Directed Explicit-State Model Checking in the Validation of Communication Protocols

Stefan Edelkamp Stefan Leue
Alberto Lluch-Lafuente

Institut für Informatik
Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee Geb. 051
D-79110 Freiburg, Germany
email: {edelkamp,lafuente,leue}@informatik.uni-freiburg.de

Technical Report No.161

Report Date: October 19, 2001

Abstract

The success of model checking is largely based on its ability to efficiently locate errors in software designs. If an error is found, a model checker produces a trail that shows how the error state can be reached, which greatly facilitates debugging. However, while current model checkers find error states efficiently, the counterexamples are often unnecessarily lengthy, which hampers error explanation. This is due to the use of "naive" search algorithms in the state space exploration.

In this paper we present approaches to the use of heuristic search algorithms in explicit-state model checking. We present the class of A* directed search algorithms and propose heuristics together with bitstate compression techniques for the search of safety property violations. We achieve great reductions in the length of the error trails, and in some instances render problems analyzable by exploring a much smaller number of states than standard depth-first search. We then suggest an improvement of the nested depth-first search algorithm and show how it can be used together with A* to improve the search for liveness property violations. Our approach to directed explicit-state model checking has been implemented in a tool set called HSF-SPIN. We provide experimental results from the protocol validation domain using HSF-SPIN.

Keywords: Model Checking, Directed Search, Protocol Validation

1 Introduction

Model Checking [4] is a formal analysis technique that has been developed to automatically validate[1] functional properties for software or hardware systems. The properties are usually specified using some sort of a temporal logic or using automata. There are two primary approaches to model checking. First, *symbolic* model checking [24] uses binary decision diagrams to represent the state set. The second formalization uses an *explicit* representation of the system's global state graph. An explicit state model checker evaluates the validity of the temporal properties over the model by interpreting its global state transition graph as a Kripke structure. In this paper we focus on explicit state model checking and its application to the validation of communication protocols. The protocol model we consider is that of collections of extended communicating finite state machines as described, for instance, in [3] and [15]. Communication between two processes is either realized via synchronous or asynchronous message passing on communication channels (queues) or via global variables. Sending or receiving a message is an event that causes a state transition. The system's global state space is generated by the asynchronous cross product of the individual communicating finite state machines (CFSMs). We follow the Promela computational model [17].

The use of model checking in system design has one great advantage over the use of deductive formal verification techniques. Once the requirements are specified and the model has been programmed, model checking validation can be implemented as a push-button process that either yields a positive result, or returns an error trail. Two primary strategies for the use of model checking in the system design process can be observed.

- *Complete validation* is used to certify the quality of the product or design model by establishing its absolute correctness. However, due to the large size of the search space for realistic systems it is hardly ever possible to explore the full state space in order to decide about the correctness of the system. In these cases, it either takes too long to explore all states in order to give an answer within a useful time span, or the size of the state space is too large to be stored within the bounds of available main memory.

- The second strategy, which also appears to be the more commonly one used, is to employ the model checker as a *debugging aid* to find residual design and code faults. In this setting, one uses the model checker as a search tool for finding violations of desired properties. Since complete validation is not intended, it suffices to use hashing-based partial exploration methods that allow for covering a much larger portion of the system's state space than if complete exploration is needed.

When pursuing debugging, there are some more objectives that need to be

[1] Within the scope of this paper we use the word "validation" to denote the experimental approach to establishing the correctness of a piece of software, while we use the word "verification" to denote the use of formal theorem proving techniques for the same purpose.

addressed. First, it is desirable to make sure that the length of the counterexample is short, so that error trails are easy to interpret. Second, it is desirable to guide the search process to quickly find a property violation so that the number of explored states is small, which means that larger systems can be debugged this way. To support these objectives we present our approach to *directed model checking*, i.e. model checking combined with heuristic search.

Our model-checker HSF-SPIN extends the SPIN framework with various heuristic search algorithms to support directed model checking. Experimental results show that in many cases the number of expanded nodes and the length of the counter-examples are significantly reduced. HSF-SPIN has been applied to the detection of deadlocks, invariant and assertion violations, and to the validation of LTL properties. In most instances the estimates used in the search are derived from the properties to be validated, but HSF-SPIN also allows some designer intervention so that targets for the state space search can be specified explicitly in the Promela code.

