths to errors. Although from a practical point of view near-to optimal solutions may be sufficient

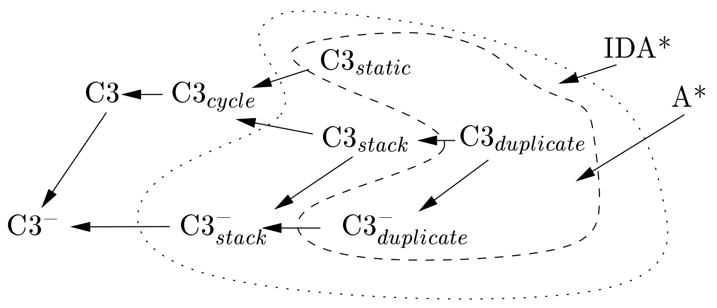

Figure 5: C3 conditions.

to help designers during the debugging phase, finding optimal counterexamples still remains an important theoretical question. Heuristic search algorithms require lower bound estimates for guaranteeing optimal solution lengths. Nevertheless, there are other issues that determine optimality of the search results.

Partial order reduction does not guarantee optimal length of paths to the set of error states in the non-reduced state space. In fact, the shortest path to an error in the reduced state space may be longer than the shortest path to an error state in the full state space. The reason is that the concept of stuttering equivalence does not make assumptions about the length of the blocks. Suppose that the transitions α and β of the state space depicted in Figure 6 are independent and that α is invisible with respect to the set of propositions p. Suppose further that the property we want to check corresponds to the LTL formula $\Box p$. With these assumptions the reduced state space for the example is stuttering equivalent to the full one. The shortest path that violates the invariant in the reduced state space is $\alpha\beta$, which has a length of 2. In the full one the path β is the shortest path to an error state and the error trail has a length of 1. Section 4 gives experimental evidence for loss of optimality when applying partial order reduction.

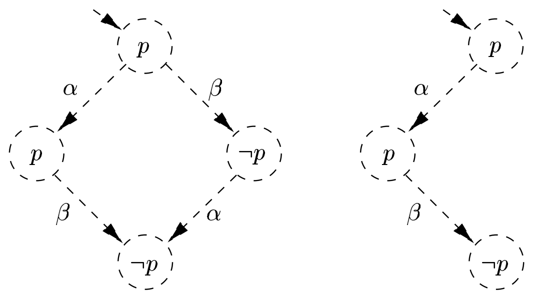

Figure 6: Example of a full state space (left) and a reduction (right).

4 Experiments

The experimental results that we report in this Section have been obtained using our experimental directed model checker HSF-SPIN[4] for which we have implemented the described reduction methods for some of the experiments.

We use a set of Promela models as benchmarks including a model of a leader election protocol[5] [4] (leader), the CORBA GIOP protocol [13] (giop), the telephony model POTS[6] [14] (pots), and a model of a concurrent program that solves the stable marriage problem [18] (marriers). The considered versions of these protocols violate certain safety properties.

4.1 Exhaustive exploration

The objective of the first set of experiments is to show how the different variants of the **C3** condition perform. We expect that stronger **C3** conditions according to hierarchy in Figure 5 lead to stronger reductions in the number of stored and expanded states and transitions.

We use the marriers, leader and giop protocols in our experiments. The POTS model is too large to be explored exhaustively. In this and all following experiments we have selected the biggest configuration of these protocols that can still be exhaustively analyzed. Exploration is performed by depth-first search.

Table 1 depicts the size of the state space as a a result of the application of different **C3** conditions. The first column indicates the size of the explored state space when no reduction is used.

As expected stronger conditions offer weaker reductions. This loss of reduction is especially evident in the second configuration of the giop protocol, where the two conditions potentially applicable in A*, namely $\mathbf{C3}^-_{duplicate}$ and $\mathbf{C3}_{static}$, are worse by about a factor of 4 with respect to the condition that offers the best reductions, namely $\mathbf{C3}^-_{stack}$.

For the marriers and giop protocols the static reduction yields a stronger reduction than condition $\mathbf{C3}^-_{duplicate}$. Only for the leader election algorithm this is not true. This is probably due to the relative high number of local cycles in the state transition graph of the processes in this model, and to the fact that there is no global cycle in the global state space.

Since our implementation of the static reduction considers only the simplest approach where one transition in each cycle is marked as sticky, we assume that the results will be even better with refined methods for characterizing transitions as sticky.

[4]Available at `www.informatik.uni-freiburg.de/~lafuente/hsf-spin`

[5]Available at `netlib.bell-labs.com/netlib/spin`

[6]Available at `www.informatik.uni-freiburg.de/~lafuente/models/models.html`

Model	No Reduction	$\mathbf{C3}_{stack}$	$\mathbf{C3}_{duplicate}$	$\mathbf{C3}^{-}_{stack}$	$\mathbf{C3}^{-}_{duplicate}$	$\mathbf{C3}_{static}$
marriers	96,695	29,501	72,536	29,501	72,536	57,067
leader	54,216	963	1,417	963	1,417	2,985
giop	664,376	65,964	284,083	65,964	284,083	231,102

Table 1: Number of stored states by depth-first search with several reduction methods.

