

On Partial-Order Reduction and Trail Improvement in Directed Model Checking

Stefan Edelkamp¹, Stefan Leue², Alberto Lluch-Lafuente²

¹ Fachbereich Informatik, Universität Dortmund, Baroper Str. 301, GB IV, D-44221 Dortmund, Germany
e-mail: stefan.edelkamp@cs.uni-dortmund.de

² Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Georges-Köhler-Allee Geb. 051, D-79110 Freiburg, Germany
e-mail: [leue|lafuente]@informatik.uni-freiburg.de

The date of receipt and acceptance will be inserted by the editor

Abstract. In this paper we address the problem of reconciling trail improvement, partial order reduction and directed explicit state model checking. While directed model checking addresses the problem of finding optimally short counterexamples, trail improvement seeks to find shorter trails to some given error state. Both directed model checking and trail improvement employ heuristic, guided search techniques such as A* or Best First Search in the exploration of the state space. Partial order reduction aims at reducing the size of the state space by exploiting the commutativity of concurrent transitions in asynchronous systems, and we show that this reduction technique can nicely co-exist with directed model checking. We finally illustrate how mitigate the long trails sometimes delivered by partial order reduction techniques and show how two combine heuristic search and partial order reduction improve the length to already provided counterexamples.

Keywords: Directed Model Checking, Trail-directed Model Checking, Heuristic Search, Partial Order Reduction, HSF-SPIN

1 Introduction

The success of model checking [9] as a software verification technology is largely founded in the automatic nature and the error trail reporting capabilities of the algorithms.

Roughly speaking, checking whether or not a system satisfies its specification is done by analyzing the state space of the system. A violation of the specification corresponds to a path in the state space, which is called error trail or counterexample. Explicit-state model checkers usually perform a depth-first search exploration for checking safety properties. Violation of safety properties correspond to finite executions of the system that

are represented by a path from the initial state to some error state. When an error is encountered during the search, the search stack contains a valid execution path from the initial system state into the error state that was found. It is hence easy to provide the user with an error explanation by dumping the search stack as an error trail.

A large number of model checking tools based on explicit state technology have been built [22,29] and successfully applied in practice [33,39].

Explicit state model checking has proven particularly successful in the analysis of concurrent software systems such as communication protocols or embedded real-time systems. Models for this type of systems are characterized by combinatorial state space explosions mainly induced by the concurrent composition of processes. The concurrent nature also causes the execution of these systems to have a high degree of nondeterminism. Explicit state model checkers such as SPIN have developed efficient abstractions and data structures to deal with these characteristics. Most notably, symmetry aspects due to concurrent composition have been studied in [7,18,31], partial order reductions have been proposed to take advantage of the commutativity of independent concurrent transitions [48], and data abstraction techniques have been proposed to reduce very large or even infinite state spaces to finite ones [12,52].

While in early work on model checking the complete verification of the model was of central interest, in recent applications the focus is more on using model checking as a debugging technique for existing requirements, designs or codes [4]. One of the issues of this focus concerns the size of the counterexamples. During debugging, counterexamples are used to understand why errors occur. We strongly believe that shorter counterexamples are easier to understand than large ones.

Our own contributions to this application of model checking are based on the use of heuristic search algo-

gorithms from the AI field. We introduced the concept of *directed explicit state model checking* [16,37]. Traditional model checking algorithms perform an uninformed state space explorations based on depth-first or breadth-first search algorithms. In analogy to AI state space exploration problems, we employ heuristic search algorithms such as A* [27] or best-first search in order to guide the search on the shortest, or a close to shortest, path into a property violating state. Best-first search accelerates the search for error states, while A* produces optimal paths to a target state when the heuristic estimate is admissible, i.e., a lower bound for the actual distance to a target state. As proposed by other authors [10], and taking into our account our own experience in this topic, error debugging in model checking has to be divided into two phases. A first *exploratory* phase faces the problem of quickly finding an error in the system. The next *fault-finding* phase concentrates on finding short counterexamples possibly exploiting errors found in the first phase.

The present paper summarizes and extends part of our previous work on directed model checking, focussing on the application of partial order reduction methods and the improvement of already established counterexamples. The bases are the SPIN 2002 contribution *Partial Order Reduction in Directed Model Checking* and the 2002 *Software Model Checking Workshop* paper *Trail-Directed Model Checking*.

In [16] we have already given links to different forms of directed model checking, including initial work [53], real-time search [3], CTL model checking [5], data flow analysis [10], and Java Verification [24,23] Moreover, we have drawn links to alternative search schemes such as genetic algorithms [21].

In addition to the current contribution, there is other some work concerning the problem of providing meaningful counterexamples that use search algorithms like Dijkstras single-source shortest path or A*. For example, in the domain of scheduling of Timed Systems [3, 2], or data-flow analysis [10]. On the other hand, recent approaches [25,50,49,32] concentrate on the analysis of counterexamples, rather than on improving them. Alternative approaches concentrate on the explanation of counterexamples to show the user what went wrong [26].

Partial order techniques have been studied by many authors. Basically, there exist two main families of partial order techniques. The first one is based on net unfoldings [6,41,44], while the second one is based on so-called diamond properties. We focus on the latter, which are called partial order reduction techniques. Several approaches have been proposed, namely those based on “stubborn” sets [51], “persistent” sets [20] and “ample” sets [47]. Although they differ in their detail, they are based on similar ideas. For an extended survey of partial order reduction methods, we refer the reader to [48].

Structure of Paper. In Section 2 we present a review of the *directed explicit state model checking* approach and its application to protocol verification.

The first contribution of this paper is contained in Section 3. It is a summary of the paper *Trail-Directed Model Checking* on the use of heuristic search to improve already found error trails. This section focuses on the *fault-finding* phase. It relies on the fact that an error trail is available. That trail could be the result of a previous verification run during the *exploratory* phase, or of a simulation run. Contrary to the rest of the paper, in this Section we also consider liveness errors, and not only safety ones.

The second contribution of this paper is an extended summary of the paper *Partial Order Reduction in Directed Model Checking* on the combination of heuristic search with partial order reduction, and is contained in Section 4. Partial order reduction techniques are known to be extremely efficient in reducing the state space to an analyzable size for concurrent systems. This reduction techniques take advantage of the commutativity of concurrent transitions. Existing partial order reduction methods are tailored to depth-first explorations and based on the existence of a search stack, which is not present in heuristic algorithms like best-first search or A*. Hence, one has to apply weaker methods. Another aspect of the application of partial order reduction is that counterexamples may be longer, since portions of the state graph containing optimal error paths can be pruned during the reduction.

Section 5 presents the third contribution of our paper, which analyzes the application of partial order reduction for shortening already established counterexamples. More precisely, we concentrate on the problem of finding shorter paths to an already established error state.

In the last section we summarize our results and outline current and future work e.g. on symmetry reduction, Promela planning and trail-directed Java program verification.

2 Directed Model Checking

In this section we review the key concepts of directed, explicit state model checking, as previously published in [16].

2.1 State Space Search in Explicit State Model Checking

Model checking is a technique to determine the validity of a property for a given model. This approach has proven to be particularly successful in the verification of software designs and code. In this domain, the properties to be checked usually are represented by temporal constraints on the valid execution sequences of the system

and are given in an automata representation or as temporal logic formulae. The models to be checked represent the state space of the system. Model checking algorithms analyze the state space in order to validate whether or not the property holds.

Models and Systems. In the rest of the paper we assume that the system being verified consists of the asynchronous composition of n processes $\mathcal{P}_0, \dots, \mathcal{P}_n$ which state space is represented by a labeled transition system. A labeled finite transition system is a quintuple $\langle S, S_0, T, AP, L \rangle$ 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 $\alpha : S \rightarrow S$, AP is a finite set of propositions and L is a labeling function $S \rightarrow 2^{AP}$. A transition α is said to be enabled in a system s if $\alpha(s)$ is defined. The execution of a transition system is defined as a sequence of states interleaved by transitions, i.e. a sequence $s_0\alpha_0s_1\dots$, such that s_0 is in S_0 and for each $i \geq 0$, $s_{i+1} = \alpha_i(s_i)$.

In current model checkers, the state spaces are either represented symbolically or explicitly. Explicit representations seems to be most adequate for asynchronous concurrent systems, and software systems in general.

Specifications and Errors. Roughly speaking, there are two kinds of temporal properties liveness and safety. Safety properties express that, under certain conditions, a *bad* event will never occur, while liveness properties express that, under certain conditions, a *good* event will ultimately occur.

In our framework we consider a safety error as a finite execution of the system violating a specification property. Safety errors are represented by finite paths in the state space ending in an error state. On the other hand, we consider liveness errors as infinite executions of the system which are represented by finite paths containing a cycle. More precisely, liveness error paths consist on an initial prefix path to a state, and a cycle through that state that contains at least one accepting state.

Verification Algorithms. Checking safety errors can be solved by applying simple reachability algorithms like depth-first search or breadth-first search, while checking liveness errors is done using the a *nested depth-first* search algorithm¹. In explicit-state model checkers, both state traversals are implemented on-the-fly, i.e., the state space is not being entirely generated in one pass and then analyzed in a second pass, instead it is generated in a stepwise fashion as the exploration is proceeding.

A General Search Algorithm. Figure 1 presents a general state expanding search (GSEA) algorithm for the verification of safety properties. The algorithm divides

```

( 1) procedure GeneralStateExpandingAlgorithm( $s$ )
( 2)    $Closed \leftarrow \emptyset$ ;
( 3)    $Open \leftarrow \emptyset$ ;
( 4)    $Open.insert(s)$ ;
( 5)   while not  $Open.empty()$  do
( 6)      $u \leftarrow Open.extract()$ ;
( 7)      $Closed.insert(u)$ ;
( 8)     if  $goal(u)$  then
( 9)       return solution;
(10)    for each  $e \in outgoing(u)$  do
(11)       $v \leftarrow to(e)$ ;
(12)      if  $v \notin Closed$  and  $v \notin Open$  then
(13)         $Open.insert(v)$ ;

```

Fig. 1. A general state expanding search algorithm.

the state space S of the model to be analyzed into three sets: the set *Open* of visited but not yet expanded states, the set *Closed* of visited and expanded states, and the rest. The algorithm performs the search by extracting states from *Open* and moving them into *Closed*. States extracted from *Open* are expanded, i.e., the respective successor states are generated. If a successor of an expanded state is neither in *Open* nor in *Closed* it is added to *Open*. Simple breadth-first and depth-first search can be defined as instances of the general algorithm presented above, where the former implements *Open* as a queue and the latter as a stack.

2.2 Blind Search Algorithms

Model checkers often use blind search algorithms like breadth-first search or depth-first search for checking safety properties. While breadth-first search guarantees to find counterexamples of minimal depth, depth-first search is more memory-efficient in practice. The latter is often preferred, since memory-efficiency is most crucial for explicit-state model checkers in order to tackle the state explosion problem. In addition to depth-first search, the model checker SPIN implements two rather “naive” strategies to reduce the length of error trails for finding shorter or even optimally short error trails.