We propose an improvement of the nested depth-first search algorithm that exploits the structure of never claims. For a broad subset of the specification patterns described in [8], such as *Response* and *Absence*, the proposed algorithm performs less transitions during state space search and finds shorter counterexamples compared to classical nested depth-first search. Given a Promela *never claim* A the algorithm automatically computes a partitioning of A in linear time with respect to the number of states in A. The obtained partitioning into non-, fully and partially accepting strongly connected components will be exploited during state space exploration.

Precursory Work. Much of the work originally appeared in two 2001 conference papers presented at the AAAI Spring Symposium on Model Based Validation of Intelligence [10] and at the 8th SPIN Workshop on Model Checking Software [9]. The former paper considers safety property analysis for simple protocols. The latter paper extends this work by providing an approach to validating LTL-specified liveness properties and experimenting with a larger set of protocols. The two papers have been merged, edited and refined for a self-contained document. Moreover, the correctness result for the improved nested depth-first search has been added and new experiments have been evaluated.

Structure of Paper. In Section 2 we review automata-based model checking. Section 3 introduces into directed search algorithms, as for instance A*. Heuristic estimate functions to be used in safety property analysis of communication protocols are suggested in Section 4. We present the HSF-SPIN tool set in Section 5 and present experimental results for safety properties in Section 6. In Section 7 we propose an improvement to the nested depth-first search algorithm used in the analysis of liveness properties and show how this algorithm can be combined with heuristic search. Experimental results on liveness property validation are given in Section 8. We discuss related work in Section 9 and conclude in Section 10.

2 Automata-based Model Checking

In this Section we review the automata theoretic framework for explicit state model checking (c.f. [4]), describe the validation algorithms in use, and present a practical model checker, the SPIN tool set.

2.1 Automata-theoretic Framework

Since we model reactive systems with infinite behaviors, the appropriate formalization for words over state sequences of these systems are Büchi automata. They inherit the syntactic structure of finite state automata but have a different acceptance condition. An infinite run of a Büchi automaton $\mathcal{A}$ over an alphabet $\Sigma_{\mathcal{A}}$ of state symbols is accepting if the set of elements of $\Sigma_{\mathcal{A}}$ that appear infinitely often in the run has a non-empty intersection with the set of accepting states of $\mathcal{A}$. This extends to finite runs by assuming that the final state will be repeated forever. The language $L(\mathcal{A}) \subseteq \Sigma^*_{\mathcal{A}}$ consists of all accepting runs of $\mathcal{A}$. It is sometimes helpful to specify requirements on reactive systems by using some form of a Temporal Logic. In this paper we use Linear Time Temporal Logic (LTL) as defined in [23]. In LTL, the operator $\Box$ represents the modality *globally* (G) and the operator $\Diamond$ represents the modality *eventually* (F).

In automata-based Model Checking we are interested in determining whether the system M, represented by Büchi automaton $\mathcal{B}$, satisfies a property specification S, given by another Büchi automaton $\mathcal{A}$. $\mathcal{A}$ can either be given directly, or it can be automatically derived from an LTL property specification. While this derivation is exponential in the size of the formula, typical property specifications result in small LTL formulae so that this complexity is not a practical problem. The Büchi automaton $\mathcal{B}$ satisfies $\mathcal{A}$ iff $L(\mathcal{B}) \subseteq L(\mathcal{A})$. This is equivalent to $L(\mathcal{B}) \cap \overline{L(\mathcal{A})} = \emptyset$, where $\overline{L(\mathcal{A})}$ denotes the complement of $L(\mathcal{A})$. Note that Büchi automata are closed under complementation. In practice, $\overline{L(\mathcal{A})}$ can be computed more efficiently by deriving a Büchi automaton from the negation of an LTL formula. Therefore, in the SPIN validation tool LTL formulae representing a desired property are first negated, and then translated into an equivalent Büchi automaton. In the parlance of the SPIN model checker [18] and its Promela input language this automaton is called a *never claim*, and we will adopt this terminology throughout this paper.

