First publ. in: International journal on software tools for technology transfer 6 (2004), 4, pp. 277-301

Partial-order reduction and trail improvement in directed model checking

Stefan Edelkamp[1], **Stefan Leue**[2], **Alberto Lluch-Lafuente**[3]

[1] Fachbereich Informatik, Universität Dortmund, Baroper Str. 301, GB IV, 44221 Dortmund, Germany
E-mail: stefan.edelkamp@cs.uni-dortmund.de
[2] Fachbereich Informatik und Informationswissenschaft, Universität Konstanz, 78457 Konstanz, Germany
E-mail: Stefan.Leue@uni-konstanz.de
[3] Dipartimento di Informatica, Università di Pisa, Via Buonarroti 2, 56127 Italy
E-mail: lafuente@di.unipi.it

Abstract. In this paper we present work on trail improvement and partial-order reduction in the context of directed explicit-state model checking. Directed explicit-state model checking employs directed heuristic search algorithms such as A* or best-first search to improve the error-detection capabilities of explicit-state model checking. We first present the use of directed explicit-state model checking to improve the length of already established error trails. Second, we show that partial-order reduction, which aims at reducing the size of the state space by exploiting the commutativity of concurrent transitions in asynchronous systems, can coexist well with directed explicit-state model checking. Finally, we illustrate how to mitigate the excessive length of error trails produced by partial-order reduction in explicit-state model checking. In this context we also propose a combination of heuristic search and partial-order reduction to improve the length to already provided counterexamples.

Keywords: Model checking – Heuristic search – Trail improvement – Partial-order reduction – HSF-SPIN

1 Introduction

The success of explicit-state model checking [11] as a software verification technology is largely founded in the automatic nature and the error trail reporting capabilities of the underlying state space exploration algorithms. Broadly 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 of the system that leads from the initial system state into some target state. We call this path an error trail, witness, or counterexample. When checking safety properties, the target state is the state violating the desired property. When checking liveness properties, we search for states that establish a property violating cyclic execution of the system. Explicit-state model checkers usually perform a depth-first search for checking safety properties and a nested depth-first search when checking liveness properties. When a target state has been encountered during the search, the search stack of the depth-first search algorithm contains an execution path from the initial system state into the target state. Similarly, when a property-violating cycle has been encountered, the information contained in the search stacks of the nested depth-first search can be used to reproduce a property-violating execution of the system. Hence, it is very easy in explicit-state model checking to provide the user with an error explanation.

A large number of model checking tools based on explicit-state technology have been built and successfully applied in practice [6]. 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 these types of systems are characterized by a combinatorial state space explosion mainly induced by the data portions of the system model as well as the concurrent composition of processes. Their concurrent nature also causes the execution of these systems to reveal a high degree of nondeterminism. Explicit-state model checkers such as SPIN [35, 36] employ efficient abstraction techniques and data structures to deal with these characteristics. Most notably, symmetry aspects due to concurrent composition have been studied [9, 25, 39], partial-order reductions have been developed to take advantage of the commutativity of independent concurrent transitions [54], and data abstraction techniques have been proposed to reduce very large or even infinite state spaces to finite tractable ones [16, 28].

While in early work on model checking the complete verification of the model was of central interest, in recent applications the focus has been more on using model

Konstanzer Online-Publikations-System (KOPS)
URL: http://www.ub.uni-konstanz.de/kops/volltexte/2008/5766/
URN: http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-57668

checking as a debugging technique for existing requirements, designs, or code artifacts [3, 14, 38]. One of the important problems to be addressed when using model checking in this way is the size of the counterexamples. During debugging, counterexamples are used to understand why errors occur. It is the working hypothesis of this paper that in most error debugging situtations "short is beautiful", i.e., that a shorter error trail is easier to understand than a longer one. We do not rule out the possibility that in some particular situations a longer error trail may be preferable over a shorter one, but many practitioners have confirmed that often our premise is appropriate and therefore claim that our approach applies to a large set of error debugging situations.

In precursory work we addressed this problem by reconciling explicit-state model checking with heuristic, directed search algorithms that had previously been used successfully in solving planning problems [8]. We introduced the concept of *directed explicit-state model checking* in [21]. Traditional model checking algorithms perform an uninformed state space exploration based on depth-first or breadth-first search algorithms. In directed explicit-state model checking we employ heuristic search algorithms such as A* [33] or best-first search [52] 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 in [12], and also reflecting our own experience, model-checking-based debugging can be used during two different phases of the software design process. During a first *exploratory* phase the focus is on quickly finding errors in the system. In this context, quickly means with as little computational effort as possible while ensuring that relatively short error trails are found. The next *fault-finding* phase concentrates on improved error explanation, in particular by finding short counterexamples with less emphasis on computational effort than during the first phase. During this phase we often exploit errors found during the first phase. This two-phase structure has bearing on the use of different heuristic search algorithms in directed explicit-state model checking, as we shall explain later in the paper.

This paper presents a revised and extended version of portions of our previous work on directed model checking, focusing on the joint use of partial-order reduction methods and directed explicit-state model checking [47], and on the improvement of already established counterexamples [22]. The rationale for addressing these questions is as follows.

- The predecessor of this paper [21] focused on the use of logically characterized system properties in directed model checking. For instance, in some of that work logical formulae over state propositions were used to derive heuristic estimates to be used in the state space search. In the current paper, we will investigate a different approach: we will use given error states as the specification of the search target in the directed model exploration. This is useful either to those who have previously "played" with the system and obtained an error state for which they now wish to obtain a shorter trail or to those who have performed nondirected model checking on the state space and wish to reach a target state along a shorter path. In both situations it is not necessary to possess a logical characterization of the state; all that is needed is an error trail that leads into the state in question. We therefore call this approach *trail improvement*. It should be noted that our approach not only permits finding the exact same error state but can also be used to find error states that are equivalent to a partial characterization of an error state. Trail improvement is primarily applicable during the fault-finding phase. While we limit ourselves to the consideration of safety properties in most of this paper, when discussing trail improvement we also consider liveness properties.
- Directed model checking can only hope to be practically applicable if it can be shown to coexist well with other state space reduction techniques. Partial-order reduction techniques are known to be extremely efficient in reducing the state space of concurrent system models to an analyzable size. The state space reduction can be an order of magnitude or more [11], and practitioners know that enabling partial-order reduction often means the difference between infeasibility and feasibility of model checking for models of practical concurrent systems. This reduction technique takes advantage of the commutativity of concurrent transitions if those are independent of each other. Consider a pair of concurrent transitions α and β that are both enabled in a given system state, that are independent of each other, which means that one does not disable the other, and that do not change the state of the system with respect to the property that we are analyzing. Since the transitions are concurrent, the system's global state space is likely to contain both the sequence $\langle \alpha, \ldots, \beta \rangle$ and the sequence $\langle \beta, \ldots, \alpha \rangle$. Given that the above conditions hold, we may reduce the global state space so that it contains only one of the two sequences, thereby eliminating all transitions and states along the path that is abstracted. Since this entails a pruning of a portion of the state space that may contain optimally short error trails, partial-order reduction will likely not preserve the optimality of a directed model checking counterexample obtained through A* and an admissible heuristic estimate. We will therefore carefully have to analyze the impact on the error path quality of partial-order reduction when applied to directed model checking. Another twist stems from the fact that existing applications of partial-order reduction in explicit-state

model checkers rely on the existence of a search stack in the search algorithm. Since this stack is not available in A*, we will have to define overapproximations of the reduction criteria originally defined for partial-order reduction.
– Finally, we are interested in reconciling the ideas of partial-order reduction and trail improvement. First, we propose partial-order-like reordering techniques for given error trails that offer a remedy to the loss of solution quality in the error trails obtained when jointly using partial-order reduction and directed model checking. Second, we investigate the use of partial-order techniques when doing trail improvement.

As we argued above, the premise of our work is that "short is beautiful", i.e., that obtaining shorter counterexamples offers improvements with respect to error explanation and interpretation. In the sections in which we present experimental results, we will mainly focus on describing the quantitative improvements that directed model checking offers and only in some instances discuss how different error trails differ qualitatively. A detailed qualitative analysis requires a deep understanding of both the models and the error trails involved. Providing all the details necessary to obtain a sufficiently deep understanding of these aspects would exceed the scope of this paper.

Related work. In a preliminary paper [21], we provided an extended discussion of related work on directed model checking. This includes initial work on guided state space search [58], the use of guided state space exploration in real-time model checking using the tool UPPAAL [4, 5], guided search in symbolic CTL model checking [7], the use of heuristic search in data flow analysis [12], and directed state space exploration in Java code verification [29, 30]. Also relevant is work on alternative heuristic state space search schemes such as genetic algorithms [27]. While for some of the abovementioned approaches [4, 5, 12] the finding of meaningful counterexamples using guided search techniques is of central interest, other recent papers [2, 31, 40, 55] concentrate on the analysis of counterexamples rather than on improving them. We are not aware of other work on improving given error trails. Several approaches to partial-order reduction techniques have been proposed, namely, those based on "stubborn" sets [56], "persistent" sets [26], and "ample" sets [53]. Although they differ in detail, they are based on similar ideas. For an extended survey of partial-order reduction methods, we refer the reader to [54]. In our paper we will focus on the ample set approach. Since many directed search algorithms do not possess a search stack, we will need to pay special attention to the cycle condition (sometimes also referred to as *cycle provisio*) when constructing ample sets. The detection of cycles without search stacks has been addressed. The *Two-Phase* algorithm [50] has been successfully used in partial-order reduction to reduce state space sizes for some protocol examples. The disadvantage of this algorithm is that it stores much information in a cache, which entails additional complexity. Recent work by Lerda et al. [45] proposes a two-phase approach to the detection of cycles in breadth-first-search-bounded model checking. To the best of our knowledge, at the time of writing partial-order reduction and directed model checking have not been combined by any other authors.

Structure of paper. In Sect. 2 we present a review of directed explicit-state model checking and its application to protocol verification. In Sect. 3 we discuss our approach to trail improvement. We present a discussion of the joint usage of directed explicit-state model checking and partial-order reduction in Sect. 4. In Sect. 5 we analyze the application of partial-order reduction techniques to the shortening of already established counterexamples. We conclude the paper with a summary of our results and an outline of current and future work in Sect. 6.

2 Directed model checking

In this section we review the key concepts of directed explicit-state model checking as published in [21] and [47]. Our discussion of uninformed and informed search algorithms is based on the presentation in [52].

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 are often represented by temporal constraints on the valid execution sequences of the system. These are typically given in an automaton representation or as temporal logic formulae. In this paper we use linear-time temporal logic (LTL) as defined in [48]. In LTL, the operator $\Box$ represents the temporal modality *globally*, and the operator $\Diamond$ represents the temporal modality *eventually*. 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. We will now present the semantic model that we assume and we will provide an introduction to state space exploration and related verification algorithms.