The first strategy uses a depth-first search that continues the exploration once an error state is found and bounds the search depth to the depth of it. Initially, there is no depth-bound or the depth-bound is set to a certain value by the user. Note that bounding the depth to a value d does not guarantee that every state actually reachable with less than d will eventually be visited by the algorithm. As a consequence, there is no guarantee to find the optimal counterexample.

This is illustrated in the search tree of Figure 2. The search visits state v for the first time (down left copy) and stores it. Goal state g cannot be reached due to the

¹ For the sake of brevity and since our work focuses on safety properties, we refer to [9] if there is interest on how liveness properties are checked.

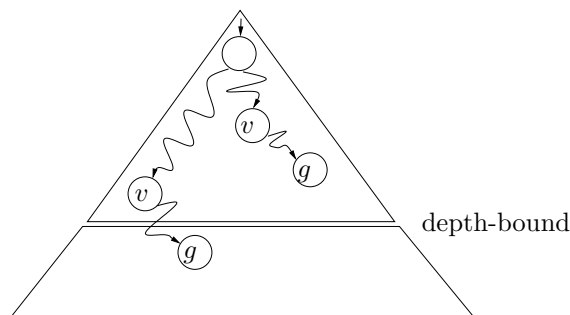


Fig. 2. Anomaly in depth-bounded search.

depth bound. When the search reaches state (v) for the second time along a shallower path (top right copy), it stops exploring its successors, since v has been visited before. As a consequence, g is not found even though it is located at a shallower depth than the search bound.

This anomaly can easily be confirmed experimentally in SPIN when adjusting the search depth manually². We call the above the described algorithm *Iterative-bounding Depth-First Search (IDFS)*.

The second strategy implemented in SPIN is an admissible variant of the first strategy. We call it *Admissible Depth-First Search (ADFS)*. Admissibility is achieved by re-exploring states that have previously been reached. More precisely, states are reopened when they are reached through a path that is shorter than the currently shortest one. This issue, called *reopening*, is also used in A^* . There are three major pitfalls in this strategy.

First, it is necessary to store the current depth of each visited state which adds considerable memory overhead. It is worth saying that some heuristic algorithm like A^* also require to store additional information with each state.

Second, the worst-case time complexity for this strategy is exponential in the size of the state space. As we shall see this also a problem of A^* . But, contrary to ADFS, reopening can be avoided in A^* under certain circumstances.

Third, even when the last remaining error state in the state space has been found, the search must continue, since the algorithm only terminates when no more states can be explored. This drawback, which is common to both options implemented in SPIN, has a very unfavorable effect on the average computational effort required by search.

2.3 Heuristic Search

Uninformed search algorithms like the ones previously described, search explore the state space independent to

² For instance, we have observed this behavior in a telephony system model written in Promela. Up to a search depth bound of 67 no error is found, from bound 68 to 139 an error is found, from bound 140 to 154 no error is found, from 155 onwards an error is found again, and so on.

the actual problem being solved. In contrast, heuristic search algorithms exploit information about the underlying problem in order to guide the search. Basically, heuristic functions establish the desirability of expanding a state. An important class of such functions are estimates, which approximate the distance from a given state to a set of goal states. The most frequently used heuristic search algorithms are A^* [27], best-first search, and IDA* [35].

*Algorithm A^** , as presented in Figure 3 for the verification of safety properties, treats *Open* as a priority queue in which the priority of a state v is determined by a value f . The f -value for a state v is computed as the sum of *i*) the length $v.g$ of the currently shortest path from the start state to v and *ii*) the estimated distance $h(v)$ from v to a goal state.

As a modification to the general state expanding search algorithm of Figure 1, A^* can move states from *Closed* to *Open* when they are reached along a shorter path. This step is called *reopening* and is necessary to guarantee that the algorithm will find the shortest path to the goal state when non-monotone heuristics are used. Monotone heuristics satisfy the property that for each state u and each successor v of u the difference between $h(u)$ and $h(v)$ is less than or equal to the cost of the transition that goes from u to v . If non-monotone heuristics are applied, the number of reopening can be exponential in the size of the state space. However, even if most of the heuristics that we use cannot be proven to be monotone, our experience has shown, that states are very rarely reopened.

An interesting property of A^* is that if h is a lower bound of the distance to a goal state, then A^* is admissible, which means that it will always return the shortest path to a goal state [45].

Best-first search (BF) can be casted as a variant of A^* that takes only h into account, that is $f(u) = h(u)$ for all states u . This greedy algorithm does not guarantee optimal results but in difference to *hill-climbing* search schemes that commit changes to a successor nodes, by backtracking to the set of visited states the algorithm is complete, i.e. the search cannot terminate in local minima or maxima. Especially for weak heuristics and a high density of goal states, BF turns out to be the algorithm of choice. It quickly establishes a first solution making it well-suited to the *exploratory* phase.

*Iterative-deepening A^** . The requirement for storing all expanded and generated states in lists is a serious disadvantage of A^* . Once all memory is exhausted the algorithm can no longer proceed. As a way out of this dilemma, we suggested the use of the iterative deepening variant of A^* , IDA* for short. IDA* is a refinement of the brute-force depth-first iterative deepening search (DFID) [35] that combines the space efficiency of depth-

```

(1) procedure  $A^*(s)$ 
(2) begin
(3)    $Closed \leftarrow \emptyset$ ;
(4)    $Open \leftarrow \emptyset$ ;
(5)    $s.f \leftarrow h(s)$ ;  $s.g \leftarrow 0$ ;
(6)    $Open.insert(s)$ ;
(7)   while not  $Open.empty()$  do
(8)      $u \leftarrow Open.extractmin()$ ;
(9)      $Closed.insert(u)$ ;
(10)    if  $goal(u)$  then
(11)      return solution;
(12)    for each  $e \in outgoing(u)$  do
(13)       $v \leftarrow to(e)$ ;
(14)       $v.g \leftarrow u.g + cost(e)$ ;  $f' \leftarrow v.g + h(v)$ ;
(15)      if  $v \in Open$  then
(16)        if  $(f' < v.f)$  then
(17)           $v.f \leftarrow f'$ ;
(18)        else if  $v \in Closed$  then
(19)          if  $(f' < v.f)$  then
(20)             $v.f \leftarrow f'$ ;
(21)             $Closed.delete(v)$ ;
(22)             $Open.insert(v)$ ;
(23)        else;
(24)           $v.f \leftarrow f'$ ;
(25)           $Open.insert(v)$ ;

```

Fig. 3. A^* search algorithm.

first search and the admissibility of A^* . The price to pay, however, is a loss of time effience. While DFID performs successive depth-first search iterations with increasing depth bound, 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 optimal solution paths if the estimator is a lower bound. We have used IDA* in combination with bit-state hashing [29] to improve the coverage of the state space exploration of very large state spaces, at the expense of a loss of completeness of the model checking procedure.

2.4 Heuristic Estimator Functions

The key to a well-functioning heuristic search with A^* or IDA* are suitable heuristic estimator functions that, for a given state s , return an approximated path length $h(s)$ to reach a goal state. Such functions can also be applied in BF. Hence, in the following we focus on this kind of functions.

In our setting, goal states are states violating a desired property and path length is measured in terms of state transitions needed to reach the goal state.

Actually, our approach can be generalized to consider path costs instead of path lengths. This is done by assigning cost to transitions. For example, we could assign a cost of 1 to each transition of the system corresponding to a communication operation and a cost of 0 to

the rest. In this case, we aim at finding the counterexample involving the minimal number of communication operations. However, for the sake of simplicity we will concentrate on the uniform cost model, where all transitions are assigned to equal costs. This is equivalent to consider path lengths.

Ideally, we would like to have an admissible well-informed estimates in order to be able to quickly find the optimal error trail with A^* . Unfortunately, finding such a function is not easy.

We have developed a number of property-dependent heuristic estimate that we summarize below³. They are neither admissible nor well-informed in general and are thus more suited for a best-first search exploration with the objective of quickly finding an error, rather than aiming at optimal trails.

Formula-based Heuristic Estimate. Assume that f is a global state formula describing a property to be satisfied by a goal state. We recursively define a function $H_f(s)$ that, for a given global system state s , computes the distance to a goal state, in which f holds. Ground terms in the state formula language include expressions such as $i@s$, which means that the Promela process with id i is in its local control state s , $empty(q)$, which says that the queue q be empty, or a that states that state proposition a be true. We directly assign model-dependent lower bound values to these arguments. For instance, $H_{i@s}(s)$ is computed using a local control state distance matrix that indicates how many steps a process has to pass at least to go from one local control state to another. Obviously, this yields a lower bound for the number of global control state transitions needed to reach the global goal state S' in which process i is in local control state s . It is fairly inexpensive to pre-compute the required local distance matrix. To compute $H_{empty(q)}(s)$ we simply return the current number of elements in the queue q , which provides an admissible heuristic estimate as long as every global state transition can at most remove one element from the queue. $H_a(s)$ is assigned to the value 0, if a holds in s , and to 1 otherwise. This heuristic estimate is admissible as well. For boolean formulae, we compute the heuristic estimates based on the conjuncts or disjuncts present in the formula. As an example, for $H_{g \vee h}$ we use $\min\{H_g(s), H_h(s)\}$ as an admissible estimate. For $H_{g \wedge h}$ we use $\max\{H_g(s), H_h(s)\}$, which is only admissible in case g and h are in fact independent. To deal with logical negation, we compute a value \overline{H}_f for a formula of the form $\neg f$. For the details we refer the reader to [16]. The formula based heuristic estimate can directly be applied to model check invariants and code assertions.

Heuristic Estimates for Deadlock Detection. Estimating the distance to *unknown* error states is more difficult. The formula-based heuristic [17] constructs a function

³ Note that we refer to the modeling language Promela which is the input language to the SPIN model checker.

that characterizes error states. 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 [17] we discuss various alternatives, including formula based approaches and an estimate derived from user-provided characterizations of local control states as deadlock-prone.

The active-processes heuristic for deadlock detection is a very simple function defined as the number of processes currently active (or non-blocked). It is a non-admissible heuristic, and bad-informed in systems involving a low number of processes.