4.2 Error Finding with A* and Partial Order Reduction

The next set of experiments is intended to highlight the impact of various reduction methods when detecting errors with A*. More precisely, we want to compare the two **C3** conditions $\mathbf{C3}^{-}_{duplicate}$ and $\mathbf{C3}_{static}$ that can be applied jointly with A*. Table 2 shows the effect of applying $\mathbf{C3}^{-}_{duplicate}$ and $\mathbf{C3}_{static}$ in conjunction with A*. The table gives the number of stored states (States), transitions performed (Transitions) and the length of the error trail (Length) for each experiment. As expected, both conditions achieve a reduction in the number of stored states and transitions performed. Solution quality is only lost in the case of leader. In the same experiment $\mathbf{C3}_{static}$ requires the storage of more states and the execution of more transitions than $C3^{-}_{duplicate}$. The reasons are the same as the ones mentioned in the previous set of experiments. On the other hand, $C3^{-}_{duplicate}$ produces a longer error trail. A possible interpretation is that more reduction leads to higher probability that the anomaly that causes the loss of solution quality occurs. In other words, the bigger the reduction is, the longer the stuttering equivalent executions and, therefore, the longer the expected trail lengths become.

Model	Reduction	States	Transitions	Length
marriers	no	5,077	12,455	50
	$\mathbf{C3}^{-}_{duplicate}$	2,988	4,277	50
	$\mathbf{C3}_{static}$	1,604	1,860	50
pots	no	6,519	2,668	67
	$\mathbf{C3}^{-}_{duplicate}$	1,662	3,451	67
	$\mathbf{C3}_{static}$	1,662	3,451	67
leader	no	7,172	22,876	58
	$\mathbf{C3}^{-}_{duplicate}$	65	3,190	77
	$\mathbf{C3}_{static}$	399	3,593	66
giop	no	31,066	108,971	58
	$\mathbf{C3}^{-}_{duplicate}$	21,111	48,870	58
	$\mathbf{C3}_{static}$	12,361	24,493	58

Table 2: Finding a safety violation with A* and several reduction methods.

4.3 Error Finding with IDA* and Partial Order Reduction Reduction

In this Section we investigate the effect of partial order reduction when the state space exploration is performed with IDA*. The test cases are the same of the previous section. Partial order reduction with IDA* uses the cycle condition $\mathbf{C3}_{stack}$.

Table 3 depicts the results on detecting a safety error with IDA* with and without applying partial order reduction. The table shows the total number of transitions performed, the maximal peak of stored states and the length of the provided counterexamples. As in the previous set of experiments, solution quality is only lost when applying partial order reduction in the leader election algorithm. On the other hand, this is also the protocol for which the best reduction is obtained. We assume that the reason is the same as indicated in the previous set of experiments.

Model	Reduction	States	Transitions	Length
marriers	no	4,724	84,594	50
	yes	1,298	4,924	50
pots	no	2,422	46,929	67
	yes	1,518	20,406	67
leader	no	6,989	141,668	56
	yes	54	50,403	77
giop	no	30,157	868,184	58
	yes	7,441	102,079	58

Table 3: Finding a safety violation with IDA* with and without reduction.

4.4 Reduction Effect of Heuristic Search and Partial Order

In this Section we are interested in analyzing the combined reduction effect of partial order reduction and heuristic search. More precisely, we have measured the reduction ratio provided by one of the techniques when the other technique is used or not, as well as the reduction ratio of using both techniques simultaneously. The Table on the left of Figure Figure 7 indicates the reduction factor achieved by partial order and heuristic search when error detecting is performed with A*. The figure also includes a diagram that helps to understand the table. The reduction factor due to a given technique is computed as the number of stored states when the search is done without applying the respective technique divided by the number of stored states when the search is done applying the technique. Recall that when no heuristic is applied, A* performs breadth-first search. A search is represented in the diagram by a circle labeled with the applied technique(s), namely heuristic search (H), partial-order reduction (PO) or both (H+PO). The labels of the edges in the diagram refer to the cells of the table which contain the measured reduction factor from one search to other. The leftmost column of the table indicates the technique(s) for which the reduction effect is measured. When testing the reduction ratios of the methods separately, we distinguish whether the other method is applied (C) or not (N).

marriers	N	C
H	2.3	6.5
PO	40.8	117.6
H+PO	267.0	

pots	N	C
H	5.9	8.4
PO	1.4	1.6
H+PO	9.5	

leader	N	C
H	1.9	2.6
PO	2.7	3.2
H+PO	5.9	

giop	N	C
H	1.3	1.3
PO	2.6	2.5
H+PO	3.3	

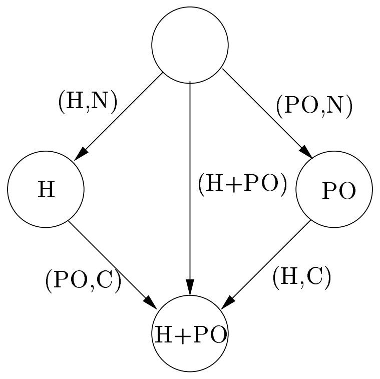

Figure 7: Table with reduction factor due partial order and heuristic search (left) and an explanatory diagram (right).