Models and systems. In the remainder of this paper we assume that the system to be verified consists of the asynchronous composition of n finite-state communicating processes $\mathcal{P}_0, \ldots, \mathcal{P}_n$. The state space of the system is represented by a labeled transition system (LTS). A finite LTS (or system, for short) is a tuple $\langle S, S_0, T, AP, L\rangle$ where S is a finite set of states, S_0 is the set of initial states, T is a finite set of transitions such that each transition $\alpha \in T$ is a partial function $\alpha : S \to S$, AP is a finite

set of atomic propositions, and L is a labeling function $S \to 2^{AP}$. We say that proposition p holds in state S iff $p \in L(S)$. In order to distinguish between states and transitions in the system and in each of its processes, we call the former *global* system states and transitions and the latter *local* process states and transitions. Each process is represented by an LTS as well as a set of variables V, a domain D^n with $n = \mid V \mid$, and a partial assignment function $\alpha : V \to D^n$. The state of a process is determined by an assignment of values from D^n to variables in V. The global system is obtained via asynchronous composition of the process LTSs. A transition α is said to be enabled in a state s if $\alpha(s)$ is defined. Each global transition corresponds to at least one local transition of a process. A transition has a *guard* and an *effect*. The guard is a boolean predicate over the variables of the system that determines whether or not the transition is enabled in a global system state. The effect determines the changes in the assignment of values to variables of the system, including the local states of the processes. The execution of a transition system is defined as a sequence of states interleaved by transitions, i.e., a sequence $s_0\alpha_0 s_1 \ldots$, such that s_0 is in S_0 and for each $i \geq 0$, $\alpha_i(s_i)$ is defined and $s_{i+1} = \alpha_i(s_i)$.

Property classes. We focus on the verification of two standard property classes: safety and liveness. 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. The violation of a safety error in an LTS is characterized by a path leading from the initial system state into an error state. Liveness properties, however, are violated not by single states but by infinite cyclic execution paths of the form uv^ω, where u and v are finite sequences of states of the LTS. Liveness errors are therefore characterized by paths uv in the state space of the LTS where a finite prefix u is succeeded by a cyclic trace v.

Automata-based model checking. The automata-based model checking approach to liveness properties [11] comprises the modeling of both the property specification and the system as a Büchi automaton. More precisely, the property specification is negated before being translated into a Büchi automaton. Accepting runs in a Büchi automaton are those that pass infinitely often through at least one accepting state. Checking whether or not the system satisfies the specification consists of checking whether the intersection of the system automaton and the automaton corresponding to the negation of the specification is empty. The intersection between both automata represents the *bad* behaviors of the system. The intersection of both automata is nonempty if and only if there is a cycle in the state space that contains an accepting state of the specification automaton. Such a bad cycle is called an accepting cycle.

Verification algorithms. Checking safety errors can be done by applying simple reachability algorithms like depth-first search or breadth-first search. Checking liveness errors is commonly accomplished using a nested depth-first search algorithm. Those algorithms are *uninformed*, i.e., they do not take information regarding structural properties of the state space into account when determining the state space exploration strategy. On the contrary, in directed model checking we consider a class of algorithms that does exactly this, and we call these algorithms *informed*. Due to the structural information they exploit, these algorithms are, among other things, capable of finding goal states (in our context: error states) along generating paths of minimal length. In keeping with terminology introduced by [52] we say that a state space exploration algorithm is *complete* if it finds an error exactly when there is an error in the system and *admissible* when the error is found using a generating path of minimal length. We sometimes refer to these minimal generating paths as "optimal" or "shortest" as well as calling the error states that are reached along these paths "optimal". For performance reasons the state traversal for current explicit-state model checkers is implemented on the fly. This means that 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 proceeds. This has the advantage that only reachable states will actually be generated. Also, if errors are present, only a portion of the state space needs to be computed, which means that the state space creation terminates when an error state has been found.

A general search algorithm. Figure 1 presents a general state expanding algorithm (GSEA) for the verification of safety properties. The algorithm divides the set of system states S into three mutually disjoint sets: the set *Open* of visited but not yet expanded states, the set *Closed* of visited and expanded states, and the set of nonexpanded states. 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 ex-

```
( 1)  procedure GeneralStateExpandingAlgorithm(s)
( 2)    Closed ← ∅;
( 3)    Open ← ∅;
( 4)    Open.insert(s);
( 5)    while not Open.empty() do
( 6)      u ← Open.extract();
( 7)      Closed.insert(u);
( 8)      if goal(u) then
( 9)        return solution;
(10)      for each e ∈ outgoing(u) do
(11)        v ← to(e);
(12)        if v ∉ Closed and v ∉ Open then
(13)              Open.insert(v);
```

Fig. 1. A general state-expanding search algorithm

panded state is in neither *Open* nor *Closed*, it is added to *Open*. Simple breadth-first and depth-first searches can be defined as instances of the GSEA, where breadth-first search implements *Open* as a queue while depth-first search implements *Open* as a stack.

2.2 Promela and the SPIN model checker

SPIN [36] is an explicit state model checking tool implementing the automata-based model checking approach [11]. Its input language Promela permits the definition of concurrent processes, called *proctypes* in Promela parlance, as well as synchronous or asynchronous communication channels and a limited set of C-like data structures. Concurrency in SPIN is interpreted using an interleaving approach. Properties can be specified in various ways. To express safety properties, the Promela code can be augmented with assertions or deadlock state characterizations. In order to express liveness properties, Promela models can be extended by never claims, a form of Büchi automaton, that express undesired properties of the model. SPIN also provides an automatic LTL to the never claim translator. To check whether a Promela model satisfies an LTL property, SPIN implements the synchronous product construction approach to determine the emptiness of the intersection of the Promela model and the never claim. SPIN uses on-the-fly state space exploration algorithms and implements various optimizations such as, for instance, partial-order reduction. Promela models can be simulated randomly, user guided, or following an error trail. As of this writing no comprehensive operational semantics of Promela has been published. However, the SPIN Web site[1] contains an informal description of the Promela semantics.[2] It is straightfoward to interpret the operational Promela semantics in terms of LTSs. In the remainder of the paper we assume this interpretation.

2.3 Uninformed search algorithms

Model checkers often use uninformed search algorithms like breadth-first search (BFS) or depth-first search (DFS) for checking safety properties. While BFS ensures that counterexamples of minimal depth are found, DFS is more memory efficient. Since memory efficiency is most crucial for explicit-state model checkers in order to tackle the state explosion problem, DFS is the search algorithm of choice in explicit-state model checkers. In addition to DFS, the SPIN model checker implements two 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

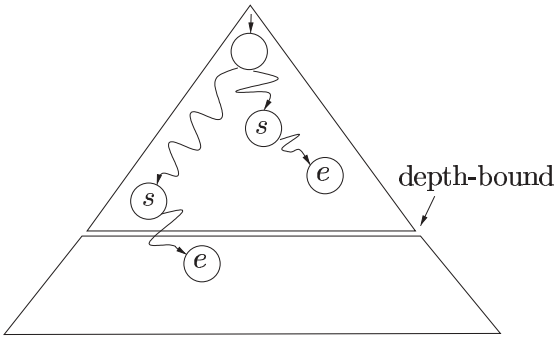

Fig. 2. Anomaly in depth-bounded search

bounds the search depth to the depth of that state. In other words, the depth bound is the depth at which the last error state was found. Initially, there is no depth bound or the depth bound is set to a certain value by the user. Unfortunately, bounding the depth to a value d does not guarantee that every state actually reachable at a search depth less than d will eventually be visited by the algorithm. As a consequence, there is no guarantee of finding the optimal counterexample. This is illustrated in the search tree of Fig. 2. The search visits state s for the first time (lower left copy) and stores it. Goal state e cannot be reached due to the depth bound. When, after backtracking, the search reaches state s for the second time at a shallower search depth (top right copy), it stops exploring its successors, since s has been visited before. As a consequence, e is not found even though it is located at a depth smaller than the depth bound. This anomaly can easily be confirmed experimentally in SPIN when adjusting the search depth manually.[3] We call the algorithm just described *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 reexploring states that have previously been reached by the algorithm. We call such previously reached states *duplicates*. More precisely, states are reexplored when they are reached through a path that is shorter than the currently shortest one on which they were previously reached. This reexploration is sometimes also called *reopening*. This strategy has three major drawbacks.

- First, it is necessary to store the current depth of each visited state, which adds considerable memory overhead.
- Second, the worst-case time complexity for the reopening strategy is exponential in the size of the state space.

[1] `http://spinroot.com`
[2] Cf. `http://spinroot.com/spin/Man/Intro.html`.

[3] For instance, we have observed this behavior in a telephony system model analyzed in SPIN. 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.

– 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 phenomenon, which is common to both options implemented in SPIN, has a very unfavorable effect on the average computational effort required by ADFS.

2.4 Informed heuristic search

Uninformed search algorithms explore the state space without exploiting knowledge about structural properties of the state space or about the property to be verified. On the contrary, heuristic search algorithms exploit information about the underlying problem in order to guide the search. In particular, heuristic functions assess the desirability of expanding a state. An important class of heuristic functions are estimates of the distance from a given state to a set of goal states. Commonly used heuristic search algorithms include A* [33], best-first search (BF), and IDA* [43].

Algorithm A,* as presented in Fig. 3 for the verification of safety properties, is a further variant of GSEA. It 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 of the GSEA in Fig. 1 and in contrast to the depth-bounded DFS algorithm described above, A* can perform a reopening of states. This means that it can move states from *Closed* to *Open* when they are reached along a path that is shorter than any path that they were reached on earlier. It is necessary to reopen some states in order to guarantee that the algorithm will find the shortest path to the goal state when nonmonotone 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[4] of the transition that goes from u to v. Notice that, if nonmonotone heuristics are used, the number of reopenings can be exponential in the size of the state space. However, even if many of the heuristics that we use cannot be proven to be monotone, our experimental experience has shown that, in practical protocol validation examples, 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 [51]. In overlaying terminology, we sometimes say that h is admissible if it is a lower bound estimate.

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

Fig. 3. A* search algorithm

Comparison of A, ADFS, and IDFS* . Both ADFS and A* are optimal search strategies and share some similarities. Both algorithms use hash tables to memorize the set of reached states for fast duplicate detection. In terms of memory consumption, both algorithms use a constant amount of additional information (search tree depth and f-value, respectively) for each state. At least for nonmonotone estimates, both algorithms perform reopening. However, both algorithms also have significant differences. First, A* stops at the first final state it encounters and uses no depth-bounded search iterations, avoiding the additional workload induced by visiting parts of the search tree more than once. Second, by applying heuristic estimates A* takes more information about the search process into account than ADFS. Theoretically, there is not much exploration gain to be expected when using ADFS instead of A*, since every optimal uninformed algorithm at least has to explore all states with a merit smaller than the optimal one. In ADFS this is the set traversed in the final iteration to prove that the error state is of optimal depth; in IDFS this is the state set encountered in the second to last iteration. Optimal exploration algorithms may, of course, behave very differently when considering the number of states they explore that meet the optimal solution value. Due to the guidance that the heuristic estimate provides, the critical state set that at least has to be explored is smaller in A* than in ADFS. In other words, A* will not consider nodes with a very large h-value even if they occur

[4] In our framework, each transition has a cost of 1.

at a small depth. Under certain assumptions, e.g., on the same tie-breaking rules and related monotone estimates, it has been shown that A* is an optimal best-first search strategy with respect to the number of nodes it expands [17]. Moreover, the search strategy in A* is more conservative, so we expect fewer reopenings in the search with admissible but nonmonotone estimates in A* than in the depth-first variant ADFS. The main advantage of ADFS compared to A* is that it may include acceleration techniques such as recursive hashing [13] and incremental state description updates. The main disadvantage of ADFS is the additional work to determine the optimal search threshold. Depending on the sequence of depths for the error states, it will explore many nodes that have a merit larger than the optimal one. A related approach to ADFS with overshooting thresholds uses a doubling strategy and is known as *refined threshold determination* [57].