2.5 Experimental Results

We implemented the directed explicit state model checking approach in the HSF-SPIN experimental toolset⁴. It combines the capabilities of the SPIN model checker with the Heuristic Search Framework (HSF) [14]. Both SPIN and HSF-SPIN use the same specification language (Promela) and trail format. We performed an extensive set of experiments using various Promela protocol models. This includes the CORBA GIOP protocol [33] and a basic call processing model called POTS as real-life protocols, as well as a number of toy examples such as the dining philosophers. We summarize our results below:

- For deadlock detection as well as for invariant and assertion violations the directed model checking approach led to substantial improvements in reducing the length of the error trails. In many examples, the length reduction factor was approximately in the range of between 0.5 and 500. As an example, checking an invariance violation in the POTS model led to a reduction in the size of the counterexample from 477 messages down to 12 messages, which greatly facilitated error explanation.
- In some instances, the exploration effort as measured in terms of states visited, states expanded and transitions taken was lower than the effort needed in by SPIN's DFS traversal. However, there also was a number of instances where the DFS traversal explored many less states than the A*-based approach. It seems that the reason for this phenomenon lies in the structure of the model and the properties of the formula to be checked.
- In most instances, the A* based search delivered optimally short counterexamples, even if the heuristic estimates were not admissible.
- The greedy BF search usually improved the counterexample length, but in many instances, returned results were not optimal. In some instances, BF deliv-

ered results close to the optimum with substantially lower exploration effort than A*.

- It became clear that the exploration effort is also dependent on the quality of the heuristic estimate function. A poor range of values provided by the estimate is a symptom of a bad-informed heuristic.
- The analysis of the dining philosophers problem, the A* based directed model checking could analyze problem instances of a size orders of magnitude larger than those analyzable with SPIN's DFS. In fact, the directed model checking approach scaled linearly in the number of philosophers, while exploration effort for the DFS search grew exponentially. As we explained in [16], directed model checking is not subject to the in this case infelicitous exploration strategy that SPIN is using in which the exploration order depends on the lexical ordering of the processes. This entails that the behavior of one process (corresponding to one philosopher) is expanded in the depth before other processes' behaviors are exploited, which is contrary to the strategy of leading the dining philosophers into a deadlock by allowing every philosopher to take exactly one step (acquiring one fork).
- Deadlock detection in the GIOP protocol proved the usefulness of IDA* with bitstate hashing to analyze systems with very large state spaces. This approach can find deadlocks, where A* and IDA* based approaches fail due to the exhaustion of memory resources.

3 Trail Improvement

In this section we discuss options for deriving improved, i.e., shorter error trails from already established ones. This is useful when error states and their corresponding error trails have been obtained during previous verification or simulation runs and when the goal is to improve the length of these given error trails. We first present a method to shorten the error trail showing the violation of a safety property. We discuss two suitable heuristic estimates for this case. Second, we present a way to improve the length of the error trail for a liveness property violation.

3.1 Heuristics for Known Error States

We now turn to the question of suitable heuristics that estimate the distance from a current global system state to a target global system state. Note that the global system state of a concurrent message based system such as it is defined by Promela is determined by the control states of all processes, the local data state of all processes, the state of all global variables, and the state of all communication channels. We assume that, as the result of a previous verification or simulation run, we are in

⁴ Source available from <http://www.informatik.uni-freiburg.de/~lafuente/hsf-spin>

the possession of a complete characterization of a global error state, given by an error trail leading into this state. We present two heuristic estimates that exploit information regarding the given error state in order to direct the search.

Hamming Distance Heuristic. Let $s \in S$ be a global state given in a suitable binary encoding, i.e., as a bit vector $s = (s_1, \dots, s_k)$. Furthermore, let s' be the error state we are searching for. One coarse estimate for the number of transitions necessary to get from s to s' is the number of bit-flips necessary to transform s into s' . The estimate is called the *Hamming distance* $H_d(s, s')$ and defined as

$$H_d(s, s') = \sum_{i=1}^k |s_i - s'_i|.$$

Obviously, $|s_i - s'_i| \in \{0, 1\}$ for all $i \in \{1, \dots, k\}$. Note that the estimate $H_d(s, s')$ runs in time linear to the size of the binary encoding of a state and that it is not admissible, since one state transition in the system may change more than one bit in the state description. Nonetheless, in practical experiments the Hamming distance turns out to provide a useful guidance when used as a goal distance estimate in heuristic search algorithms. In other words, despite its inadmissibility it imposes a valuable ordering of the states that the directed search algorithm exploits during state space exploration.

Finite State Machine (FSM) Distance Heuristic. Another distance metric centers around the local states of component processes. Given a target global state $s' \in S$ the FSM heuristic estimate $H_m(s)$ is defined as the sum of the distances between the local states of \mathcal{P}_i in the current global system state s and the local control states of \mathcal{P}_i in g in the state transition graph of the process, for each local process \mathcal{P}_i , i.e.,

$$H_m(s) = \sum_{i=1}^n D_i(pc_i(s), pc_i(s')),$$

where $pc_i(s)$ denotes the local state of process \mathcal{P}_i in global state s . The shortest paths between states in the local state transition graph of each process \mathcal{P}_i are stored in a matrix D_i . The matrix D_i can be pre-computed in time cubic to the number of the states in the state transition graph of process \mathcal{P}_i . Johnson's algorithm [11] can also be applied, offering an asymptotically better time complexity if the graph is sparse. Anyway, note that local state transition graphs are small in comparison to the overall search space, such that the running time of this algorithm does not constitute a severe burden⁵.

⁵ In fact, experimental results with our model checker HSF-SPIN shows that computing these tables requires some milliseconds running time, while the total running time is usually higher than one second.

Once the matrices D_i are constructed, $H_m(s)$ can be computed in time linear to the number of processes of the system.

In contrast to the Hamming distance, the FSM distance does not take the current queue contents and the values of local and global variables into account. Therefore, while the Hamming distance is capable of directing the search into exactly the same given error state, the FSM distance will guide the search into finding the shortest path into states that are equivalent to the original error state in the following sense: The equivalence is defined through the local control states of all processes in the system. We expect that the search will then be directed into equivalent error states that could potentially be reachable through shortest paths. We contend that in many situations this type of heuristic estimate is nevertheless useful, since many errors only depend on the local control states of all processes and not on the data state of the system. Next we prove that the FSM heuristic is monotone.

Theorem 1. *The FSM heuristic estimate is monotone.*

Proof. We have to show that for each transition $s \rightarrow s'$ of the system, we have $H_m(s) \leq 1 + H_m(s')$. Let s'' be the goal state, and let \mathcal{P}_j be the corresponding process of the transition. We have

$$H_m(s) = \sum_{i=1}^n D_i(pc_i(s), pc_i(s'')),$$

which can be rewritten as

$$D_j(pc_j(s), pc_j(s'')) + \sum_{i=1, i \neq j}^n D_i(pc_i(s), pc_i(s'')).$$

On the other hand, we have

$$H_m(s') = \sum_{i=1}^n D_i(pc_i(s'), pc_i(s'')),$$

which, in turn, can similarly be rewritten as

$$D_j(pc_j(s'), pc_j(s'')) + \sum_{i=1, i \neq j}^n D_i(pc_i(s'), pc_i(s'')).$$

We know that in an asynchronous system, a transition changes only the local state of the process performing the local transition, which is \mathcal{P}_j in this case. As a consequence,

$$\sum_{i=1, i \neq j}^n D_i(pc_i(s), pc_i(s''))$$

is equal to

$$\sum_{i=1, i \neq j}^n D_i(pc_i(s'), pc_i(s'')),$$

since

$$\forall i = 1..n, i \neq j : pc_i(s) = pc_i(s').$$

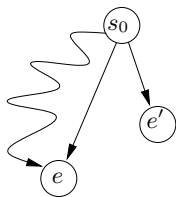


Fig. 4. Two ways of trail improvement: searching for the same error state e or searching for equivalent error states e' .

Moreover, it is easy to see that that $D_j(pc_j(s), pc_j(s'')) \leq 1 + D_j(pc_j(s'), pc_j(s''))$. It follows that $H_m(s) \leq 1 + H_m(s')$. \square

Corollary 1. *The FSM heuristic estimate is admissible.*

Proof. The stated result follows immediately from Theorem 1 since every monotone heuristic is also admissible [46].

3.2 Improving Safety Trails

Trail improvement can be used in two ways. It can be used to reduce the length of an error trail to a given error state, or it can be used to find an improved error trail to some state that is equivalent to a previously found error state with respect to property f . This is illustrated in Figure 4, one can aim at finding a shorter path to state error e , or find a shorter path to a state e' also violating the same property f .

The difference is determined by the nature of the specification, i.e., whether it characterizes precisely one global system state, or whether it characterizes an equivalence class of more than one state violating f . Let us now assume that f is a property specification and that a previous model checking or simulation run has returned e as a state violating f .

Formally speaking, searching for state e entails searching for the violation of a property f_e , stating that *state e is never reached*. Property f_e can be represented by an invariant which negates a formula uniquely characterizing state e . Such a formula ϕ_e requires each variable (including the local control states) to have the same values as in e . Thus, f_e is $\neg\phi_e$ and we can hand this over to a model checker.

Whether we search for e or for any state e' violating the same property f , we can apply A* with the heuristics described in the previous section. We believe that the heuristics can guide the algorithm in order to find error states at shallower depths. For example, the Hamming distance can help if error states are very similar in their binary encodings, while the FSM distance if error states have similar local states. Actually, the only difference during the verification is that if we are looking for a shorter path to e and we find a state e' violating the

same property, we just ignore it and continue the search until e is finally found.

To validate our approach we extended our experimental Promela model checker HSF-SPIN, which inherits some of the capabilities of SPIN and includes various search algorithms like depth-first search, breadth-first search, A* and best-first, as well as many heuristic functions.

All the experiments have been executed on a SUN workstation with a 248 Mhz UltraSPARC-II CPU under Solaris 5.7. If nothing else is stated, the depth bound is set to 10,000 and no state compression technique is used. We let the experiments run for 24 hours with a maximum memory consumption of 512 MB.

Benchmark Models. In the rest of the paper we will use a set of Promela models in our experiments.

The leader election algorithm [13] (**leader**) solves the problem of finding a leader in a ring topology. In the original algorithm each node of the ring has a distinct identifier. The algorithm guarantees that the node with the highest identifier is recognized as leader by every other node. In our *faulty* version every node has the same identifier, leading to a violation of an invariant which states that when a process decides that he is the leader, the number of leaders is exactly one.

We also use a model of a concurrent program that solves the stable marriage problem [42] (**marriers(n)**). The model consists of n concurrent processes (the suitors), which are looking for wife. A process that has found an appropriate pair is idle, until another process steal his pair. If this happens, the earlier looks for a new pair. When each process has a wife, the algorithm terminates. We found an example program trying to solve this problem, which contains a deadlock due to a race condition caused by two actions, which should be performed atomically.

The CORBA GIOP protocol [33] (**giop(n,m)**) is a key component of the OMG's Common Object Request Broker Architecture (CORBA) specification. It specifies a standard protocol that enables interoperability between ORB's from different vendors. The architecture of the model includes a set n ORB clients that communicate with m ORB servers via the GIOP interface. This communication is done via the GIOP and the transport levels. As explained in [33] a deadlock was revealed in the early development of the model. Indeed, the error is similar to a known problem in the TCP protocol and is documented in the GIOP specification [40]. Additionally we have another version of the protocol that violates a response property requiring that whenever a user (ORB Client) process sends a request, a reply will be eventually received.