As an example we consider the commonly used *response* property which states that, whenever a certain request event occurred, a response event will eventually follow. Assume that the state following the occurrence of the request is represented by the state predicate p, and that a state following the response is denoted by q. The corresponding LTL formula is $\phi : \Box(p \rightarrow \Diamond q)$ and its negation is $\neg\phi : \Diamond(p \wedge \Box \neg\, q)$. The Büchi automaton and the corresponding Promela never claim for the negated response property are illustrated in Figure 1.

The emptiness of $L(\mathcal{B}) \cap \overline{L(\mathcal{A})}$ is determined using an on-the-fly algorithm based on the synchronous product of $\mathcal{N}$ and $\mathcal{B}$, where $L(\mathcal{N}) = \overline{L(\mathcal{A})}$. Assume that $\mathcal{N}$ is in state s and $\mathcal{B}$ is in state t. $\mathcal{B}$ can perform a transition out of t if $\mathcal{N}$ has a successor state s' of s such that the label of the edge from s to s' represents

Figure 1: Büchi automaton for response property (top left) and for its negation (bottom right).

a proposition satisfied in t. A run of the synchronous product is accepting if it contains a cycle through at least one accepting state of $\mathcal{N}$. $L(\mathcal{B}) \cap \overline{L(\mathcal{A})}$ is empty if the synchronous product does not have an accepting run.

We use the standard distinction of safety and liveness properties. Safety properties refer to states, whereas liveness properties refer to paths in the state transition diagram. Safety properties can be validated through a simple depth-first search on the system's state space, while liveness properties require a two-fold nested depth-first search. When property violations are detected, the model checker will return a witness (counterexample) which consists of a trace of events or states encountered.

2.2 Search Algorithms

For the validation of safety properties a simple complete state graph traversal algorithm is sufficient. This is usually either a depth-first (DFS) or a breadth-first (BFS) search algorithm. When a property violating state is encountered, the search stack contains the witness that will be made available to the user. BFS finds errors with minimal witness length, but is rather memory inefficient. DFS is more memory efficient, but tends to produce witnesses of non-optimal length.

Since liveness properties refer to execution paths, a different search approach is needed. The detection of liveness property violations entails searching for accepting cycles in the state graph. This is typically achieved by nested depth-first search (Nested-DFS) that can be implemented with two stacks as shown in Figure 2. As for safety properties, the search stacks will be used to construct the witness. In case a property violation is discovered, the first stack will contain the path into an accepting state, while the second stack will illustrate the cycle through the accepting state.

2.3 The Model Checker SPIN

SPIN [18] is a model checking tool implementing the above discussed approach to automata-based model checking. 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

Nested-DFS(s)
  $hash(s)$
  for all successors s' of s **do**
    if s' not in the hash table **then** **Nested-DFS**(s')
  if $accept(s)$ **then** **Detect-Cycle**(s)

Detect-Cycle(s)
  $flag(s)$
  for all successors s' of s **do**
    if s' on *Nested-DFS*-Stack **then**
      exit *LTL-Property violated*
    else if s' not flagged **then** **Detect-Cycle**(s')

Figure 2: Nested Depth-First Search

C-like data structures. Concurrency in SPIN is evaluated using an interleaving approach. Properties can be specified in various ways. To express safety properties, the Promela code can be augmented with assertions and deadlock state characterizations. In order to express liveness properties, Promela models can be extended by never claims that express undesired properties of the model. SPIN also provides an automatical linear temporal logic (LTL) to never claim translator. 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. SPIN has a line-oriented as well as a graphical user interface, called XSPIN. For a more detailed discussion of SPIN we refer to the literature on the SPIN web site[2].

2.4 Error Trails

If property violations are found, error trails contain important debugging information. Succinctness of these trails is essential for an easy comprehension of the discovered design faults. Lengthy trails can impede proper error trail interpretation.

We illustrate the impact of long error trails with the following example. We refer to the preliminary design of a Plain Old Telephony System (POTS) that we first presented in [20]. This model was generated with the visual modeling tool VIP. It is a "first cut" implementation of a simple two-party call processing, and we know that it is full of faults of various kind. However, in [20] we used SPIN to show that this model is actually capable of connecting two telephones. The

[2] `netlib.bell-labs.com/netlib/spin`.

model consists of two user processes `UserA` and `UserB` representing the environment behaviour of the switch, as well as two phone handler processes `PhoneHA` and `PhoneHB` representing the software instances that control the internal operation of the switch according to signals (on-hook, off-hook, etc.) received from the environment. Due to space constraints we have to rely on an intuitive understanding of call processing behaviour and the type of signals that are used, for a more detailed description we refer to [20].