Best-first search. (BF) can be considered a reopening-free variant of A* that takes only h into account, i.e., $f(u) = h(u)$ for all states u. It can also be viewed as an instance of the GSEA, where *Open* is implemented as a priority queue and states are prioritized solely according to the heuristic value. BF is a greedy algorithm that, like *hill climbing* [52], does not guarantee optimal results. However, contrary to hill climbing, which always follows the most promising successor without backtracking, BF is complete. Especially for weak heuristics and a high density of goal states, according to our experimental results BF appears to be the algorithm of choice. It quickly establishes a first solution, making it well suited to be used during the *exploratory* phase of the software process.

Iterative-deepening A.* The need to store all expanded and generated states in lists seriously limits the applicability of A* when memory is a strictly constrained resource. Once all memory is exhausted, the algorithm can no longer proceed. As a way out of this dilemma we suggested the use of iterative-deepening A*, IDA* for short. IDA* is an informed variant of the brute-force depth-first iterative-deepening search (DFID) [43] that combines the space efficiency of depth-first search and the admissibility of A*. The price to pay, however, is a loss of time efficiency compared to A*. While DFID performs successive depth-first search iterations with increasing depth bounds, in IDA* increasing cost bounds are used to limit search iterations. The initial cost bound is the f value of the initial state. In each of the following iterations, states for which the f value is higher than the current cost bound are not explored. If no more states can be explored, the algorithm enters the next iteration. The minimal f value of all states reached but not yet expanded in the current iteration is used as the cost bound for the next iteration. IDA* simulates A* much like DFID simulates BFS: while A* expands in each iteration the state in the search horizon with minimal value f, IDA* uses in each iteration as cost bound the minimal f value of the states that were reached but not yet expanded in the previous iteration. Like A*, IDA* guarantees optimal solution paths if the heuristic used is admissible. We have used IDA* in combination with bit-state hashing [34] 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.

2.5 Heuristic functions

The key to a well-functioning heuristic search with BF, A*, or IDA* is an informative heuristic estimator function that, for a given state s, returns an estimated path length $h(s)$ of global system state transition steps necessary to reach a goal state s from a given current state. In our setting, goal states are states violating a desired property. The approach presented here can be generalized to consider path costs instead of path lengths. This can be achieved by assigning cost weights 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. We would then aim at finding the counterexample involving the minimal number of communication operations. However, for the sake of simplicity in this paper we will concentrate on a uniform cost model where all transitions are assigned equal costs. Ideally, when using A* we would like to have 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 estimates that we summarize below. 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. However, we found that in the experiments we carried out, even inadmissible heuristics led to optimal or very close to optimal results when applied to A*.

Formula-based heuristic estimate. Assume that f is a global state formula describing a property to be satisfied by an error state. We recursively define a function $H_f(s)$ that, for a given global system state s, computes the distance to a state in which f holds. Ground terms in the state formula language include expressions such as $i@s_i$, which means that the Promela process with ID i is in its local control state s_i, $empty(q)$, which requires that the communication channel[5] q be empty, or a, which states that state proposition a is true. We directly assign model-dependent lower bound values to these arguments. For instance, $H_{i@s_i}(s)$ is computed using a local control state distance matrix that indicates how many computation steps a process has to take at least in order to

[5] In Promela a communication channel is modeled as a special kind of variable that represents a FIFO message buffer.

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 a global goal state in which process i is in local control state s_i. To compute $H_{empty(q)}(s)$, we simply return the current number of elements in channel q, which provides an admissible heuristic estimate as long as every global state transition can at most remove one element from the channel. $H_a(s)$ returns the value 0 if a holds in s, and 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 [21]. The formula-based heuristic estimate can be applied directly to model check invariants and code assertions.

Heuristic estimates for deadlock detection. While feasible, characterizing a deadlock state using a state formula f is nontrivial and possibly leads to very uninformed heuristics. As an alternative, we have proposed a heuristics that is based on a function characterizing the number of active, i.e., nonblocked processes. It is nonadmissible and not particularly informative for models with small numbers of processes, but it worked well on some practical examples. To improve deadlock detection, we have also included heuristic estimates based on user-provided characterizations of local control states as deadlock prone.

2.6 Experimental results

We implemented the directed explicit-state model checking approach in the HSF–SPIN experimental tool.[6] It combines the capabilities of the SPIN model checker with the heuristic search framework (HSF) [19]. Both SPIN and HSF–SPIN use the same specification language (Promela) and trail format. In our predecessor paper [21] we performed an extensive set of experiments using various Promela protocol models.

- The leader election algorithm example [18] (`leader(n)`) solves the problem of finding the leader among a number of nodes in a ring topology. In the original algorithm, each of the n nodes of the ring has a distinct identifier. The algorithm guarantees that the node bidding for leadership with the highest identifier is recognized as leader by every other node. In our *faulty* version of the leader election algorithm, which we use in the experiments, every node has the same identifier. This causes the violation of an invariant stating that when a process decides that it is the leader, the number of leaders is exactly one.
- We also use the Promela model of a concurrent solution to the stable marriage problem [49] authored by us (`marriers(n)`). The model consists of n concurrent processes (the suitors) that are looking for a partner (a wife). A process $\mathcal{P}_i$ that has found a partner k is idle. If a different process $\mathcal{P}_j$ decides that he is better suited for pairing up with k, then k is assigned to $\mathcal{P}_j$ and $\mathcal{P}_i$ continues the search for a new pair. When each process has a partner, the algorithm terminates. Our model contains a deadlock due to a race condition caused by two actions that should be performed atomically.
- The CORBA GIOP protocol [41] (`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 ORBs from different vendors. The architecture of the model includes a set of n ORB clients that communicate with m ORB servers via the GIOP interface. As explained in [41], 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 [32]. 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 eventually be received.
- Model `pots` is a preliminary design of a plain old telephony system (POTS). This model was generated with the visual modeling tool VIP [42]. It is a "first cut" implementation of a two-party call processing model of which 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 invariance 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 use a model of an imperfect elevator system (`elevator(n)`).[7] It violates a simple response property: whenever an elevator request button is pressed at any given floor, the elevator will eventually reach the floor where it was requested and open its door.
- Finally, we use a potentially deadlock-free solution to Dijkstra's dining philosophers problem (`philo(n)`). It involves a number of philosophers sitting around a table. There is a plate in front of each philoso-

[6] Source available from `http://www.informatik.uni-freiburg.de/~lafuente/hsf-spin`.

[7] Based on the model described in `http://www.inf.ethz.ch/~biere/applets/elsim/`.

pher and a fork between each pair of adjacent plates. A philosopher needs two forks to eat the spaghetti on his own plate. The problem is to find a protocol that allows the philosophers to use the forks in such a manner that they can all eat. A winning strategy consists of every philosopher taking his left and right fork as a single atomic action. This avoids the occurrence of deadlocks but violates a response property stating that whenever a philosopher is eating, his left neighbor will eventually eat. A deadlock, however, occurs if every philosopher decides to pick up his left fork and not to release it before a second fork has been acquired.

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, `philo`(8) denotes an instance of the dining philosopher's problem with eight philosophers.

We now summarize our previously obtained experimental results (cf. [47]).

- For deadlock detection as well as for invariance 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 0.5 to 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 required to perform SPIN's depth-first traversal. However, there also was a number of instances where the depth-first traversal explored many fewer states than the A*-based approach. The reason for this phenomenon lies in the structure of the model and the properties of the formula to be checked. It became clear that the exploration effort is dependent on the quality of the heuristic estimate function. A poor range of values provided by the estimate is a symptom of a poorly informed heuristic.
- In most instances, the A*-based search delivered optimally short counterexamples, even if the heuristic estimates were not admissible.
- The greedy best-first search usually improved the counterexample length, but in many instances returned results were not optimal. In some instances, best-first delivered results close to the optimum with substantially lower exploration effort than A*.
- In the deadlock detection in 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 depth-first search. In fact, the size of the state space explored by A* scaled linearly in the number of philosophers, while with depth-first search it grew exponentially. As we explained in [21], directed model checking is not subject to the exploration strategy, in this case infelicitous, 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. This is contrary to the strategy of leading the dining philosophers into a deadlock by allowing every philosopher to take exactly one step, namely, acquiring a fork.
- Deadlock detection in the GIOP protocol proved the usefulness of IDA* jointly 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 a method to obtain improved error trails in a situation in which the error state is given, not by a logic formula, but by an existing error trail. Very much like in the remainder of this paper, we follow the conjecture that in this setting "short is beautiful", i.e., we aim at obtaining shorter error trails. This approach is practically useful when error states and their corresponding error trails have been obtained during previous verification or simulation runs. We consider both the improvement of trails illustrating the violation of safety and liveness properties. We first present the heuristic estimates that we will use throughout the section.

3.1 Heuristics for known error states

We now present suitable heuristics that estimate the distance from a current global system state to a target global system state if that target state is given. 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 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 a 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 $s = (s_1, \ldots, s_k)$. Furthermore, let e be the error state we are searching for. One coarse estimate for the number of transitions necessary to get from s to e is the number of

bit flips necessary to transform s into e. The estimate is called the *Hamming distance* $H^e_d(s)$ and is defined as

$$H^e_d(s) = \sum_{i=1}^{k} |s_i - e_i|.$$

Obviously, $|s_i - e_i| \in \{0, 1\}$ for all $i \in \{1, \ldots, k\}$. The computation of the estimate $H^e_d(s)$ can be performed in time linear in the size of the binary encoding of a state. The obtained estimate is not admissible since one state transition in the system may change more than one bit in the state description. Nonetheless, as we shall see, 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 $e \in S$ the FSM heuristic estimate $H^e_m(s)$ is defined as the sum of the distances between the local states of each $\mathcal{P}_i$ in system state s and in system state e, i.e.,

$$H^e_m(s) = \sum_{i=1}^{n} D_i(pc_i(s), pc_i(e)),$$

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 precomputed in time cubic in the number of the states in the state transition graph of process $\mathcal{P}_i$. Johnson's algorithm [15] can also be applied, offering an asymptotically better time complexity if the graph is sparse. The computation of the distance matrix does not impose a severe burden on the overall computation since local state transition graphs are small in comparison to the global state space.[8] Once the matrices D_i are constructed, $H^e_m(s)$ can be computed in time linear in 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: two global states are equivalent if the local states of the processes in both states are the same. 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 some situations this type of heuristic estimate is useful since some errors depend only 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 $s \to s'$ of the system: $H^e_m(s) \leq 1 + H^e_m(s')$. Let e be the goal state and let $\mathcal{P}_j$ be the corresponding process of the transition. We have

$$H^e_m(s) = \sum_{i=1}^{n} D_i(pc_i(s), pc_i(e)),$$

which can be rewritten as

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

On the other hand, we have

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

which in turn can similarly be rewritten as

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

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,

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

is equal to

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

since

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

Moreover, it is easy to see that $D_j(pc_j(s), pc_j(e)) \leq 1 + D_j(pc_j(s'), pc_j(e))$. It follows that $H^e_m(s) \leq 1 + H^e_m(s')$. □

The proof of Theorem 1 does not generally hold for synchronous communication in which one global transition implies more than one local transition. Promela possesses both synchronous and asynchronous communication primitives. In case of synchronous communication, the sum of the local distances between s and s' may drop by more than the path distance 1 between s and s', which means that the heuristic estimate is not monotone.