Model **pots** is a preliminary design of a Plain Old Telephony System (POTS). This model was generated with the visual modeling tool VIP [34]. It is a "first cut" implementation of a simple two-party call processing,

and we know that it is full of faults of various kinds. The model consists of two user processes representing the environment behavior of the switch, as well as two phone handler processes representing the software instances that control the internal operation of the switch according to signals (on-hook, off-hook, etc.) received from the environment. The model violates an invariant requiring that it cannot happen that all user processes and one phone handler process are in *conversation* states, indicating that they presume the two phones to be connected, while the second phone handler is not in a conversation state.

We also have a model of an elevator (**elevator**)⁶, which violates a simple response property: whenever a button is pressed in a level, the elevator will eventually reach the level and open its door.

Last we use a deadlock-free solution to the well-known Dijkstra’s dining philosopher problem (**phil**₀). The problem involves a number of philosophers sitting around the table. There is a plate in front of each philosopher and a fork between each pair of adjacent plates. A philosopher needs two forks to eat the spaghettis contained in its own plate. The problem is to find a protocol to get and release the forks. Our deadlock-free solution is very simple. A philosopher just takes its left and right fork in a single action. This avoids deadlocks but violates a response property stating that whenever a philosopher is eating, his left neighbor will eventually eat.

In the remainder of the paper we use the indicated abbreviations for the models. For scalable protocols we indicate the number of instances using brackets after the name of the protocol. For example, **phil**₀(8) denotes an instance of the dining philosopher’s problem with 8 philosophers.

Experiments In the following set of experiments we assume that we are in the possession of an error trail that is non-optimal in terms of its length, obtained from a verification run with DFS. We will show that a shorter error trail can be found using directed model checking with the A* algorithm. The heuristic estimates are obtained by applying the Hamming and FSM distance metrics to an error state derived from the given error trail. More precisely, the provided error trail is simulated in order to generate the last state, which is the error state that is stored and used as basis for the heuristic functions. The time required for generating the error state is not significant.

Table 1 depicts the results of searching for different safety error states with depth-first search, and improving the error trail provided by that algorithm with A*. Table 2 is similar, but contains results obtained by the two strategies described in Section 2.2, namely IDFS and ADFS. We consider two kind of improvements: searching states violating the same property f or searching for

⁶ Based on the model described in <http://www.inf.ethz.ch/~biere/applets/elsim/>

marriers(4)					
	DFS	f, H_d	f, H_m	f_e, H_d	f_e, H_m
s	407,009	26,545	225,404	126,479	1,754,408
l	121	99	66	121	121
m	58 MB	8 MB	26 MB	22 MB	248 MB
r	367 s	6 s	126 s	62 s	1,150 s
pots					
	DFS	f, H_d	f, H_m	f_e, H_d	f_e, H_m
s	118,099	988	13,865	4,432	14,714
l	987	89	81	89	88
m	58 MB	7 MB	11 MB	7 MB	12 MB
r	81 s	5 s	5 s	5 s	5 s
leader(8)					
	DFS	f, H_d	f, H_m	f_e, H_d	f_e, H_m
s	36	10,733	3,161	10,773	3,173
l	71	71	69	71	71
m	3 MB	10 MB	6 MB	10 MB	6 MB
r	1 s	6 s	1 s	6 s	1 s
giop(2,1)					
	DFS	f, H_d	f, H_m	f_e, H_d	f_e, H_m
s	218	988	30,629	23,518	446,689
l	134	67	65	134	134
m	3 MB	5 MB	22 MB	18 MB	266 MB
r	1 s	1 s	8 s	21 s	128 s

Table 1. Improving various trails corresponding to safety errors with A*.

exactly the same error state e described by formulae f_e which in practice is constructed by simulating the error trail. Estimates applied are the Hamming distance and the FSM distance. The table includes the number of stored states (s), the length of the error trail (l) and time and memory consumption (r and m). If an experiment runs out of time or of memory we write o.t. and o.m., respectively.

In all cases, our heuristic search approach outperforms the blind search strategies in terms of computational effort. This is specially evident in **pots** and **leader**. In the earlier case, ADFS runs out of time because a high number of re-openings are necessary, while in the latter case both IDFS and ADFS run out memory, since the portion of the state space below the last error state is too large to be completely stored.

In some cases, for example **leader** or **giop**, the blind search strategies are able to find shorter counterexamples. This is not the case of **pots**, where IDFS delivers a counterexample that is far from optimal, while A* finds near-to-optimal error trails.

We now concentrate on the results obtained by A*. The first significant result is that finding a shorter path to exactly the same error state f_e is not always possible. Indeed, a shorter path was only found in **pots**.

On the other hand, searching for shorter paths to equivalent errors mostly requires less computational effort. In some cases, like **marriers** the difference between the original error state and the equivalent error states found is only in the data values. More precisely, only the

marriers(4)				
	f, IDFS	f, ADFS	f_e, IDFS	f_e, ADFS
s	1,042,407	1,042,407	2,218,127	2
l	77	66	121	121
m	147 MB	147 MB	311 MB	311 MB
r	677 s	698 s	1,424 s	1430 s

pots				
	f, IDFS	f, ADFS	f_e, IDFS	f_e, ADFS
s	313,677	o.t.	387,604	o.t.
l	428	o.t.	897	o.t.
m	151 MB	o.t.	188 MB	o.t.
r	182 s	o.t.	232 s	o.t.

leader(8)				
	f, IDFS	f, ADFS	f_e, IDFS	f_e, ADFS
s	o.m.	o.m.	o.m.	o.m.
l	68	68	68	68
m	o.m.	o.m.	o.m.	o.m.
r	1073 s	1073 s	1074 s	1074 s

giop(2,1)				
	f, IDFS	f, ADFS	f_e, IDFS	f_e, ADFS
s	58,703	58,703	547,839	548,989
l	58	58	134	134
m	37 MB	37 MB	322 MB	323 MB
r	15 s	15 s	153 s	155 s

Table 2. Improving various trails corresponding to safety errors with various blind-search strategies.

information regarding which wife is associated to which suitor is different. In the original error trail state, all suitors are pretending to marry the same woman, while in the other deadlock states only two of them think they will marry the same person. The race condition is the same, but in the original error, there are more processes involved.

Contrary, in `pots` the states differ also in the local state of the processes. Recall that the invariant violated in this model requires two users and one handler to be in a conversation state, while the other handler is not. In the original counterexample, the handler that is not in conversation are in different states, hence corresponding to two different violations of the same property.

The Hamming distance mostly requires less computational effort than the FSM distance. The only case in which this is not true is in `leader`, where the Hamming Distance, which is not an admissible heuristic, is misleading the search. On the other hand, the FSM distance always provides counterexamples of equal or smaller length, as expected.

3.3 Improving Liveness Trails

Recall that a liveness error trail consists of a path with an initial prefix to a seed state and a cycle starting from that state. Hence, we propose a simple strategy. First, we shorten the path from the initial state to the seed state. Then, we shorten the path from the seed state to itself. Finally, we put the shortened paths together; the

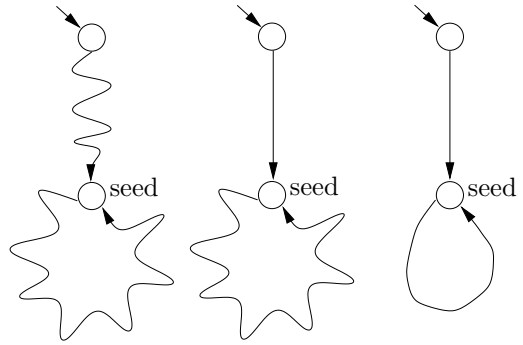


Fig. 5. Liveness error trail shortened in two phases. First the path to the seed is shortened (center), and then the cycle (right).

elevator(3)			
	NDFS	A^*, H_d	A^*, H_m
s	192	58,856	69,162
l	231+90	189+90	189+90
m	3 MB	16 MB	19 MB
r	1 s	30 s	25 s

philo(16)			
	NDFS	A^*, H_d	A^*, H_m
s	2,603	92	742
l	5,223+4	63+4	63+4
m	11 MB	6 MB	6 MB
r	35 s	1 s	1 s

giop(2,1)			
	NDFS	A^*, H_d	A^*, H_m
s	7,331	355,249	931,901
l	288+2	288+2	288+2
m	8 MB	235 MB	609 MB
r	3 s	549 s	430 s

Table 3. Improving the trail of liveness properties in various protocols.

result is a path corresponding to a liveness error trail. Figure 5 illustrates this.

In order to shorten the paths we just apply the technique described in the previous section, i.e we use A^* with the Hamming distance or with the FSM distance.

Experiments. We present experiments on shortening liveness error trails provided by the nested depth-first search algorithm (NDFS). We use A^* with the FSM and Hamming distances, to shorten first the initial prefix and then the cycle. Table 3 depicts the results. Note that the length of a liveness trail is denoted by the sum of the length of the prefix path and the cycle.

In the case of `giop` the computational effort required is significant, but the error trail is not improved even when applying the FSM distance. In the rest of the test cases, we are able to shorten the initial prefix, but not the cycle. Recall that this heuristic is admissible and guarantees to find the shortest path to a given state. Hence, when a path to a state is not improved, that path is already optimal.

Note that in most cases the applying the Hamming distance requires to store less states than applying the FSM distance. On the other hand, the latter case requires less time. The reason is that computing the Hamming distance for a state requires more time than computing the FSM distance, especially if the number of processes in the system is low, while the size of a state representation is large as is the case of `giop`, where applying the Hamming distance requires to explore about three times less states but more time.

4 Partial Order Reduction and Directed Model Checking

Partial order reduction methods exploit the commutativity of concurrent transitions in asynchronous systems in order to reduce the size of the state space. Concurrent transitions can be interleaved in any order. If the order of execution is irrelevant with respect to the property to be analyzed, then a set of concurrent transitions defines an equivalence class of execution paths with respect to that property. The goal of partial order reduction is to reduce the state space of the system by replacing that portion of the state space that corresponds to this equivalence class by just one representative which results in pruning the state space graph. The construction of the reduced state space needs to ensure that the reduced state space is equivalent to the original one with respect to the property to be analyzed. In other words, the construction must ensure that the property to be verified is satisfied in the reduced state space if and only if it is satisfied in the non-reduced original state space. Due to its popularity, we mainly follow the ample set approach in our paper. Nonetheless, most of the reasoning presented in this section can be adjusted to any of the other approaches.