In most cases the reduction factor provided by one of the techniques when working alone ((H,N) and (PO,C)) improves when the other technique is applied ((H,C) and (PO,C)). This is particularly evident in the case of the marriers model, where the reduction factor is about doubled. The giop model is the only case where reduction ratio is not improved. The expected reduction factor provided by the application of both techniques can be computed as the product of the reduction factor provided by heuristic when partial order is not used and the reduction factor provided by partial order when heuristic is applied (or vice versa). The measured reduction factor provided by the application of both techniques is in fact always near to the expected one. For example, in the case of leader expected reduction factor is $1.9 \times 3.2 = 6.08$ (or $2.7 \times 2.6 = 7.02$) and the measured reduction factor is 5.9. This result shows that both techniques do not interfere significantly with each other when they are combined.

5 Conclusions

When combining partial order with directed search two main problems must be considered. First of all, common partial order reduction techniques require to check a condition which entails the detection of cycles during the construction of the reduced state space. Depth-first search based algorithms like IDA* can easily detect cycles during the exploration. On the other side, heuristic search algorithms like A* are not well-suited for cycle detection. Stronger cycle conditions or static reduction methods have to be used. We have established a hierarchy of approximation conditions for ample set condition **C3** and our experiments show that weaker the condition, the better the effect on the state space search.

The second problem is that partial order reduction techniques do not preserve optimality of the length of the path to error states. In other words, when partial order is used there is no guarantee to find the shortest counterexample that lead to an error, which is one of the core objectives of the paradigm of directed model checking.

Experimental results that we have presented show that partial order reduction has positive effects in combination with directed search strategies. Although optimality is lost in some cases, significant reductions can be achieved even when using A* with weaker methods than classical cycle conditions. Static reduction, in particular, seems to be more promising than other methods applicable with A*. Partial order reduction provides drastic reductions when error detection is performed by IDA*. We have also analyzed the combined effect of heuristics and reduction, showing than in most cases the reduction effect of one technique is accentuated by the other. Experimental results also show that both techniques do not interfere with each other when they are combined.

References

[1] R. Alur, R. Brayton, T. Henzinger, S. Qaderer, and S. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 340–351. Springer, 1997.

[2] C.-T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 241–257, 1996.

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

[4] D. Dolev, M. Klawe, and M. Rodeh. An o(n log n) unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 1982.

[5] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *8th International SPIN Workshop on Model Checking Software*, Lecture Notes in Computer Science 2057, pages 57–79. Springer, 2001.

[6] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.

[7] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Trail-directed model checking. In *Workshop on Software Model Checking*. Electronic Notes in Theoretical Computer Science, Elsevier, 2001. To Appear.

[8] P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke, editor, *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90), Rutgers, New Jersey, 1990*, number 531, pages 176–185, Berlin-Heidelberg-New York, 1991. Springer.

[9] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on on Systems Science and Cybernetics*, 4:100–107, 1968.

[10] G. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Orlando, Fl., June 1992.

[11] G. J. Holzmann. Algorithms for automated protocol verification. *AT&T Technical Journal*, 69(2):32–44, Feb. 1990. Special Issue on Protocol Testing, Specification, and Verification.

[12] L. J. O. Jamieson M. Cobleigh, Lori A. Clarke. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *Proceedings of the 23rd ICSE.*

[13] M. Kamel and S. Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *International Journal on Software Tools for Technology Transfer*, 2(4):394–409, 2000.

[14] M. Kamel and S. Leue. Vip: A visual editor and compiler for v-promela. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, pages 471–486. Springer, 2000.

[15] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[16] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 345–357, 1998.

[17] F. J. Lin, P. M. Chu, and M. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *ACM*, pages 126–135, 1988.

[18] D. McVitie and L. Wilson. The stable marriage problem. *Communications of the ACM*, 14(7):486–492, 1971.

[19] D. Peled. Ten years of partial order reduction. In *Computer Aided Verification*, number 1427 in Lecture Notes in Computer Science, pages 17–28. Springer, 1998.

[20] D. A. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in Systems Design*, 8:39–64, 1996.

[21] A. Valmari. A stubborn attack on state explosion. *Lecture Notes in Computer Science*, 531:156–165, 1991.

[22] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Conference on Design Automation (DAC)*, pages 599–604, 1998.

[23] A. L. Zobrist. A new hashing method with application for game playing. Technical Report 88, University of Wisconsin, Computer Science Department, 1970.