[8] In fact, experimental results with our model checker HSF–SPIN show that computing these tables requires some milliseconds of running time, while the total execution time of the directed model checking run is usually higher than 1 s.

There are two possible solutions. First, if one modifies the estimate so that for each global transition at most one local transition is counted when determining the overall value, then the resulting heuristics would remain monotone. The second solution, which has also been adopted in HSF–SPIN, is to measure the length of a trail as the number of steps such that a system transition involving two local transitions in different processes is counted as two steps.

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

Proof. The stated result follows immediately from Theorem 1 since every monotone heuristic is also admissible [52] if the application of the heuristic applied to a goal state yields the value 0. The FSM heuristic estimate satisfies this condition since

$$H_m^e(e) = \sum_{i=1}^{n} D_i(pc_i(e), pc_i(e)) = 0.$$

3.2 Improving safety trails

Trail improvement can be used in two ways, as illustrated in Fig. 4. It can be used to reduce the length of an error trail to a given error state e that violates property f or it can be used to find a shorter error trail to some state e' that also violates 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 global 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. More formally, 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 that negates a formula uniquely characterizing state e. Such a formula ϕ_e requires each variable (including those representing the local control states) to have the same value as in e. Let f_e be denoted by $\neg\phi_e$. We can now use f_e as the property specification for an A* guided model checking run, as described in the previous section. This will lead to error states at shallower depths. The Hamming distance heuristic helps if error states are very similar in their binary encodings, while the FSM distance directs the search very effectively if error states have similar local states.

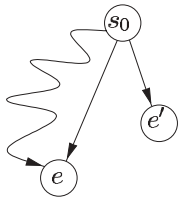

Fig. 4. Two methods of trail improvement: searching for the same error state e or searching for equivalent error states e'

To validate these ideas we use the directed model checker HSF–SPIN, which we described in Sect. 2. All experiments were performed 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 h with a maximum memory consumption of 512 MB.

Experiments. In the following set of experiments we assume that we are in the possession of an error trail obtained through a previous verification run using a DFS state exploration. We will show that a shorter error trail can be found using directed model checking with the A* algorithm in a more efficient way than using blind search algorithms like IDFS and ADFS. The heuristic estimates are obtained by applying the Hamming and FSM distance metrics to the error state derived from the given error trail. More precisely, the provided error trail is simulated in order to generate the last global system state in the trail that is the error state. It is stored and used as the basis for the heuristic estimate functions in a subsequent directed model checking run on the original model. We noticed that the time required for generating the error state from a given error trail is not significant compared to the time required to perform the subsequent directed exploration. Table 1 depicts the results of searching for different safety errors with uninformed DFS and improving the resulting error trail with A*. Table 2 contains results obtained by the uninformed IDFS and ADFS strategies. We consider the two types of error state specifications that we alluded to above: searching equivalent states violating the same property f or searching for exactly the same error state e by using the specification formula f_e. Estimates applied are the Hamming distance (H_d^e) and the FSM distance H_m^e. The table includes the number of stored states (s) and the length of the error trail (l) as well as time (t) and memory consumption (m) for the verification run. When an experiment runs out of time or memory, we write *o.t.* or *o.m.*, respectively.

Result interpretation. In all experiments of Table 1 the heuristic search approach outperforms the blind search strategies in terms of computational effort, where computational effort is measured in terms of the rows s, m, and r. This is especially evident in the models `pots` and `leader`. In Table 2 ADFS runs out of time for `pots` because of the large number of reopenings that are necessary for this model. On the other hand, for `leader` 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. Note that in `leader` the algorithms are able to deliver an error trial before running out of memory. In some instances, for example for the `leader` or `giop` models, the uninformed search strategies are capable of producing shorter counterexamples. This is not the case for the `pots` model, where IDFS delivers a coun-

Table 1. Improving various trails corresponding to safety errors with directed model checking using A*

	DFS	A*			
		f,H^e_d	f,H^e_m	f_e,H^e_d	f_e,H^e_m
marriers(4)					
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	1150 s
pots					
s	118 099	988	13 865	4432	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)					
s	36	10 733	3161	10 773	3173
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)					
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 2. Improving various trails corresponding to safety errors with various uninformed search strategies

	IDFS,f	f,ADFS	IDFS,f_e	ADFS,f_e
marriers(4)				
s	1 042 407	1 042 407	2 218 127	22 128 127
l	77	66	121	121
m	147 MB	147 MB	311 MB	311 MB
r	677 s	698 s	1424 s	1430 s
pots				
	IDFS,f	ADFS,f	IDFS,f_e	ADFS,f_e
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)				
	IDFS,f	ADFS,f	IDFS,f_e	ADFS,f_e
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)				
	IDFS,f	ADFS,f	IDFS,f_e	ADFS,f_e
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

terexample that is far from optimal while A* finds a near-optimal error trail. The nonoptimality of the A* search in these cases can be traced back to the use of nonadmissible heuristic estimates.

We now concentrate on those results obtained by directed model checking using 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 is feasible and in most instances requires less computational effort. In some cases, as in the `marriers` model, the original error state and the equivalent error states that the search finds differ only in the data values. More precisely, only the information regarding which wife is paired up with which suitor is different, but not the control states of the processes. In the error state determined by the original error trail, 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 the scenario leading to the original error state involves more processes.

By contrast, in the `pots` model the states differ also in the local state of the processes. Recall that the invariant violated in this model requires two telephone users and one phone handler process to be in a conversation state, while the other phone handler is not. In the original counterexample, the phone handler that is not in the conversation state is in a different state in the shortened trail. This corresponds to a different violation of the identical property.

The use of the Hamming distance heuristic mostly requires less computational effort than the FSM distance. The only case in which this is not true is in the `leader` example, where the inadmissible Hamming distance heuristic misleads the search. As expected, in all experiments the FSM distance provides counterexamples of equal or smaller length.

3.3 Improving liveness trails

Recall that a liveness error trail consists of a path with an initial prefix leading from the initial system state to a state forming the seed of a cycle and an accepting cycle starting from that seed state. We propose the following strategy to improve liveness error trails, illustrated in Fig. 5. First, we shorten the path from the initial state

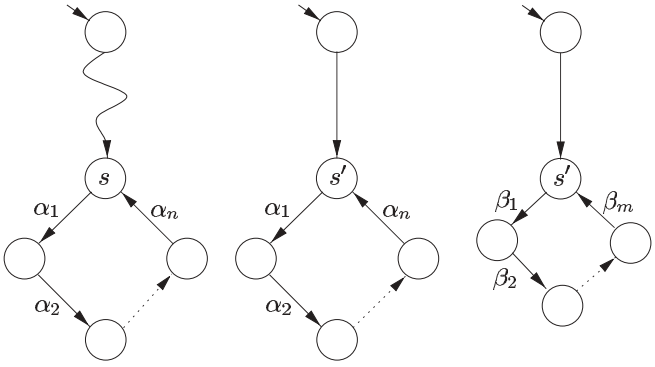

Fig. 5. A liveness error trail (*left*) shortened in two phases: search for a shorter path leading to an equivalent seed state s' (*center*) and search for a shorter cycle through the seed (*right*)

to either the original seed state or to an *equivalent* seed. Then, we shorten the cycle through the seed.

Shortening the path to a seed. The first strategy when looking for a shorter prefix to a cycle seed is to shorten the path from the initial state to the original seed state. This can be done by using A* with heuristics like the Hamming or the FSM distance. Another possible strategy consists of searching for *equivalent seeds*. Let s be the original seed state of some cycle and let $c = \langle \alpha_1, \ldots, \alpha_n \rangle$ be the sequence of transitions of this cycle. We know that executing the sequence from c to s results in state s, i.e., $\alpha_n(\ldots(\alpha_1(s))) = s$. The question that arises is whether there are other reachable states s' in the state space so that executing the sequence c from s' results in an accepting cycle. We call such states s' equivalent seeds. Consider the processes and variables involved in the sequence c. In the automata-based model checking approach during the synchronous product construction, the transitions of a system M are executed in synchrony with the negated property specification automaton N. Each state s of the product can be considered as a pair (s_M, s_N), where s_M is a state of M and s_N is a state of N. Each transition α can then be considered a pair (α_M, α_N), where α_M is a transition in M and α_N is a transition in N. We know that every cycle in the intersection consists of a cycle in both M and N. Let $c_M = \langle \alpha_{M,1}, \ldots, \alpha_{M,n} \rangle$ be the sequence of transitions in M that corresponds to c. Similarly, let $c_N = \langle \alpha_{N,1}, \ldots, \alpha_{N,n} \rangle$ be the sequence of transitions in N that corresponds to c.

We now discuss how to identify the states for which executing the system sequence c_M is possible and results in a cycle. Recall that each transition α_M of the system corresponds to a state change in one process. A transition has a guard determining the enabledness of the transition and an effect determining the changes in the variables. The variables involved in a transition are the variables that appear either in the guard or in the effect. Let us denote these variables by V_{c_M} and the set of processes involved in c_M by $\mathcal{P}_c$. It is obvious that the values of the variables and of the local states of processes not involved in executing transitions from the sequence c_M play no role in determining whether or not c_M is executable in a state and whether its execution results in a cycle. A sufficient but not strictly necessary condition for a state s'_M to be an equivalent seed is that the values of the variables of V_{c_M} and the local states of the processes of $\mathcal{P}_c$ are identical to those in the original seed s_M. This is an overapproximative characterization of the set of all equivalence seeds. Assume that the sequence c_M consists of two transitions of the same process $\mathcal{P}_i$ and that the first transition increments by one a numerical variable x, while the second transition decrements the value of x by one. In this case the value of x obviously plays no role in the cycle, i.e., irrespective of the value of x at a state s'_M the question of whether the sequence of transitions results in a cycle depends only on whether $\mathcal{P}_i$ is in the same local state as in the original seed. However, to make this approximation more precise would require a more thorough analysis of the effect of the sequence c_M, which is beyond the scope of this paper. We now apply the same idea to identify those states for which executing the system sequence c_N is possible and results in a cycle in the property automaton. The transitions of N are all side-effect free. Hence, we only have to require that the local state of N and the value of each variable involved in c_N must be the same as in s. In summary, we consider a state s' to be an equivalent seed with respect to the original seed s and the cycle sequence c if the local states of the processes involved in c, the local state of the never claim, and the values of the variables of the system and the never claim involved in c are the same in both s' and s. We denote this set of states by $[s]$.