Our objective now is to use SPIN in order to debug the POTS model. We are first interested in knowing whether certain inconsistent global system states are reachable. For instance, such an inconsistent state is reached when 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. Let p and q denote state propositions that are true when phone handlers A and B are in the conversation state, respectively. Let r and s denote state propositions representing the fact that phones A and B are in the conversation state, respectively. The absence requirement for this inconsistent global system state, which is a safety property, can be characterized by the LTL formula

$$\neg\Diamond(p \wedge \neg q \wedge r \wedge s).$$

We used SPIN to validate this property. It turns out not to be valid and SPIN produces an error trail leading into a global system state violating the property as partially illustrated in Figure 3. For the experienced in analyzing call processing sequences it becomes clear that the undesired state is reachable because of race conditions and a lack of synchronization between the `UserB` and the `PhonHB` processes, which probably calls for using synchronous communication at this interface. The error trail that SPIN produces has a length of 2,765 steps and comprises 462 message exchanges - it is obvious that analyzing a trail of that length to locate the cause of an error is an arduous task. The length of the trail is surprising since using some backward analysis, and when knowing the underlying state machine model, it is easy to come up with a much shorter trail by hand, for instance the trail comprising just 16 messages given in Figure 4.

The trail length phenomenon is partly due to the high degree of nondeterminism inside the system which can be attributed to the highly concurrent nature of a telephony switch. Another contributing factor is the search strategy that SPIN uses when exploring the system's state space. Resolution of nondeterminism in Promela is random, but SPIN implements this using a fixed priority scheme based on the lexical structure of the Promela model[3]. SPIN will first explore many execution sequences that do not lead to the establishment of a phone call. This means for instance that one phone calls the other, but then decides to hang up, or both phones try to call each other concurrently, before the call sequences converges towards the successful establishment of a call. The depth-first search strategy that SPIN employs will first try to explore all action variants of the first process, and then try out the next process, and so on. How-

[3] Roughly speaking, this means the lexically first transition in the "first" proctype instance is preferred over other concurrently enabled transitions.

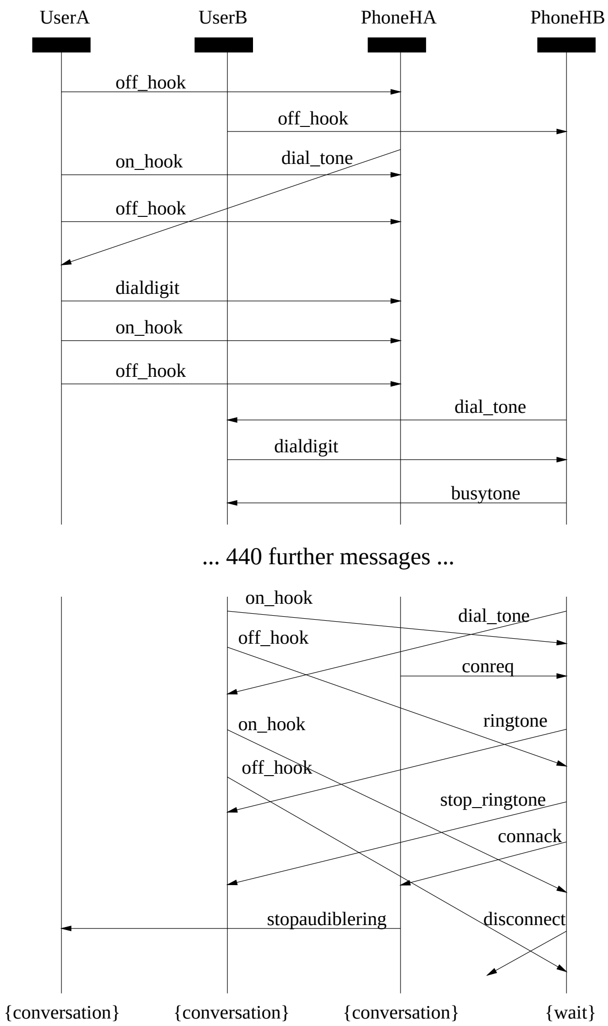

Figure 3: POTS example, error trail produced by SPIN. Names in curly brackets denote local control states reached at the end of the trail.