4.1 Ample Set Construction

The algorithm for generating a reduced state space explores only some of the successors of a state. Recall that 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 is denoted as $enabled(s)$. The algorithm selects and follows only a subset of this set called the *ample set* and is denoted as $ample(s)$. A state s is said to be *fully expanded* when $ample(s) = enabled(s)$, otherwise it is said to be *partially expanded*. Along this section let $\mathcal{S} = \langle S, S_0, T, AP, L \rangle$ be a labeled transition system.

Independence. As we argued above, partial order reduction techniques are based on the observation that the order in which some transitions are executed may not be relevant. This leads to the concept of transitions independence. This concept encompasses the property that

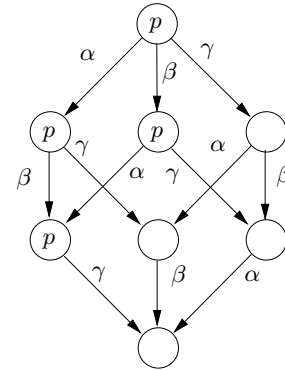


Fig. 6. Illustration of independence and invisibility of transitions.

executing independent transition in one order does not preclude the execution in another order, and that any order of execution leads to the same state. More formally, two transitions $\alpha, \beta \in T$ are independent if for each state $s \in S$ in which both transitions are defined, the following two properties hold:

1. α and β do not disable each other: $\alpha \in enabled(\beta(s))$ and $\beta \in enabled(\alpha(s))$.
2. α and β are *commutative*: $\alpha(\beta(s)) = \beta(\alpha(s))$.

Two transitions are dependent if they are not independent.

Invisibility. A further fundamental concept is the fact that some transitions are *invisible* with respect to the set of atomic propositions that occur in the property specification, which means that they do not alter the truth of any proposition in the set. A transition α is invisible with respect to a set of propositions P if for each pair of states s, s' , if $s' = \alpha(s)$, $L(s) \cap P = L(s') \cap P$. In the following, we will say that a transition is invisible, if it is invisible with respect to the set of propositions that appear in the safety LTL formulae being checked.

Figure 6 illustrates independence and invisibility of transitions. Transitions α, β and γ are pairwise independent. Transitions α and β are invisible with respect to the set of propositions $P = \{p\}$, while γ is not. The figure also illustrates why partial order reduction techniques are said to exploit diamond properties of a system.

Stuttering Equivalence. We now present the concept of *stuttering equivalence* with respect to an LTL formula. Let P be the set of atomic propositions that appear in the formula. A *block* is defined as a finite execution containing invisible transitions only. Intuitively, two executions are stuttering equivalent if they can be defined as a concatenation of blocks such that the atomic propositions of the i -th block of both executions have the same intersection with P , for each $i > 0$. Figure 7 depicts two stuttering equivalent paths with respect to an LTL property in which only propositions p and q occur.

Two transition systems are stuttering equivalent if and only if they have the same set of initial states and

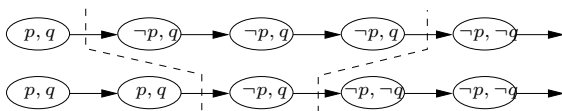


Fig. 7. Stuttering equivalent executions.

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} formulae⁷ cannot distinguish between stuttering equivalent transition systems [9]. In other words, if \mathcal{M} and \mathcal{N} are two stuttering equivalent transition systems, then \mathcal{M} satisfies a given LTL_{-X} specification if and only if \mathcal{N} also does.

Ample Set Construction for LTL_{-X} . The main goal of the ample set construction is to select a subset of the successors of a state such that the reduced state space is stuttering equivalent to the full state space with respect to a property specification, which contains a set P of atomic propositions. The construction should offer a significant reduction without requiring a big computational overhead. The following four conditions are necessary and sufficient for the proper construction of a partial order reduced state space for a given set of atomic propositions P [9].

Condition C0: The set $ample(s)$ is empty exactly when $enabled(s)$ is empty.

Condition C1: Along every path in the full state space that starts at s , a transition that is dependent on a transition in $ample(s)$ does not occur 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 of s must be invisible with respect to P .

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

Conditions **C0**, **C1** and **C2** or their approximations can be implemented independently from the particular search algorithm used. It was shown in [9] that the complexity of checking **C0** and **C2** does not depend on the search algorithm. Checking Condition **C1** is more complicated. In fact, it has been shown to be at least as hard as checking reachability for the full state space. It is, however, usually over-approximated by checking a stronger condition [9] that can be checked independent to the search algorithm. In following lines we shall see that the complexity of checking the cycle condition **C3** depends on the search algorithm used.

⁷ LTL_{-X} is the linear time temporal logic without the next-time operator X .

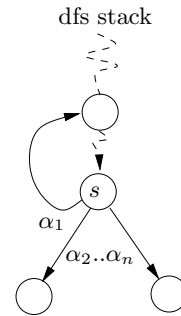


Fig. 8. Reduction example for depth-first search.

Dynamically Checking the Cycle Condition. Condition **C3** is commonly over-approximated using the following condition:

Condition C3_{cycle}: Every cycle in the reduced state space contains at least one state that is fully expanded.

Hence, checking **C3** can be reduced 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 [9]. Consequently, avoiding ample sets containing backward edges except when the state is fully expanded ensures satisfaction of **C3** when using stack-based search algorithms. The resulting stack-based cycle condition **C3_{stack}** can be stated as follows [30]:

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 depicted on Figure 8 illustrates how **C3_{stack}** is used. The set of enabled transitions in state s is $\{\alpha_1, \dots, \alpha_n\}$. 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.

The implementation of **C3_{stack}** for depth-first search strategies marks each expanded state on the stack with an additional flag, so that stack containment can be checked in constant time. Depth-first strategies recording visited states will not consider every cycle in the state space on the search stack, since there might exist exponentially many of them. However, **C3_{stack}** is still a sufficient condition for **C3** since every cycle contains at least a backward edge.

Safety Cycle Condition. Condition **C3⁻** has been implicitly proposed in [30]. It is a relaxation of **C3** that is only correctly applicable to the verification of safety properties:

Condition C3⁻: If for each state of a cycle in the reduced state space, a transition α is enabled, then

α must be in the ample set of some of the successors of some of the states of the cycle.

The authors of [30] propose an approximation of the $\mathbf{C3}^-$ condition defined as follows:

Condition $\mathbf{C3}_{stack}^-$: 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 8. Condition $\mathbf{C3}_{stack}^-$ does not characterize the set $\{\alpha_1\}$ as a valid candidate for the ample set. Contrary to $\mathbf{C3}_{stack}$, condition $\mathbf{C3}_{stack}^-$ 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. Condition $\mathbf{C3}_{stack}^-$ is not sufficient to guarantee $\mathbf{C3}$ which is necessary for checking liveness properties correctly.

Statically Checking the Cycle Condition Before describing this technique, it is necessary to clarify that it does not consist of checking the cycle condition statically. The condition is checked dynamically, but the checking method profits from information gathered statically, i.e. before the verification process starts. In contrast to the previous approaches this 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 [36] present what they call 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 structure of the processes of the system is analyzed before the global state space generation begins. The method is independent from the search algorithm to be used during the verification.

Some transitions are marked with a special flag, called *sticky* and a cycle condition is used, such that no such transition is allowed in an ample set of a state if the state is not fully expanded. Hence, sticky transitions must be selected such that they enforce the cycle condition $\mathbf{C3}$. Marking at least one transition in each local cycle as *sticky* guarantees that at least one state in each global cycle is fully expanded, which satisfies condition $\mathbf{C3}$. This approach can be refined as follows. 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. Assume, for example, that a local cycle C_1 has an overall incrementing effect on 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. The resulting cycle condition $\mathbf{C3}_{static}$ is defined as follows

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

4.2 Ample Set Reduction with Directed Search

When combining partial order reduction with directed search two major problems have to 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, general state expanding algorithms like A* are not well-suited for cycle detection. Stronger cycle conditions or static reduction methods must be applied.

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

Checking the Cycle Condition with a GSEA. Detecting cycles with general state expanding search algorithms that do not perform a depth-first traversal of the state space is more complex. For a cycle to exist, it is necessary to reach an already visited state. If during the search a state is found to be already visited, checking that this state is part of a cycle requires to check if this state is reachable from itself, which increases the time complexity of the algorithm from linear to quadratic in the size of the state space. Therefore, the commonly adopted approach assumes that a cycle exists whenever an already visited state is found. Using this idea leads to weaker reductions, since it is known that state spaces of concurrent systems usually contain a high number of paths leading to a same state, which is precisely one of the facts that partial order reduction is trying to exploit. The resulting condition [1,8] is defined as:

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

We use the example of Figure 9 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 $\mathbf{C3}_{duplicate}$ forbids s' in any ample set if s is not fully expanded. Hence, $\{\alpha_1\}$ and $\{\alpha_1, \alpha_2\}$ are examples of non valid ample sets. On the other hand, the set $\{\alpha_2\}$ is not refuted.

Safety Cycle Condition for a GSEA. The cycle condition $\mathbf{C3}_{stack}^-$ defined above cannot be used with general node expanding algorithms that do not use a search stack, since cycles cannot be efficiently detected. Therefore, we propose an alternative condition based on the

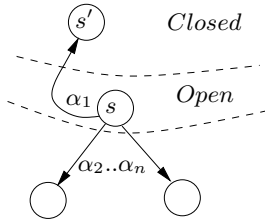


Fig. 9. Reduction example for GSEA.

same idea as $\mathbf{C3}_{duplicate}^-$ in order to enforce the cycle condition $\mathbf{C3}^-$, which is sufficient to guarantee a correct reduction when checking safety properties.

Condition $\mathbf{C3}_{duplicate}^-$: 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 9 illustrates that the set of transitions $\{\alpha_1, \alpha_2\}$ is rejected as ample set by condition $\mathbf{C3}_{duplicate}^-$, but not by $\mathbf{C3}_{duplicate}^-$.

In order to prove the correctness of partial order reduction with condition $\mathbf{C3}_{duplicate}^-$ for general state expanding algorithms we prove a lemma. We use induction on the state expansion ordering, starting from a completed exploration and moving backwards with respect to the traversal algorithm. As a by-product, the more general setting in the lemma also proves the correctness of partial order reduction according to condition $\mathbf{C3}_{duplicate}^-$ for depth-first, breadth-first, best-first, and A* like search schemes. The lemma fixes a state $s \in S$ after termination of the search and ensures that each enabled transition is executed either in the ample set of state s or in a state that appears later on in the expansion process. Therefore, no transition is omitted. Applying the lemma to all states s in S implies $\mathbf{C3}^-$, which, in turn, ensures a correct reduction.