Shortening the cycle. Shortening the cycle through a given seed s consists of searching for a shorter path from s to itself. Our approach consists of using A* for this purpose. Once A* finds a shorter path from s to itself, it remains to be verified that the shorter cycle is indeed accepting. We perform this in an efficient manner by adding to each state s' a flag that indicates whether an accepting state was crossed to reach s'. When A* reopens a flagged state while exploring an unflagged state, the flag for the reopened state will be removed. This way we record for each state s' whether the path from the initial state to s' contains an accepting state. If a nonaccepting cycle is found and it is not accepting, the algorithm continues the search.

Heuristic estimates for seeds. To achieve the shortening of a path to a seed, we apply directed model checking using A* and the Hamming or FSM distances as heuristic estimates. In particular, to shorten the initial path to the original seed s we use the Hamming distance H^s_m and the FSM distance H^s_d as heuristic functions. When aiming at equivalent seeds we use alternative heuristics based on a characterization of the equivalence class of s. For example, this can be accomplished by using variants of

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

`elevator`(3)					
	NDFS	A*			
		s, H_d^s	s, H_m^s	$[s], H_d^{[s]}$	$[s], H_m^{[s]}$
s	192	58 856	69 162	60 346	61 979
l	231 + 90	189 + 90	189 + 90	175 + 90	175 + 90
m	3 MB	16 MB	19 MB	16 MB	16 MB
r	1 s	31 s	25 s	27 s	24 s
`philo`(8)					
	NDFS	A*			
		s, H_d^s	s, H_m^s	$[s], H_d^{[s]}$	$[s], H_m^{[s]}$
s	57	18	11	18	11
l	71 + 68	31 + 4	31 + 4	31 + 4	31 + 4
m	3 MB	4 MB	4 MB	4 MB	4 MB
r	1 s	1 s	1 s	1 s	1 s
`giop`(2,1)					
	NDFS	A*			
		s, H_d^s	s, H_m^s	$[s], H_d^{[s]}$	$[s], H_m^{[s]}$
s	7331	355 249	931 901	1730	1929
l	288 + 2	288 + 2	288 + 2	52 + 2	52 + 2
m	8 MB	235 MB	609 MB	6 MB	6 MB
r	3 s	549 s	430 s	1 s	1 s

the Hamming or FSM distance estimates that take only the common part of the states of $[s]$ into account. More precisely, we can base the Hamming distance estimate exclusively on the bits corresponding to the variables and local states that characterize $[s]$. We denote this variant of the Hamming distance by $H_d^{[s]}$. For the FSM distance we use only the local states that characterize $[s]$, i.e., the local states that are equal for every state $[s]$. We denote this heuristic estimate by $H_m^{[s]}$.

Experiments. We present experiments on shortening liveness error trails that were produced by the nested depth-first search algorithm (NDFS). We use A* with the FSM and Hamming distance estimates as described above. Let s denote the search for the same original seed and $[s]$ denote the search for an equivalent seed. Table 3 depicts the results. We define the length of a liveness trail as the sum of the length of the prefix path plus the length of the cycle and use a corresponding sum notation in the rows labeled l.

Result interpretation. In the case of `giop` the computational effort required when aiming at the same seed is significant, but the error trail is not improved. However, if we aim at equivalent seeds, our approach is able to find a significantly shorter error trail, with much less effort. This is possible because the cycle involves only one process. As a consequence there are many equivalent seeds in the state space. In all other examples we are able to shorten the initial prefix. However, the cyclic part of the trace was only shortened in `philo`. Note that in most cases application of the Hamming distance requires the storage of fewer states than does application of the FSM distance. On the other hand, the FSM distance leads to improved running times. The reason is that computing the Hamming distance for a state requires more time than computing the FSM distance, in particular when the number of processes in the system is small while the size of a state representation is large. This phenomenon becomes obvious in the `giop` example, where applying the Hamming distance requires the exploration of about three times fewer states while consuming substantially 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 the portion of the state space that corresponds to this equivalence class by just one representative. This results in pruning portions of 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 nonreduced original state space. Due to its popularity, we mainly follow the ample set approach [11] 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 by $enabled(s)$. The algorithm selects and explores only a subset of this set called the *ample set*, which we denoted by $ample(s)$. A state s is said to be *fully expanded* when $ample(s) = enabled(s)$, otherwise it is said to be *partially expanded*. In the following discussion let $\mathcal{S} = \langle S, S_0, T, AP, L\rangle$ denote 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 transition independence, which encompasses the property that executing independent transitions in one order does not preclude their 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: $\alpha \in \mathit{enabled}(\beta(s))$ and $\beta \in \mathit{enabled}(\alpha(s))$.
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. This is equivalent to saying that they do not alter the truth value 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 and s', if $s' = \alpha(s)$, then $L(s) \cap P = L(s') \cap P$. In the following discussion, 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.

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 ith 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

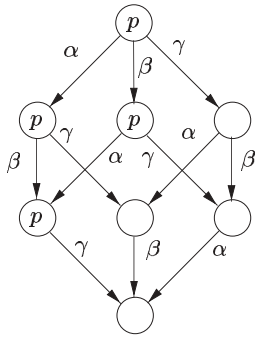

Fig. 6. Illustration of independence and invisibility of transitions

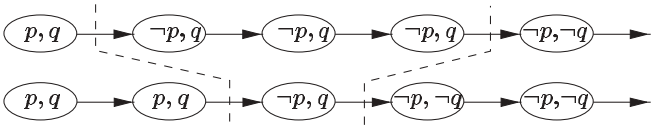

Fig. 7. Stuttering equivalent executions

systems are stuttering equivalent if and only if they have the same set of initial states and for each execution in one of the systems starting from an initial state there exists a stuttering equivalent execution in the other system starting from the same initial state. It can be shown that LTL_{-X} formulae[9] cannot distinguish between stuttering equivalent transition systems [11]. 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 that contains a set P of atomic propositions. The construction should offer a significant reduction without entailing a substantial 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 [11]:

Condition C0: The set $\mathit{ample}(s)$ is empty exactly when $\mathit{enabled}(s)$ 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 $\mathit{ample}(s)$ does not occur without a transition in $\mathit{ample}(s)$ 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 of the particular search algorithm used in the model checking. It was shown in [11] 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 overapproximated by checking a stronger condition [11] that can be checked irrespective of which search algorithm is used in the state space exploration. In addition, this approximation reduces the number of ample set candidates from exponential in $\mathit{enabled}(s)$ to linear in the number of processes

[9] LTL_{-X} is the linear-time temporal logic without the next-time operator X.

since it requires an ample set to be composed of all the enabled transitions belonging to a process. In the following discussion we shall see that the complexity of checking the cycle condition **C3** does indeed depend on which search algorithm is used.

Dynamically checking the cycle condition. Condition **C3** is commonly overapproximated 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. During a DFS in the search space cycles can easily be detected: every cycle contains a *backward edge*, i.e., an edge that links back to a state that is stored on the search stack [11]. 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 [37]:

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 in Fig. 8 illustrates how **C3**$_{stack}$ is used. The set of enabled transitions in state s is $\{\alpha_1, \ldots, \alpha_n\}$. Transition α_1 closes a cycle on the stack and cannot be included in any ample set candidate, except when the state is fully expanded. Therefore, the set of transitions $\{\alpha_2\}$ is a valid candidate, while $\{\alpha_1, \alpha_2\}$ and $\{\alpha_1\}$ are examples of invalid ample sets. 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 one backward edge.

Cycle condition for safety properties. Condition **C3**$^-$ has been implicitly proposed as a relaxation of **C3** in [37]:

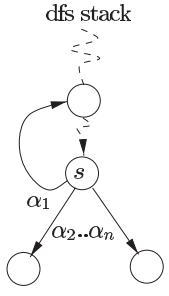

Fig. 8. Reduction example for depth-first search

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.

Note that **C3**$^-$ is only correctly applicable to the verification of safety properties. A disadvantage of this approximation is that it still relies on cycle detection. As a remedy, the authors of [37] propose the following approximation of the **C3**$^-$ condition:

Condition 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 in Fig. 8. Condition **C3**$^-_{stack}$ does not characterize the set $\{\alpha_1\}$ as a valid candidate for the ample set. Contrary to **C3**$_{stack}$, condition **C3**$^-_{stack}$ accepts $\{\alpha_1, \alpha_2\}$ as a valid ample set since at least one transition (α_2) of the set leads to a state that is not on the search stack of the depth-first search. This illustrates that condition **C3**$^-_{stack}$ is not sufficient to fully guarantee **C3**, which is necessary for preserving the correctness of liveness properties during partial-order reduction.

Statically checking the cycle condition. The approximation of the cycle condition **C3** that we introduce now is checked dynamically, but the checking method benefits from information gathered statically, i.e., before the verification process starts. In contrast to the previous approaches this overapproximation method explicitly exploits the structure of the underlying concurrent system. Recall that the global system is constructed as the asynchronous composition of several components. The authors of [44] 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 control flow structure of the individual processes within the system is analyzed before state space generation begins and all local control flow cycles are determined. The method is independent of the search algorithm to be used during the verification. Some transitions are marked with a special flag, called *sticky*. It is then necessary to enforce that no such transition be allowed in an ample set of a state if the state is not fully expanded. Consequently, sticky transitions must be selected such that they enforce the cycle condition **C3**. Marking at least one transition in each local cycle as sticky guarantees that at least one state in each global cycle will be fully expanded, which entails condition **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 overapproximating 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. The first problem lies in the fact that, as we argue above, common partial-order reduction techniques require a condition to be checked, which entails the detection of cycles during the construction of the reduced state space. Depth-first-search-based algorithms can easily detect cycles during the exploration by looking up the contents of the current search stack. On the other hand, general state-expanding algorithms like A* or BF are not well suited for cycle detection since they do not possess a search stack. Alternative cycle conditions or static reduction methods such as the ones we discussed above must be applied. The second problem is that partial-order reduction techniques do not preserve admissibility of the directed state space search. In other words, when partial-order reduction is used in the presence of directed model checking, there is no guarantee that the shortest counterexample that leads to an error will be found. Although we will not always require the shortest counterexample to effectively explain an error, we need to assess the extent of the negative impact that partial-order reduction has on the improvements achievable by 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 have already been visited, checking that this state is part of a cycle requires checking whether 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 the same state, which is precisely one of the features of concurrent system models that partial-order reduction is trying to exploit. The resulting condition [1, 10] 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 Fig. 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 nonvalid ample sets. On the other hand, the set $\{\alpha_2\}$ is not refuted.