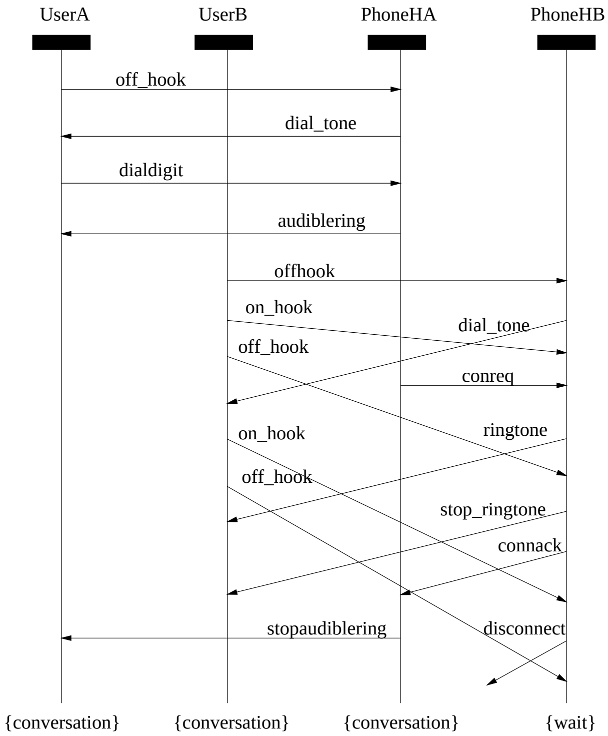

Figure 4: POTS example, manually generated shorter error trail.

ever, the target state would be reached much more quickly if all processes did a few steps so that a phone call was established. In conclusion, SPIN is following a rather uniformed search strategy that neither takes knowledge about the model nor knowledge about the property to be validated into account when deciding which of the possible successor states to explore first. If, however, the state space of the `PhoneA`, `PhoneB` and `PhoneHA` processes were explored in such a way that every state transition brought them nearer to their own local conversation state and if `PhoneHB` avoided the conversation state, and if globally such transitions were preferred over non-approximating transitions, then a much shorter error trail into the property violating state could be expected. It is the objective of this paper to present guided search algorithms using heuristic guidelines in the state exploration similar to the one just described. When discussing experimental results, we will see that for the POTS example the automatically obtained shortest error trail is 1.5 orders of magnitude shorter than the one generated by SPIN's exploration.

3 Heuristic Search Algorithms

In this Section we introduce heuristic search algorithms as alternatives to complete state space exploration in model checking. We will restrict the discussion to safety property searches and extend the discussion to liveness properties later on in this paper.

The detection of a safety property violation is equivalent to finding a state in which the property is violated. The algorithms used for finding the property violating states are typically depth-first and breadth-first searches. Depth-first search (DFS) is memory efficient, but does not provide optimal solutions. Breadth-first search (BFS), on the other hand, is complete and optimal but very inefficient.

State space exploration in model checking safety properties can be casted as a search for a path to a failure state in the underlying problem graph. Since this graph is implicitly generated by node expansions, in contrast to ordinary graph algorithms the search terminates once a target state has been found. BFS and DFS explore the state space without additional knowledge about the search goal. The selection of a successor node in these algorithms is following a fixed, deterministic selection scheme. Heuristic search algorithms, however, take additional search information in form of an estimation function into account. This function returns a number representing the desirability of expanding a node. When the nodes are ordered so that the one with the best evaluation is expanded first and if the evaluation function estimates the cost of the cheapest path from the current state to a desired one, the resulting greedy best-first search (BF) often finds solutions fast. However, it may suffer from the same defects as depth-first search – it is not optimal and the search may be stuck in dead ends or local minima.