Lemma 1. *For each state $s \in S$ the following is true: when the search of a general search algorithm terminates, each transition $\alpha \in enabled(s)$ has 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 whose 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 been previously neither visited nor expanded. Evidently the expansion order of $\alpha(s)$ is higher

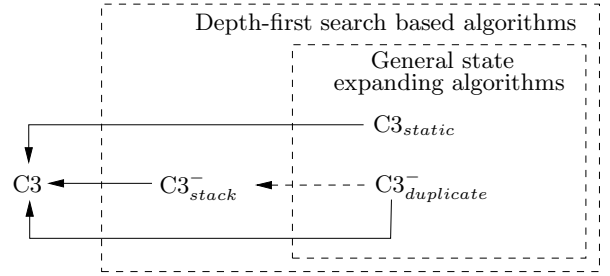


Fig. 10. C3 conditions.

than n . Condition $\mathbf{C1}$ implies that the transitions in $ample(s)$ are all independent from those in $enabled(s) \setminus ample(s)$ [9]. A transition $\gamma \in enabled(s) \setminus ample(s)$ cannot be dependent from a transition in $ample(s)$, since otherwise in the full graph there would be a path starting with γ and a transition depending on some transition in $ample(s)$ would be executed before a transition in $ample(s)$. Hence, transitions in $enabled(s) \setminus ample(s)$ are still enabled in $\alpha(s)$ and contained in $enabled(\alpha(s))$. By the induction hypothesis the lemma holds for $\alpha(s)$ and, therefore, transitions in $enabled(s) \setminus ample(s)$ are selected in $\alpha(s)$ or in a state that is expanded after it. Hence the lemma also holds for s .

Hierarchy of Cycle Conditions. Figure 10 depicts a diagram with all the presented cycle conditions for checking safety properties. Arrows indicate which condition enforces which other. In the following, we will say that a condition A is stronger than a condition B if A enforces B . The dashed arrow from $\mathbf{C3}_{duplicate}^-$ to $\mathbf{C3}_{stack}^-$ denotes that when the search is done with a depth-first search based algorithm, condition $\mathbf{C3}_{stack}^-$ enforces $\mathbf{C3}_{duplicate}^-$, but not vice versa. The dashed regions contain the conditions that can be correctly used with general state expanding algorithms, and those that work only for depth-first search like algorithms. For a given algorithm, the arrows also denote that a condition will produce better or equal reduction.

Solution Quality and Partial Order. One of the goals of directed model checking is to find short paths to errors. Although from a practical point of view near-to optimal solutions may be sufficient to help designers during the debugging phase, finding optimal counterexamples still remains an interesting theoretical question.

Partial order reduction, however, does not preserve optimality of the length of error trails for the full 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 in the full state space. Intuitively, the reason is that the concept of stuttering equivalence does not make assumptions about the length of equivalent blocks.

Suppose that transitions α and β of the state space depicted in Figure 11 are independent and that α is invisible with respect to the set of propositions p . Suppose

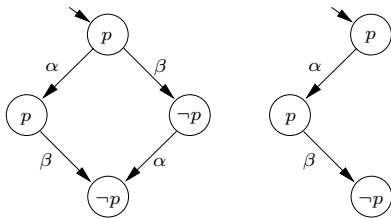


Fig. 11. Example of a full state space (left) and a reduction (right).

further that we want to check the invariant $\mathbf{G}p$, where p is an atomic proposition. 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 consists of transitions α and β , which has a length of 2. In the full one, the initial path with transition β is the shortest path to an error state, such that the corresponding error trail has a length of 1. In Section 5 we will see an approach to mitigate this problem.

4.3 Experiments with Partial Order Reduction

We now present experimental results, intended to highlight the impact of various reduction methods when detecting errors with A^* . More precisely, we want to compare the cycle conditions $\mathbf{C3}_{duplicate}^-$ and $\mathbf{C3}_{static}$ that can be applied jointly with A^* . In addition we include experiments that ignore the cycle condition in order to show how much reduction is lost due to the cycles conditions.

The estimates used for each model are the formula-based heuristic for **pots** and **leader**, the active-process heuristic for **giop** and the Hamming distance for the **marriers** model, where the goal state is obtained by a depth-first search verification run. We have selected two examples, where partial order reduction and heuristic search offer both small and large exploration gains. For instance, heuristic search has a better effect in **pots** and **leader**, while partial order reduction performs better in **giop** and **leader**.

Table 4 shows the effect of applying $\mathbf{C3}_{duplicate}^-$ and $\mathbf{C3}_{static}$ in conjunction with A^* . In addition A^* is used without partial order reduction and with reduction but ignoring the cycle reduction ($\mathbf{C3}_0$).

As expected, both cycle conditions reduce the number of stored states and transitions performed. In all cases except **leader**, condition $\mathbf{C3}_{static}$ offers better reductions than $\mathbf{C3}_{duplicate}^-$. 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

marriers(4)				
	no	$\mathbf{C3}_0$	$\mathbf{C3}_{duplicate}^-$	$\mathbf{C3}_{static}$
s	26,545	10,348	20,049	10,348
l	99	99	99	99
m	8 MB	6 MB	7 MB	6 MB
r	6 s	2 s	6 s	2 s
pots				
	no	$\mathbf{C3}_0$	$\mathbf{C3}_{duplicate}^-$	$\mathbf{C3}_{static}$
s	6,654	5,429	5,574	5,429
l	81	81	81	81
m	7 MB	6 MB	6 MB	6 MB
r	3 s	2 s	2 s	2 s
leader(8)				
	no	$\mathbf{C3}_0$	$\mathbf{C3}_{duplicate}^-$	$\mathbf{C3}_{static}$
s	558,214	104	104	97
l	76	119	119	96
m	272 MB	3 MB	3 MB	3 MB
r	237 s	1 s	1 s	1 s
giop(2,1)				
	no	$\mathbf{C3}_0$	$\mathbf{C3}_{duplicate}^-$	$\mathbf{C3}_{static}$
s	29,915	5,683	5,574	11,981
l	58	58	58	58
m	20 MB	7 MB	13 MB	10 MB
r	11 s	2 s	7 s	4 s

Table 4. Finding a safety violation with A^* and several reduction methods.

will be even better with refined methods for characterizing transitions as sticky.

Note that in some cases the cycle conditions are not refusing any ample set. For example, applying condition $\mathbf{C3}_{static}$ in **marriers** and **pots**, and applying condition $\mathbf{C3}_{duplicate}^-$ in **leader** has the same effect as ignoring the cycle condition. This does not occur, however, in the **giop** model. Moreover, in all cases, ignoring the cycle condition does not avoid finding the error in the model.

Solution quality is only lost in the case of **leader**. This is also the case in which a more drastic reduction is achieved. 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 are the stuttering equivalent executions and, therefore, the longer are the expected trail lengths. Table 4 also shows that the overhead introduced by partial order reduction and heuristic search does not avoid time reduction.

In the next set of experiments we are interested in analyzing the combined state space reduction effect of partial order reduction and heuristic search. More precisely, we have measured the reduction ratio (size of full state space vs. size of reduced state space) provided by one of the techniques when the other technique is enabled or not, as well as the reduction ratio of using both techniques simultaneously.

The same models and estimates of the previous experiments are used here. When partial order reduction is applied the static cycle condition is used. Note that we

marriers(3)		
marriers(3)	N	C
H	1.1	1.2
PO	2.3	2.3
H+PO	5.9	

pots		
	N	C
H	3.7	4.2
PO	1.1	1.2
H+PO	4.5	

leader(8)		
	N	C
H	43.2	1.9
PO	109.1	4.9
H+PO	210.4	

giop(2,1)		
	N	C
H	2.6	2.5
PO	1.4	1.3
H+PO	3.4	

Fig. 12. Table with reduction factor due to partial order and heuristic search

selected the models and heuristics in order to have all four combinations of small/large reduction of heuristic search and partial order.

Figure 12 indicates the reduction factor achieved by partial order and heuristic search when A* is used as the search algorithm. The reduction factor due to a given technique t is computed as the number of stored states when the search is performed without applying t divided by the number of stored states when the search is done applying t . Recall that when no heuristic is applied, A* performs like Dijkstra’s algorithm. 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).

In most cases the reduction factor provided by one of the techniques when working alone ((H,N) and (PO,N)) or combined with the other ((H,C) and (PO,C)) is almost the same. The best situation is **pots**, where the expected gain of applying both independently would be $3.7 \times 1.1 = 4.1$ while the combined effect is a reduction of 4.5 which indicates a synergetic effect.

However, in the case of **leader**, each technique drastically avoids the effect of the other. The combined reduction is 210.4 while the expected gain is $109.1 \times 43.2 = 4,713$. The reason is that partial order reduction performs very good in this example, avoiding the exponential grow of the state space and leading to a linear scaling behavior. Hence, most of the paths that would be discarded by A* are being discarded by the partial order reduction already.

On the other hand, additional experiments show that the relationship between the model being analyzed, the

heuristic strategy being applied, and the combined reduction effect is not clear. For example, applying a more informed heuristic in **marriers** improves the combined effect, which also occurs when applying a less informed heuristic.

5 Optimizing Error Trails using Partial Order Techniques

In this section we discuss a remedy for the problem of solution quality loss due to partial order reduction. We also analyze the joint usage of partial order reduction and the trail improvement method discussed in Section 3.

5.1 Reordering Trails

In practice, solution loss happens when a process can perform an action that will lead to an error state, but the search algorithm delays exploring that action, by considering actions of other process that are actually irrelevant.

We shall see that this problem can be addressed by post-processing the error trail after the verification has finished. The intuition behind this idea is to ignore those transitions in the error trail that are independent from the transition that directly leads to the error state. Independence of these transitions means they are not relevant which justifies to ignore them. In order to obtain a reduced trail representing an actual execution of the system it is also necessary that ignored transitions *cannot enable* transitions that occur later in the original trail. Moreover, removed transitions cannot be visible, since otherwise the resulting execution would not be stuttering equivalent to the original one. In the following we formalize these ideas.

Definition 1. We say that a transition α *can enable* a transition β if and only if there exists a state $s \in S$ such that $\beta \notin \text{enabled}(s)$, but $\alpha \in \text{enabled}(s)$ and $\beta \in \text{enabled}(\alpha(s))$.

Definition 2. A transition α_j of an execution $r = s_0 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} s_n$ is *irrelevant* with respect to the execution if and only if all of the following conditions hold:

- α_j is invisible,
- α_j is independent from each transition α_i , where $j < i \leq n$, and
- α_j cannot enable any transition α_i , where $j < i \leq n$.

A transition that is not irrelevant is called relevant.

The following lemma states that by removing an irrelevant transition from an execution, a stuttering equivalent execution will be preserved.

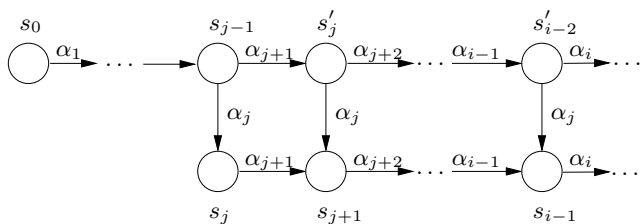


Fig. 13. Illustration of the lemma's proof.