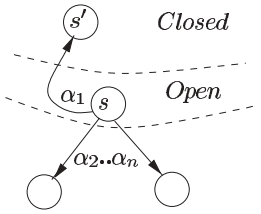

Fig. 9. Reduction example for GSEA

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 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.

Like the comparison of conditions $\mathbf{C3}^{-}_{stack}$ and $\mathbf{C3}_{stack}$, Fig. 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 show the correctness of partial-order reduction with condition $\mathbf{C3}^{-}_{duplicate}$ for general state-expanding algorithms, we prove a lemma stating the completeness of execution of all transitions leading out of any given state once the general state exploration terminates. 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 nth expanded state. If s is fully expanded, the lemma trivially holds for s. Otherwise we have that $\mathit{ample}(s) \subset \mathit{enabled}(s)$. Transitions in $\mathit{ample}(s)$ are selected in s. Since $\mathit{ample}(s)$ is accepted by condition $\mathbf{C3}^{-}_{duplicate}$, there is a transition $\alpha \in \mathit{ample}(s)$ 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 than n. Condition **C1** implies that the transitions in $\mathit{ample}(s)$ are all independent of those in $\mathit{enabled}(s) \setminus \mathit{ample}(s)$ [11]. A transition $\gamma \in \mathit{enabled}(s) \setminus \mathit{ample}(s)$ cannot be dependent on a transition in $\mathit{ample}(s)$, since otherwise in the full graph there would be a path starting with γ and a transition depending on some transition in $\mathit{ample}(s)$ would be executed before a transition in $\mathit{ample}(s)$. Hence, transitions in $\mathit{enabled}(s) \setminus \mathit{ample}(s)$ are still enabled in $\alpha(s)$ and contained in $\mathit{enabled}(\alpha(s))$. By the induction hypothesis the lemma holds for $\alpha(s)$, and therefore transitions in $\mathit{enabled}(s) \setminus \mathit{ample}(s)$ 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 condition. In the following discussion, 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 inner dashed region contains the conditions that can be correctly used with general state-expanding algorithms. The outer dashed region contains the condition that works only for depth-first-search-like algorithms. For a given algorithm, the arrows also denote that a condition at the tail of the arrow will produce better or equal reduction than the condition at the head.

Solution quality and partial order. One of the goals of directed model checking is to find short paths to errors. Partial-order reduction, however, does not preserve the 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 Fig. 11 are independent and that α is invisible with respect to the set of propositions p. Suppose 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 Sect. 4.3 we will provide experimental results illustrating the practical impact that the theoretical loss of solution optimality has. In Sect. 5 we will introduce an approach to mitigating this problem for a given error trail that was generated.

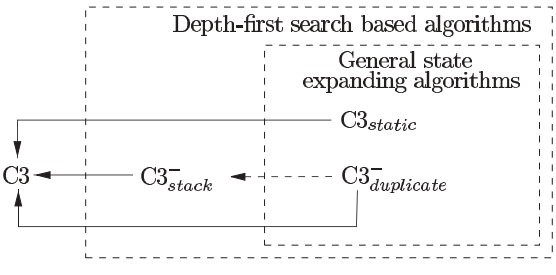

Fig. 10. C3 conditions

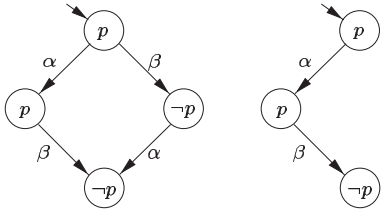

Fig. 11. Example of a full state space (*left*) and a reduction (*right*)

4.3 Experiments with partial-order reduction

We now present experimental results in which we highlight the impact of various reduction methods when detecting safety errors with A*. We also assess the impact of the loss of solution quality on practical verification problems.

The combination of A* and partial-order reduction is implemented by replacing $e \in \mathit{outgoing}(u)$ in A* (cf. Fig. 3) by $e \in \mathit{ample}(\mathit{outgoing}(u))$. In order to assess the potential of using the heuristic distance estimate values during the ample set selection, we perfomed additional experiments not documented in this paper. In these experiments we computed *every* possible valid ample set[10] and selected one according to some *heuristic* strategies. Recall that the common implementation of the ample set selection is to arbitrarily choose the first valid ample set that has been computed. The heuristic strategies we considered included (a) the selection of the ample set that included a state with the best heuristic value, (b) the ample set with the best mean value of the heurisitc esti-

[10] Note that applying the approximation for **C1** suggested in [11] reduces the number of ample sets to $O(|n|)$, where n is the number of processes in the system.

mates for the states in this ample set, and (c) the ample set that included the smallest number of states. We found no significant improvement using any of these heuristics over arbitrary ample set seletion and have therefore not pursued this idea any further.

In the following discussion, we will 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 altogether in order to show how much reduction is lost due to the cycle conditions.

Isolated effect of partial-order reduction. The heuristic estimates that we use in our set of experiments are the formula-based heuristics for `pots` and `leader`, the active-process heuristics for `giop`, and the Hamming distance for the `marriers` model. In this last case, the goal state against which the Hamming distance is measured is derived from a previously executed DFS run. We have selected two examples for which 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` than in the other examples, 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*. Column $\mathbf{C3}_0$ illustrates the effect of ignoring condition **C3** when computing the ample sets. The column labeled "no" contains the experimental values obtained when A* was used without any partial-order reduction. As expected, both $\mathbf{C3}^{-}_{duplicate}$ and $\mathbf{C3}_{static}$ 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 relatively large number of local cycles in the state transition graph of the processes in the `leader` model, while at the same time there is no cycle in the global state space. Since our implementation of the static reduction considers only the simplest approach where one transition in each cycle is marked as sticky, we assume that the results will be even better with refined static reduction methods.

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

`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	6654	5429	5574	5429
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	5683	17 703	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

Ignoring the cycle condition **C3** entails an abstraction of the state space that is not sound, i.e., the resulting state space may no longer contain a property-violating state that was present in the original state space. We provide the figures in column $\mathbf{C3}_0$ only to illustrate the computational impact that condition **C3** has on the computation of the state space reduction. However, it is interesting to observe that in all experiments error states were found, i.e., pruning the state space in this nonvalid way did not eliminate the error states that we were looking for. It is interesting to observe that in some models the cycle condition does not restrict the construction of any ample set. For example, applying condition $\mathbf{C3}_{static}$ in `marriers` and `pots` as well as applying condition $\mathbf{C3}^{-}_{duplicate}$ in `leader` has the same effect on the size of the explored state space as ignoring the cycle condition. Note that a stronger cycle condition, which entails a larger reduction of the state space, can delay the detection of an error since a more effective reduction obviously increases the probability of an optimal solution being pruned from the state space. In other words, larger reductions mean longer stuttering equivalent executions, which entail longer expected error trails. This phenomenon can be observed in `leader(8)`, where ignoring the cycle condition requires the exploration of a few more states than applying the condition $\mathbf{C3}_{static}$. Solution quality is lost only in the case of `leader`, but note that this is also the model for which partial-order reduction is most effective. Table 4 also shows that, in spite of the overhead introduced by partial-order reduction and heuristic search, the running times for the reduced models are still smaller than for the depth-first search exploration.

Effects of joint usage. 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 and when the other is not enabled. We also determine the reduction ratio of using both techniques simultaneously. The same models and estimates

Table 5. Table with reduction factor due to partial-order and heuristic search

`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	

of the previous experiments are used here. When partial-order reduction is applied, the static cycle condition is used. Note that, in order to obtain indicative numbers, we selected the models and heuristics so that we obtain all four combinations of small and large reduction effects due to heuristic search and partial-order reduction. Table 5 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 factors provided by one of the techniques when working alone ((H,N) and (PO,N)) or combined with the other ((H,C) and (PO,C)) are almost identical. A very favorable situation is `pots`, where the expected gain of applying both techniques independently would be $3.7 \times 1.1 = 4.1$ while the combined effect is a reduction of 4.5. Another favorable case is the `marriers` example. This indicates a potentially synergetic effect of the joint application of both techniques. However, in the case of `leader`, each technique significantly reduces the effect of the other. The combined reduction is 210.4, while the expected gain is $109.1 \times 43.2 = 4713$. The reason is that partial-order reduction performs very well in this example. Hence, most of the paths that would be discarded by A* are being discarded by the partial-order reduction already.

In conclusion, we have shown that for the practical examples analyzed here, partial-order reduction and directed model checking using heuristic search can nicely coexist. In some situtations, applying both techniques jointly leads to larger state space reductions than applying each technique in isolation.

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 Sect. 3.

5.1 Reordering trails

We contend that in practical examples the loss of solution quality when jointly using partial-order reduction and heuristic search can be traced to the following phenomenon. One of the concurrent processes in the system has an enabled transition that will lead to an error state, but the search algorithm defers exploring that transition by first considering actions of other processes that are actually irrelevant with respect to the reaching of the error state. This occurs since partial-order reduction ensures that stuttering equivalent sequences of transitions will be explored without ensuring that these sequences have a minimal length. We shall see that this problem can be addressed by postprocessing the error trail after the verification has finished. The intuition behind this approach is to ignore those transitions in the error trail that are independent of the transition that directly leads to the error state. Independence of these transitions means they are not relevant, and we propose to ignore those irrelevant transitions. In order to obtain a reduced trail representing an actual execution of the system, it is also necessary that ignored transitions *be incapable of enabling* transitions that occur later in the original trail. Moreover, removed transitions may not be visible since otherwise the resulting execution would not be stuttering equivalent to the original one. In the following discussion 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 $\beta \notin enabled(s)$, but $\alpha \in enabled(s)$ and $\beta \in enabled(\alpha(s))$.*

Definition 2. *A transition α_j of an execution $r = s_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} s_n$ 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 of 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 obtained.

Lemma 2. *Let* $r = s_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} s_n$ *be an execution of the system and* α_j *a transition that is irrelevant with respect to* r*. Let further* $r' = s_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{j-1}} s_{j-1} \xrightarrow{\alpha_{j+1}} s'_j \xrightarrow{\alpha_{j+2}} \ldots \xrightarrow{\alpha_n} s'_{n-1}$ *be an execution obtained by eliminating transition* α_j *from execution* r*. Then,* r' *is an execution of the system that is stuttering equivalent to* r *(cf. Fig. 12).*

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 occurs after state s_{j-1} since the prefix $s_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{j-1}} s_{j-1}$ is an execution of the system by definition. Let α_i with $j < i < n$ be the first such transition. We have that $\alpha_i \notin \mathit{enabled}(s'_{i-2})$. Consider state s'_{i-2}. Transition α_j is enabled in that state since α_j is independent of all α_k with $j < k \leq n$. Due to the commutativity of independent transitions, the execution $s_{j-1} \xrightarrow{\alpha_{j+1}} s'_j \xrightarrow{\alpha_{j+2}} \ldots \xrightarrow{\alpha_{i-1}} s'_{i-2} \xrightarrow{\alpha_j} s''_{i-1}$ and the execution $s_{j-1} \xrightarrow{\alpha_j} s_j \xrightarrow{\alpha_{j+1}} \ldots \xrightarrow{\alpha_{i-1}} s_{i-1}$ 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. □

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 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 13 illustrates this problem using an example of a full state space and a possible reduction. As in the example of Fig. 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 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 α_1, α_2, α_3, and α_4 can be established as the shortest path in the reduced state space. Applying the approach described above to trail reordering may lead to the error path $\alpha_1\alpha_2\alpha_4$, which is of length 3. This is possible since α_3 is invisible, it is independent of α_4, it cannot enable α_4, and hence it can be ignored. On the other hand, the optimal error path in the full state space is of length 2: $\alpha_5\alpha_6$.