Lemma 2. Let $r = s_0 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} s_n$ be an execution of the system, and α_j a transition that is irrelevant with respect to r . Let further $r' = s_0 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{j-1}} s_{j-1} \xrightarrow{\alpha_{j+1}} s'_j \xrightarrow{\alpha_{j+2}} \dots \xrightarrow{\alpha_n} s'_{n-1}$ be an execution which is obtained by eliminating transition α_j from execution r . Then, r' is an execution of the system that is stuttering equivalent to r .

Proof. Suppose that r' is not an execution of the system, then at least one transition of the execution is not enabled. It is easy to see that this transition belongs to the suffix of r' that occur after state s_{j-1} since the prefix $s_0 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{j-1}} s_{j-1}$ is an execution of the system by definition. Let α_i with $j < i < n$ be the first such transition. We have that $\alpha_i \notin \text{enabled}(s'_{i-2})$. Consider state s'_{i-2} . Transition α_j is enabled in that state, since α_j is independent from all α_k with $j < k \leq n$. Due to the commutativity of independent transitions the execution $s_{j-1} \xrightarrow{\alpha_{j+1}} s'_j \xrightarrow{\alpha_{j+2}} \dots \xrightarrow{\alpha_{i-1}} s'_{i-2} \xrightarrow{\alpha_j} s''_{i-1}$ and the execution $s_{j-1} \xrightarrow{\alpha_j} s_j \xrightarrow{\alpha_{j+1}} \dots \xrightarrow{\alpha_{i-1}} s_{i-1}$ end in the same state, i.e. $s''_{i-1} = s_i$. We know that α_i is enabled in s_i , but not in s'_{i-2} . Therefore, α_j can enable α_i which contradicts our assumptions. Moreover, it is easy to see that if the *eliminated* transition is invisible, then the resulting execution is stuttering equivalent to the original one. \square

Our approach to trail improvement successively eliminates all irrelevant transitions from a counterexample. Note that by eliminating an irrelevant transition, previously relevant transitions may become irrelevant, for instance if they are only dependent on the eliminated transition. To perform the elimination efficiently, we must start eliminating irrelevant transitions at the end of the original trail. By Lemma 2, every elimination of an irrelevant transition yields a stuttering equivalent execution.

Optimality. While our trail improvement method may shorten a given error trail, the resulting improved trail may not be optimal in terms of trail length. Figure 14 illustrates this problem using an example of a full state space and a possible reduction. As in the example of Figure 11 an error state is one in which proposition p does not hold. Suppose that the following pairs of transitions are independent: (α_3, α_4) , (α_6, α_7) and (α_6, α_8) , and that only α_6 and α_4 are visible and negate the value of the

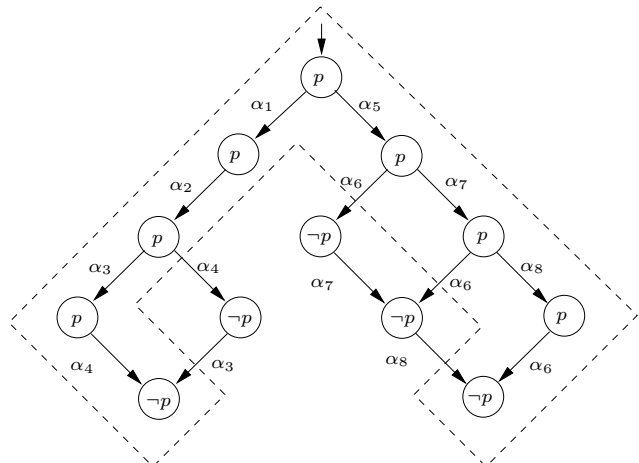


Fig. 14. Another example of a full state space and a reduction (dashed region).

proposition p . Assume that we reduce the state space to only include those states contained in the area delimited by the dashed line. Then, the path formed by transitions $\alpha_1, \alpha_2, \alpha_3$ and α_4 can be established as shortest path in the reduced state space. Applying the above described approach to trail reordering may lead to the error path $\alpha_1\alpha_2\alpha_4$ which is of length 3. This is a possible, since α_3 is invisible. It is independent from α_4 , cannot enable α_4 , and hence can be ignored. On the other hand, the optimal error path in the full state space is of length 2; namely $\alpha_5\alpha_6$.

Experiments. We now report on some experiments to show that by reordering the events of an error trail one can mitigate the loss of solution quality caused by partial order reduction. The only experiment in which we have observed the loss of quality is the assertion violation checking in the election algorithm. Figure 5 depicts the results of the experiments. We apply A* with the FSM distance as heuristic estimate function in the state space search, combined with partial order reduction using $\mathbf{C3}_{duplicate}^-$ and $\mathbf{C3}_{static}$. After the verification we apply the above described trail reordering algorithm to the same error trail. The leader election protocol is invoked with different numbers of processes as indicated by the the leftmost column. The second and fourth columns contain the lengths of the error trails produced by A* guided model checking combined with the respective partial order reduction method. The third and fifth columns denote the length of the trails resulting obtained from reordering, while the rightmost column shows the length of the optimal counterexample in the original state space.

The results show that our method is able to shorten trails obtained by A* with partial order reduction, leading to near-to optimal error trails. The shortened trails are very similar to the original ones. The same pair of processes assume that they are the leaders. The differ-

leader(n)					
n	$C3_{duplicate}^-$	reord.	$C3_{static}$	reord.	opt.
3	49	46	49	46	46
4	63	53	56	53	52
5	77	59	66	59	56
6	91	63	76	63	60
7	105	67	86	67	64
8	119	71	96	71	68

Table 5. The effect of reordering trails for recovering solution quality measured in error trail length.

ences are only due to transitions corresponding to other processes that are not significant for the error.

Although not shown, the time required for the re-ordering algorithm can be neglected.

5.2 Trail Improvement with Partial Order Reduction.

We now discuss the combination of partial order reduction with the trail improvement approach described in Section 3.

Recall that the ample set construction in partial order reduction depends on the visibility of transitions with respect to a property f . A transition is visible with respect to f if it does not change the truth value of the propositions that appear in the specification of f . In the following, we will denote with $po(f)$ the application of partial order reduction with respect to property specification f . Also recall that the full state space and the reduced state space after applying $po(f)$ are semantically equivalent. This means that if there is a state violating f in the full state space then there is also a state violating f in the reduced state space. However, not every state violating f in the full state space is also included in the reduced state space, i.e., the application of $po(f)$ may entail a pruning of states violating f .

Recall that we consider two ways of improving a trail corresponding to the violation of a safety property f : finding a shorter trail that violates the same property and finding a shorter trail to exactly the same error state e described by the original trail. In the first case we search for states violating f , while in the latter we search for states violating the specification f_e , which uniquely characterizes e .

Let us now assume that f is a property specification and that a previous model checking or simulation run has returned S as a state violating f . Applying $po(f)$ to improve the trail leading to S may not yield the desired result since $po(f)$ may decide to prune the sub-tree containing e . On the other hand, $po(f)$ may safely be used if we wish to improve the trail leading to some state e' that also violates f since we know that at least one such state will remain in the reduced state space.

We now focus on alternative solutions to use partial order reduction in improving the error trail leading to

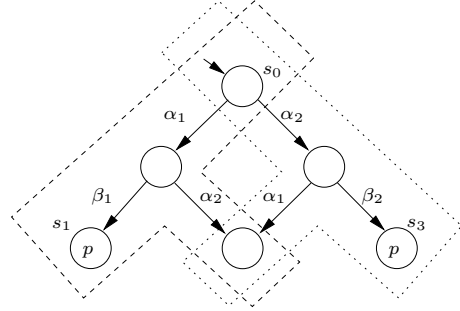


Fig. 15. Different ample sets lead to different state spaces.

the given state e . As explained in Section 3.2 the search for a given state e can be expressed using a propositional formula $f_e = \neg\phi_e$, where ϕ_e characterizes the control state of all processes, the values of all data variables and the contents of all queues in state e . Now, applying $po(f_e)$ guarantees that at least one state violating f_e will be found. There is only one such state, namely e . Unfortunately, this solution has a severe drawback. Since formula the original Since the formula ϕ_e refers to every variable and local state in e , almost every transition in the system is visible. More precisely, only self-transitions that do not change the value of any variable will be invisible with regard to f_e . As a consequence, the ample set rules will mostly force full expansions, resulting in an insignificant state space reduction.

In the following, we will propose a solution to this problem for the case when the provided error state was found using $po(f)$ ⁸ We shall see that requirements on the ample set construction method allow us to apply A^* with $po(f)$ instead of $po(f_e)$ in order to find the shortest path to a state found with depth-first search and $po(f)$.

We first show an example to illustrate that the generated reduced state space depends on the election of the ample set among the different ample set candidates. Consider Figure 15. Transitions α_1 and α_2 are independent and invisible with regard to the atomic proposition p . At state s_0 , both $\{\alpha_1\}$ and $\{\alpha_2\}$ are valid ample sets. Selecting the latter produces the dotted system, while selecting the earlier produces the dashed one. Both systems preserve violation of the invariant $\neg p$. If a first exploration selects $\{\alpha_1\}$ as the ample set of state s_0 , state s_1 is provided as error state. A second exploration that applies reduction with regard to the invariant and that selects $\{\alpha_2\}$ will find a state violating the invariant, but fail to find state s_1 .

In order to show that a A^* applying $po(f)$ with $C3_{duplicate}^-$ is able to find a state provided by depth-first search and $po(f)$ we have to make some assumptions on the strategy used for choosing ample sets among the various candidates.

⁸ Note that our trail improvement approach can also be applied to trails obtained by a verification without po , by a random simulation or manually.

We require that during the state space construction the candidates are considered always in the same order and that the first subset satisfying the ample set conditions is chosen. This assumption is true in typical implementations like the ample set approach implemented in the SPIN model checker or in our experimental model checker HSF-SPIN.

Theorem 2. *It is possible to find a state that violates a specification formula f provided by depth-first search and $po(f)$ by applying a GSEA with $po(f)$ and using $C\mathcal{G}_{duplicate}^-$ as cycle condition.*

Proof. Suppose the contrary, i.e., that there is a state in the system that is visited by a depth-first search and $po(f)$, but not by a GSEA with $po(f)$. Let s be the first state visited by the depth-first search but not with the GSEA. State s must be the result of a transition of the ample set of a state s' during the depth-first exploration. Let $ample_1(s)$ be that ample set. State s' has been visited before s in the depth-first search. Hence according to our assumption, s' is visited by the GSEA. Since we have assumed that the order in which the ample set candidates are considered is always the same, $ample_1(s)$ must have been checked for validity and refused by the ample set rules for GSEA. Since the only difference between that rules and the rules for the depth-first search is the cycle condition, this means that every transition in $ample_1(s)$ leads to an already visited state during the GSEA which contradicts our assumption that state s is not visited by the GSEA. \square

Experiments. In the next set of experiments we try to find the optimal path to a state e previously found by depth-first search verification run with partial order reduction. The path to e is a counterexample for a specification f . We use A* with the FSM distance as heuristic for this purpose and apply partial order reduction. Unfortunately, our model checker inherits SPIN's partial order reduction method which is not exactly the ample set approach, but a weaker method. Hence, we are not able to compare $po(f)$ against $po(f_e)$ for a given specification f and state e violating. However, as we have argued above applying $po(f_e)$ does not reduce the state space in practice, and is thus similar to avoid partial order reduction. Moreover, HSF-SPIN's partial order reduction is weaker than applying $po(f)$. Hence, the experiments we now present can help us to understand what would happen when comparing $po(f)$ against $po(f_e)$.