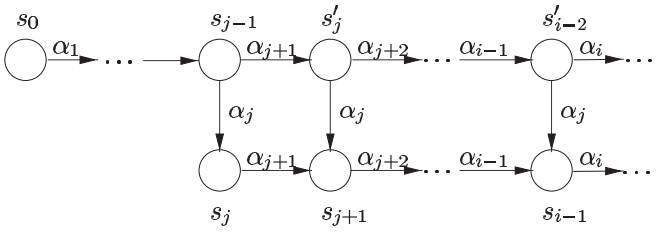

Fig. 12. Illustration of lemma's proof

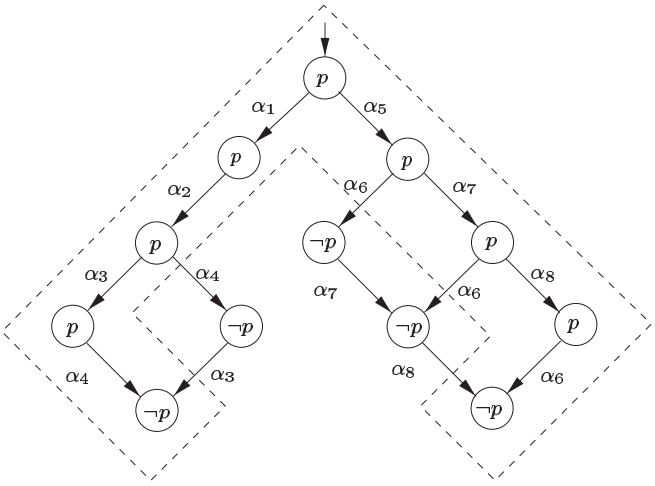

Fig. 13. Another example of a full state space and a reduction (*dashed region*)

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 `leader` example. Table 6 depicts the results of performing reordering on the suboptimal error trails for this example. As algorithms we use DFS and A*, both combined with partial-order reduction. DFS uses condition $\mathbf{C3}^-_{duplicate}$, while A*, which in this case employs the FSM distance as heuristic estimate, uses $\mathbf{C3}^-_{duplicate}$ and $\mathbf{C3}_{static}$. After the verification we apply the trail reordering algorithm described above to the delivered trail. The length of the originally obtained trail (bef.), the shortened one (aft.), and the optimal one (opt.) are shown. The leader election protocol example is used with different numbers of processes as indicated by the numbers in the left column.

The results show that our method is efficient in shortening trails obtained by A* or DFS with partial-order

Table 6. Effect of reordering trails for recovering solution quality measured in error trail length

	`leader`(n)						
n	$\mathbf{C3}^{-}_{stack}$		$\mathbf{C3}^{-}_{duplicate}$		$\mathbf{C3}_{static}$		opt.
	bef.	aft.	bef.	aft.	bef.	aft.	
3	51	49	49	46	49	46	46
4	66	55	63	53	56	53	52
5	80	59	77	59	66	59	56
6	94	63	91	63	76	63	60
7	108	67	105	67	86	67	64
8	122	71	119	71	96	71	68

reduction. The reordering also yields near-optimal error trails. We observed that the reordered trails are qualitatively very similar to the original ones. In the reduced trail the same two processes assume that they are the leader. The only differences between the original and the reduced trail relate to transitions in other processes that are not significant for the error. Although not shown, the running time required for the reordering algorithm can be neglected.

5.2 Trail improvement with partial order reduction

We now discuss the combined use of partial-order reduction and trail improvement as described in Sect. 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 discussion, we will denote by $\mathrm{po}(f)$ 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 $\mathrm{po}(f)$ 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 $\mathrm{po}(f)$ may entail a pruning of states violating f.

As we saw in Sect. 3, trail improvement can be used in two ways. It can either 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 a property f. The difference is determined by the nature of f, i.e., whether f 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 s as a state violating f. Applying $\mathrm{po}(f)$ to improve the trail leading to s may not yield the desired result since $\mathrm{po}(f)$ may decide to prune the subtree containing s. On the other hand, $\mathrm{po}(f)$ may safely be used if we wish to improve the trail leading to some state s' that also violates f since by the soundness of partial-order reduction we know that at least one such state will remain in the reduced state space.

We now focus on alternative solutions to using partial-order reduction in improving the error trail leading to the given state s. The search for a given state s can be expressed using a propositional formula ϕ_s that characterizes the control state of all processes, the values of all data variables, and the contents of all queues in state s. We can now formulate the invariant $\neg\phi_s$ expressing that s will never be reached and hand this over to a model checker. Now, applying $\mathrm{po}(\neg\phi_s)$ guarantees that at least one state violating f_s will be found. There is only one such state, namely, s. Unfortunately, this solution has a severe drawback. Since the formula ϕ_s characterizing a state s refers to every variable and local state in s, 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_s. As a consequence, the ample set rules will enforce an almost complete expansion of the state space.

In the following discussion, we shall propose a solution to this problem for the case where the provided error state was found using $\mathrm{po}(f)$.[11] We shall see that procedural constraints on the ample set construction algorithm allow us to apply A* with $\mathrm{po}(f)$ instead of $\mathrm{po}(f_s)$ in order to find the shortest path to a state found with depth-first search and $\mathrm{po}(f)$. We first show an example to illustrate that the generated reduced state space depends on the selection of the ample set among the different ample set candidates. Consider Fig. 14. 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 subgraph, while selecting the former produces the dashed one. Both systems represented by these subgraphs preserve violation of the invariant $\neg p$. If a first exploration

[11] Note that our trail improvement approach can also be applied to trails obtained by a verification without partial-order reduction, by a random simulation, or by manual inspection or simulation.

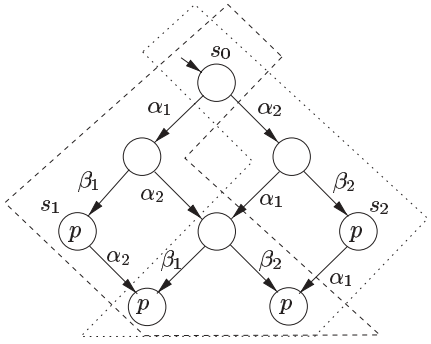

Fig. 14. Different ample sets lead to different state spaces

selects $\{\alpha_1\}$ as the ample set of state s_0, state s_1 is provided as the 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 GSEA applying $\mathrm{po}(f)$ with $\mathbf{C3}^-_{duplicate}$ is capable of finding a state provided by depth-first search and $\mathrm{po}(f)$, we need to make some assumptions regarding the strategy used for choosing ample sets among the various candidates. We require that during the ample set construction candidate transitions always be considered in the same order. We also assume that the first subset satisfying the ample set condition is chosen. The ample set constructions in SPIN and in HSF–SPIN work exactly according to this scheme.

Theorem 2. *It is possible to find a state that violates a specification formula f produced by depth-first search and $po(f)$ by applying a GSEA with $po(f)$ and using $\boldsymbol{C3}^-_{duplicate}$ 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 $\mathrm{po}(f)$ but not by a GSEA with $\mathrm{po}(f)$. 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' was visited before s in the depth-first search. 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 those 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. □

Experiments. In the next set of experiments we aim at finding the optimal path to a state previously found with depth-first search and partial-order reduction. We use A* with the FSM distance as heuristics for this purpose and apply partial-order reduction with the cycle condition $\mathbf{C3}^-_{duplicate}$. Unfortunately, our model checker HSF–SPIN inherits SPIN's partial-order reduction method, which is weaker than the ample set approach. As a consequence, we cannot compare $\mathrm{po}(f)$ against $\mathrm{po}(f_s)$ for a given specification f and state s violating f. However, as we have argued above, applying $\mathrm{po}(f_s)$ does not reduce the state space in practice and is therefore similar to not using partial-order reduction at all. Moreover, HSF–SPIN's partial-order reduction is weaker than applying $\mathrm{po}(f)$. Hence, the experiments presented in Table 7 can help us understand what would happen in a comparion of $\mathrm{po}(f)$ with $\mathrm{po}(f_s)$. In most cases the error trail cannot be improved, which means that there is either a unique path to the established error state in the reduced state space or there are several paths with the same cost. Only in the case of `pots` are we 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.

Table 7. Trail improvement with partial-order reduction

`pots`			
	DFS+PO	A*+PO	A*
s	118 012	19 366	20 999
l	897	88	88
m	58 MB	14 MB	15 MB
r	84 s	9 s	9 s
`giop`(2,1)			
	DFS+PO	A*+PO	A*
s	326	204 416	446 689
l	134	134	134
m	3 MB	124 MB	266 MB
r	1 s	59 s	131 s

6 Conclusions and future work

The first contribution of this paper lies in the introduction of the concept of trail improvement. We have shown how to use directed model checking techniques in order to improve existing error trails. We suggested two heuristic estimates, the Hamming distance and the FSM distance metrics. We then illustrated the use of these techniques to reduce the length of safety error trails. The heuristics were used to search for either exactly the given error state or for a property equivalent state. A main result was that it is easier to improve results if one searches for equivalent states, which leads to better improvements as well as lower computational costs. The inadmissible Hamming distance appears to require less computational effort compared to the admissible FSM distance heuristics.

As the second main contribution of our paper we analyzed the joint usage of partial-order reduction techniques and directed model checking. We gave effective over-approximations of the cycle condition **C3** that permit partial-order reduction to work well for search algorithms that do not employ a stack, such as A*. We also showed that the potential loss in solution quality is not problematic for the practical examples that we have experimented with.

Finally, we reconciled trail improvement with partial-order reduction. We suggested a reordering technique that improves the length of suboptimal error trails obtained through partial-order reduction. We also analyzed the application of heuristic search and partial-order reduction in order to find shorter paths to a given state. The first step in this scheme consists of finding an error state using partial-order reduction with respect to some specifi-

cation property and a depth-first search. We showed that it is possible to reduce the length of the resulting error trail by applying A* as the search algorithm in a post-processing step. Together with partial-order reduction we could further reduce the exploration effort.

As we mentioned earlier, the working hypothesis of our paper is that "short is beautiful", i.e., that shorter error trails are preferable over longer ones in common error debugging situations. We have also argued that other approaches to error explanation (cf. [2, 31, 40, 55]) are orthogonal in that they focus on qualitative rather than quantiative properties of error trails. We intend to pursue further research to assess the impact of short error trails on their error explanation capabilities.

In [46] we analyze the combination of heuristic search and symmetry reduction to mitigate the state explosion problem of automated verification. Symmetry reduction reduces the state space to be explored to an equivalent smaller one by exploiting symmetries in the system, while heuristic search 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, we observed that in some instances the computational effort of using search heuristics may offset the time saved by heuristic techniques applied in symmetry reduction. In other work, partial-order reduction has been combined with symmetry reduction [24]. Hence, we plan to perform experiments combining all three techniques.

In [20] a fragment of SPIN's input language Promela is compiled into an action planning description language to take advantage of more involved state-to-goal approximations that planning tools offer. Refined estimates for improved error detection are introduced, such as the *relaxed plan* and *pattern database heuristic*, that come along with an *enforced hill climbing* search engine.

Work described in [23] extends the idea of using structural heuristics to improve the verification of Java byte code as originally proposed in [29, 30]. The authors of [23] devise heuristics applicable to the finite-state machine structure of the Java byte code. These heuristics have been derived from the FSM distance heuristics that we describe in this paper. That work also includes an approach to trail-directed search.

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. Alur R, Brayton R, Henzinger T, Qaderer S, Rajamani S (1997) Partial-order reduction in symbolic state space exploration. In: 9th conference on computer-aided verification (CAV). Lecture notes in computer science, vol 1254. Springer, Berlin Heidelberg New York, pp 340–351
2. Ball T, Naik M, Rajamani S (2003) From sympton to cause: Localizing errors in counterexample traces. In: 30th annual ACM symposium on principles of programming languages (POPL)
3. Ball T, Rajamani SK (2001) Automatically validating temporal safety properties of interfaces. In: 7th international SPIN workshop on software model checking. Lecture notes in computer science, vol 2057. Springer, Berlin Heidelberg New York, pp 103–122
4. Behrmann G, Fehnker A, Hune T, Larsen KG, Petterson P, Romijn J (2001) Guiding and cost-imality in UPPAAL. In: AAAI spring symposium on model-based validation of intelligence, pp 66–74
5. Behrmann G, Fehnker A, Hune T, Larsen KG, Petterson P, Romijn J, Vaandrager FW (2001) Efficient guiding towards cost-imality in uppaal. In: Conference on tools and algorithms for the construction and analysis of systems (TACAS). Lecture notes in computer science, vol 2031. Springer, Berlin Heidelberg New York
6. Bérard B, Bidoit M, Finkel A, Laroussinie F, Petit A, Petrucci L, Schnoebelen P (2001) Systems and software verification: model-checking techniques and tools. Springer, Berlin Heidelberg New York
7. Bloem R, Ravi K, Somenzi F (2000) Symbolic guided search for CTL model checking. In: ACM/IEEE Design Automation Conference (DAC), pp 29–34
8. Bonet B, Geffner H (2001) Planning as heuristic search. Artif Intell 129(1–2):5–33
9. Bosnacki D, Dams D, Holenderski L (2000) Symmetric SPIN. In: 7th SPIN workshop on software model checking. Lecture notes in computer science, vol 1885. Springer, Berlin Heidelberg New York, pp 1–19
10. Chou C-T, Peled D (1996) 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, vol 1055. Springer, Berlin Heidelberg New York, pp 241–257
11. Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge, MA
12. Cobleigh JM, Clarke LA, Osterweil LJ (2001) The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In: 23rd IEEE international conference on software engineering (ICSE), pp 37–46
13. Cohen JD (1997) Recursive hashing functions for n-grams. ACM Trans Inf Sys 15(3):291–320
14. Corbett JC, Dwyer MB, Hatcliff J, Laubach S, Pasareanu CS, Robby, Zheng H (2000) Bandera: Extracting finite-state models from Java source code. In: 22nd IEEE international conference on software engineering (ICSE)
15. Cormen TH, Leiserson CE, Rivest RL (1990) Introduction to algorithms. MIT Press, Cambridge, MA
16. Dams D, Gerth R, Grumberg O (1997) Abstract interpretation of reactive systems. ACM Trans Programm Lang Sys 19(2):111–149
17. Dechter R, Pearl J (1985) Generalized best-first strategies and the optimality of A*. J ACM 32
18. Dolev D, Klawe M, Rodeh M (1982) An O(n log n) unidirectional distributed algorithm for extrema finding in a circle. J Algorithms 3:245–260
19. Edelkamp S (1999) Data structures and learning algorithms in state space search. PhD thesis, University of Freiburg, Infix
20. Edelkamp S (2003) Promela planning. In: 10th SPIN workshop on model checking software. Lecture notes in computer science, vol 2648. Springer, Berlin Heidelberg New York, pp 197–212
21. Edelkamp S, Leue S, Lluch-Lafuente A (2004) Directed explicit-state model checking in the validation of communication protocols. Int J Softw Tools Technol Transfer 5(2–3): 247–267. `http://www.di.unipi.it/%7Elafuente/papers/index.html#phd`
22. Edelkamp S, Lluch-Lafuente A, Leue S (2001) Trail-directed model checking. In: Stoller SD, Visser W (eds) Electronic Notes in Theoretical Computer Science, vol 55. Elsevier, Amsterdam
23. Edelkamp S, Mehler T (2003) Byte code distance heuristics and trail direction for model checking Java programs. In: 2nd workshop on model checking and artificial intelligence, pp 69–76

24. Emerson EA, Jha S, Peled D (1997) Comibing partial order and symmetry reduction. In: 3rd conference on tools and algorithms for the construction and analysis of systems (TACAS). Lecture notes in computer science, vol 1217. Springer, Berlin Heidelberg New York, pp 19–34
25. Emerson EA, Sistla AP (1993) Symmetry and model checking. In: 5th international conference on computer-aided verification (CAV). Lecture notes in computer science, vol 697. Springer, Berlin Heidelberg New York, pp 463–378
26. Godefroid P (1991) Using partial orders to improve automatic verification methods. In: 2nd conference on computer-aided verification (CAV). Lecture notes in computer science, vol 531. Springer, Berlin Heidelberg New York, pp 176–185
27. Godefroid P, Khurshid S (2002) Exploring very large state spaces using genetic algorithms. In: 8th conference on tools and algorithms for the construction and analysis of systems (TACAS). Lecture notes in computer science, vol 2280. Springer, Berlin Heidelberg New York, pp 266–280
28. Graf S, Saidi H (1997) Construction of abstract state graphs of infinite systems with PVS. In: 9th conference on computer-aided verification (CAV). Lecture notes in computer science, vol 1254. Springer, Berlin Heidelberg New York
29. Groce A, Visser W (2002) Heuristic model checking for java programs. In: 9th SPIN workshop on model checking software. Lecture notes in computer science, vol 2318. Springer, Berlin Heidelberg New York, pp 242–245
30. Groce A, Visser W (2002) Model checking java programs using structural heuristics. In: International symposium on software testing and analysis (ISSTA). ACM Press, New York
31. Groce A, Visser W (2003) What went wrong: Explaining counterexamples. In: Workshop on software model checking (SPIN). Lecture notes in computer science, vol 2648. Springer, Berlin Heidelberg New York, pp 121–135
32. Object Management Group (1997) The common object request broker architecture and specification. Revised version 2.1
33. Hart PE, Nilsson NJ, Raphael B (1968) A formal basis for heuristic determination of minimum path cost. IEEE Trans Sys Sci Cybern 4:100–107
34. Holzmann GJ (1990) Design and validation of computer protocols. Prentice-Hall, Englewood Cliffs, NJ
35. Holzmann GJ (1997) The model checker Spin. IEEE Trans Softw Eng 23(5):279–295
36. Holzmann GJ (2003) The Spin model checker, primer and reference manual. Addison-Wesley, Reading, MA
37. Holzmann GJ, Godefroid P, Pirottin D (1992) Coverage preserving reduction strategies for reachability analysis. In: 12th international conference on protocol specification, testing, and verification (PSTV)
38. Holzmann GJ, Smith MH (2002) An automated verification method for distributed systems software based on model extraction. IEEE Trans Softw Eng 28(4):364–377
39. Ip CN, Dill DL (1996) Better verification through symmetry. Formal Methods Sys Des 9:41–75
40. Jin H, Ravi K, Somenzi F (2002) Fate and free will in error traces. In: 8th conference on tools and algorithms for the construction and analysis of systems (TACAS). Lecture notes in computer science, vol 2280. Springer, Berlin Heidelberg New York, pp 445–459
41. Kamel M, Leue S (2000) Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. Int J Softw Tools Technol Transfer 2(4): 394–409
42. Kamel M, Leue S (2000) 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, vol 1785. Springer, Berlin Heidelberg New York, pp 471–486
43. Korf RE (1985) Depth-first iterative-deepening: An optimal admissible tree search. Artif Intell 27(1):97–109
44. Kurshan RP, Levin V, Minea M, Peled D, Yenigun H (1998) Static partial order reduction. In: 4th conference on tools and algorithms for construction and analysis of systems (TACAS). Lecture notes in computer science, vol 1384. Springer, Berlin Heidelberg New York, pp 345–357
45. Lerda F, Sinha N, Theobald M (2003) Symbolic model checking of software. In: Electronic notes in theoretical computer science, vol 89. Elsevier, Amsterdam
46. Lluch-Lafuente A (2003) Symmetry reduction and heuristic search for error detection in model checking. In: 2nd workshop on model checking and artificial intelligence (MoChArt), pp 77–86
47. Lluch-Lafuente A, Leue S, Edelkamp S (2002) Partial order reduction in directed model checking. In: SPIN workshop on model checking software. Lecture notes in computer science, vol 2318. Springer, Berlin Heidelberg New York, pp 112–127
48. Manna Z, Pnueli A (1992) The temporal logic of reactive and concurrent systems: specification. Springer, Berlin Heidelberg New York
49. McVitie DG, Wilson LB (1971) The stable marriage problem. Commun ACM 14(7):486–492
50. Nalumasu R, Gopalakrishnan G (1997) A new partial order reduction algorithm for concurrent system verification. In: Hardware description languages and their applications (CHDL). Chapman & Hall, London
51. Nilsson NJ (1980) Principles of artificial intelligence. Tioga, Palo Alto, CA
52. Pearl J (1985) Heuristics. Addison-Wesley, Reading, MA
53. Peled DA (1996) Combining partial order reductions with on-the-fly model-checking. Formal Methods Sys Des 8:39–64
54. Peled DA (1998) Ten years of partial order reduction. In: 10th conference on computer-aided verification (CAV). Lecture notes in computer science, vol 1427. Springer, Berlin Heidelberg New York, pp 17–28
55. Sharygina N, Peled D (2001) A combined testing and verification approach for software reliability. In: Formal methods of increasing software productivity, internatioanl symposioum of Formal Methods Europe, pp 611–628
56. Valmari A (1991) A stubborn attack on state explosion. Lecture notes in computer science, vol 531. Springer, Berlin Heidelberg New York, pp 156–165
57. Wah BW, Shang Y (1995) Study of ida*-style searches. J Tools Artif Intell 3(4):493–523
58. Yang CH, Dill DL (1998) Validation with guided search of the state space. In: Conference on design automation (DAC), pp 599–604