Table 6 depicts the results. In `giop` the error trail cannot be improved, which means that there is a unique path to the established state in the reduced state space, or several with the same cost. In the case of `pots` we are able to find a shorter path. On the other hand, the results obtained by A* when no partial order reduction is applied show that there is no better path to the provided state in the original state space.

pots			
	DFS+PO	A*+PO	A*
s	118,012	13,166	14,714
l	897	88	88
m	58 MB	11 MB	12 MB
r	84 s	5 s	6 s
giop(2,1)			
	DFS+PO	A*+PO	A*
s	326	163,704	446,689
l	134	134	134
m	3 MB	100 MB	266 MB
r	1 s	39 s	131 s

Table 6. Trail Improvement with Partial Order Reduction.

6 Conclusions and Future Work

We have discussed how to apply partial order reduction and heuristic search in order to improve the bug-finding capabilities of explicit-state model checking.

First, heuristic search can be applied in order to improve the length of an already established error trail. We have considered two ways of improvement: searching for shorter trails that violate the same property and searching for shorter trails to exactly the same error state described by the original trail. We have proposed two heuristics to be applied by A* for this purposes, one of which guarantees to find the shortest path to the given error state. Our experimental results show that our approach is able to efficiently shorten error trails in most cases.

Second, partial order reduction exploits the commutativity of concurrently executed transition in order to explore a reduced behaviorally equivalent state space. There are two issues that have to be considered when combining this reduction technique with heuristic search algorithms. First, most of the approaches to partial order reduction rely on the cycle detection capabilities of the exploration algorithm. When using algorithms like A* one has to apply weaker methods, which lead to weaker reductions. Second, the length of the counterexamples in the reduced state space can be larger than in the original one.

We have proposed a solution for this problem, which reorders the obtained counterexample in order to obtain a counterexample in which the error appears earlier. On the other hand, we have analyzed how one can apply heuristic search and partial order reduction in order to find shorter paths to a given state. At first, in order to guarantee no reduction is possible, we have to guarantee that the given state is present in the reduced state space. However, we show that if the given state was found by applying partial order reduction with respect to a specification property, one can apply A* with the same partial order reduction method to find a shorter path to the given state.

In current work [38] we are analyzing the combination of heuristic search and symmetry reduction to mitigate the state explosion problem of automated verification. The first method reduces the state space to be explored to an equivalent smaller one by exploiting symmetries in the system, while the second techniques guides the search in the direction of errors. Both techniques are, at first, orthogonal and can be combined without drastic changes in the search algorithms. However, finding the shortest path to an already established state requires to apply heuristics which running time can waste the time saved by heuristic techniques applied in symmetry reduction. On the other hand, partial order reduction has been combined with both symmetry [19]. Hence, we plan to perform experiments combining the three techniques.

The gains of heuristic search for model checking rises the question in improved estimates for the search. In [15] parts of SPIN's input language Promela is compiled into an action planning description language to access more involved state-to-goal approximations. The transformation exploits the representation of protocols as communicating finite state machines. Thereby, refined estimates for improved error detection in directed protocol validation are introduced. For example, the *relaxed plan* and *pattern database heuristic*, which come along with an *enforced hill climbing* search engine. Through not yet throughout competitive, the experimental results are encouraging. In simple protocols, planners perform close to state-of-the-art model checkers.

In in-situ program verification, no model of the system is needed or inferred. The verification process is performed by enumerating and simulating all branches to non-deterministic choice points. To the contrary, Bytecode validation as in the Java Pathfinder project (JPF-2) runs different execution traces by simulating a virtual machine for the programming language [28]. It has shown considerable successes in automated bug-finding for Java programs. To improve search efficiency, many enhancements have been implemented. One of the most effective techniques is directed search. Different, so-called structural heuristic aim at coverage metrics based on thread-interleaving and branching statistics. In [43] the authors experimented with JPF-2 and devised heuristics based on low-level state representation the finite state automata representation of the byte-code that is statically available for each class. The heuristic has been derived from the FSM distance heuristic as explained in this paper. Moreover, the work targets trail-directed search, that is, given an error trail for the instance, find a possibly shorter one.

Acknowledgements. The first author is supported by DFG grants Ed 74/2-1 and 74/3-1. The third author is supported by DFG grant Ot64/13-2.

References

1. R. Alur, R. Brayton, T. Henzinger, S. Qaderer, and S. Rajamani. Partial-order reduction in symbolic state space exploration. In *9th Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science 1254, pages 340–351. Springer, 1997.
2. G. Behrmann, A. Fehnker, T. Hune, K.G. Larsen, P. Pettersen, and J. Romijn. Guiding and cost-inality in UPPAAL. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 66–74, 2001.
3. G. Behrmann, A. Fehnker, T. Hune, K.G. Larsen, P. Pettersen, J. Romijn, and F. W. Vaandrager. Efficient guiding towards cost-inality in uppaal. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science 2031. Springer, 2001.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *5th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science 1579. Springer, 1999.
5. R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Design Automation Conference (DAC)*, pages 29–34. ACM/IEEE, 2000.
6. S. Bornot, R. Morin, P. Niebert, and S. Zennou. Black box unfolding with local first search. In *8th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science 2280. Springer, 2002.
7. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric SPIN. In *7th SPIN Workshop on Software Model Checking*, Lecture Notes in Computer Science 1885, pages 1–19. Springer, 2000.
8. C.-T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In *2nd Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science 1055, pages 241–257. Springer, 1996.
9. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
10. J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *23rd International Conference on Software Engineering (ICSE)*, pages 37–46. IEEE Computer Society, 2001.
11. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
12. D. Dams and R. Gerth. The bounded retransmission protocol revisited. In *Infinity, Electronic Notes in Theoretical Computer Science*, volume 9. Elsevier, 1997.
13. D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, (3):245–260, 1982.
14. S. Edelkamp. *Data Structures and Learning Algorithms in State Space Search*. PhD thesis, University of Freiburg, 1999. Infix.
15. S. Edelkamp. Promela planning. In *10th SPIN Workshop on Model Checking Software*, Lecture Notes in Computer Science. Springer, 2003. To appear.

16. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Software Tools for Technology Transfer*, 2003. To appear.
17. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *8th SPIN Workshop on Model Checking Software*, Lecture Notes in Computer Science 2057, pages 57–79. Springer, 2001.
18. E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *5th International Conference on Computer-Aided Verification (CAV)*.
19. E.A. Emerson, S. Jha, and D. Peled. Combining partial order and symmetry reduction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science 1217, pages 19–34. Springer, 1997.
20. P. Godefroid. Using partial orders to improve automatic verification methods. In *2nd Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science 531, pages 176–185. Springer, 1991.
21. P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science 2280, pages 266–280. Springer, 2002.
22. M. G. Gouda. Protocol verification made simple: a tutorial. *Computer Networks and ISDN Systems*, 25(9):969–980, 1993.
23. A. Groce and W. Visser. Heuristic model checking for java programs. In *9th SPIN Workshop on Model Checking Software*, Lecture Notes in Computer Science 2318. Springer, 2002.
24. A. Groce and W. Visser. Model checking java programs using structural heuristics. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, 2002.
25. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *10th International SPIN Workshop on Software Model Checking*, Lecture Notes in Computer Science, page To appear. Springer, 2003.
26. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking Software*, Lecture Notes in Computer Science. Springer, 2003. To appear.
27. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
28. K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000.
29. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.
30. G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *12th International Conference on Protocol Specification, Testing, and Verification (PSTV)*, 1992.
31. C. N. Ip and D. L. Dill. Better verification through symmetry. In *Formal Methods in System Design*, volume 9, pages 41–75, 1996.
32. H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science 2280, pages 445–459. Springer, 2002.
33. 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.
34. M. Kamel and S. Leue. VIP: A visual editor and compiler for v-Promela. In *6th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science 1785, pages 471–486. Springer, 2000.
35. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
36. R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In *4th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science 1384, pages 345–357. Springer, 1998.
37. A. Lluch-Lafuente. *Directed Search in the Verification of Communication Protocols*. PhD thesis, University of Freiburg, Faculty of Applied Sciences, 2002.
38. A. Lluch-Lafuente. Error detection with symmetry reduction and heuristic search, 2002. Submitted.
39. P. Maggi and R. Sisto. Using SPIN to verify security properties of cryptographic protocols. In *9th SPIN Workshop on Software Model Checking*, Lecture Notes in Computer Science 2318, pages 187–204. Springer, 2002.
40. Manual. The common object request broker architecture and specification, 1997. Revised Version 2.1.
41. K. McMillan. Using unfoldings to avoid the state space explosion problem in the verification of asynchronous circuits. In *4th Workshop on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science 531, pages 164–174, 1992.
42. D.G. McVitie and L.B. Wilson. The stable marriage problem. *Communications of the ACM*, 14(7):486–492, 1971.
43. T. Mehler and S. Edelkamp. Trail-directed Java program verification. Technical Report 180, Institute of Computer Science at Freiburg University, 2002.
44. P. Niebert, M. Huhn, S. Zennou, and D. Lugiez. Local first search - a new paradigm for partial order reduction. In *Conference on Concurrency Theory (CONCUR)*, 2001.
45. N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co., Palo Alto, California, 1980.
46. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
47. D. A. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in Systems Design*, 8:39–64, 1996.
48. D. A. Peled. Ten years of partial order reduction. In *10th Conference on Computer-Aided Verification (CAV)*, number 1427 in Lecture Notes in Computer Science 1427, pages 17–28. Springer, 1998.
49. N. Sharygina and D. Peled. A combined testing and verification approach for software reliability. In *Formal Methods of Increasing Software Productivity, International Symposium of Formal Methods Europe*, pages 611 – 628, 2001.

50. S. Rajamani T. Ball, M. Naik. From symptom to cause: Localizing errors in counterexample traces. *To appear in Principles of Programming Languages*, 2003.
51. A. Valmari. A stubborn attack on state explosion. *Lecture Notes in Computer Science*, 531:156–165, 1991.
52. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state space. In *10th Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science 1427, pages 88–97. Springer, 1998.
53. 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.