Algorithm A* [16] combines best-first and breadth-first search for a new evaluation function $f(u)$ by summing the generating path length $g(u)$ and the

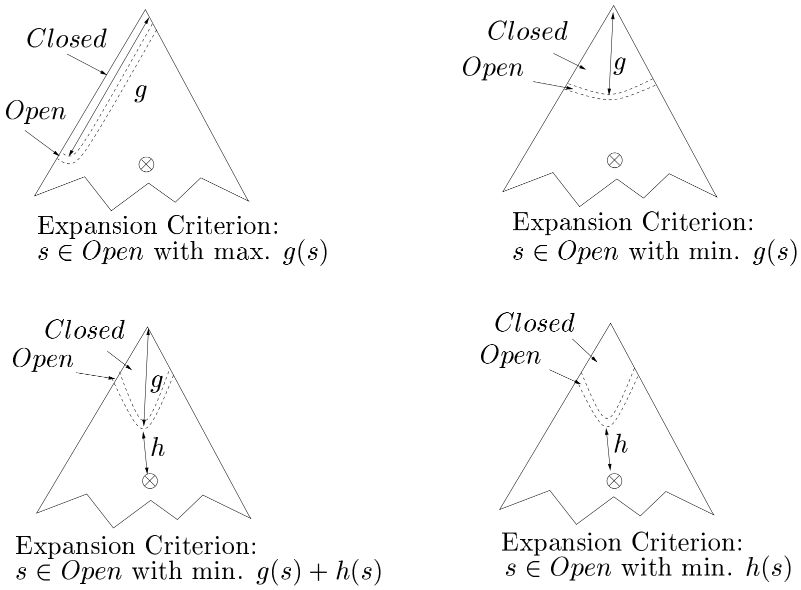

Figure 5: Different search strategies: DFS (top left), BFS (top right), A* (bottom left) and BF (bottom right)

estimated cost $h(u)$ of the cheapest solution starting from u. Figure 5 displays the effect of A* compared to DFS, BFS and BF and Table 1 depicts the algorithm in pseudo code. The node expansion of u is indicated by access to the successor set $\Gamma(u)$. The set *Closed* denotes the set of all already expanded nodes and the list *Open* contains all generated but not yet expanded nodes. Similar to Dijkstra's single-source shortest path algorithm [7], A* successively extracts the node u with minimal merit $f(u)$ from the set *Open* and terminates if this node represents a failure state.

As the combined merit $f(u) = g(u)+h(u)$ merely changes the ordering of the nodes to be expanded, on finite problem graphs A* is complete. Moreover, by changing the weights of the edges in the problem graph from 1 to $1+h(v)-h(u)$, it can also be observed that A* in fact performs the same computation as Dijkstra's single-source shortest-path algorithm on the re-weighted graph. If for all edges (u, v) we have $1 + h(v) - h(u) \geq 0$, optimality of A* is inherited from the optimality of Dijkstra's algorithm. It can also be shown that the path length for every expanded node is optimal, so that we correctly terminate the search at the first target node.

If $1+h(v)-h(u) < 0$, negatively weighted edges affect the correctness proof of Dijkstra's algorithm. In this case we have $f(u)+1+h(v)-h(u) < f(v)$ such that nodes that have already been expanded might be encountered on a shorter path. Contrary to Dijkstra's algorithm, A* deals with them by possibly re-inserting nodes from the set of already expanded nodes into the set of *Open* nodes (re-

A*(s)

$Open \leftarrow \{(s, h(s))\}$; $Closed \leftarrow \{\}$

while $(Open \neq \emptyset)$

  $u \leftarrow Deletemin(Open)$; $Insert(Closed, u)$

  if $(failure(u))$ **exit** *Safety Property Violated*

  for all v **in** $\Gamma(u)$

    $f'(v) \leftarrow f(u) + 1 + h(v) - h(u)$

    if $(Search(Open, v))$

      if $(f'(v) < f(v))$

        $DecreaseKey(Open, (v, f'(v))$

    else if $(Search(Closed, v))$

      if $(f'(v) < f(v))$

        $Delete(Closed, v)$; $Insert(Open, (v, f'(v))$

    else $Insert(Open, (v, f'(v))$

Table 1: The A* algorithm searching for violations of safety properties.

opening). On every path from s to u the accumulated weights in the two graph structures differ by $h(s)$ and $h(u)$ only. Consequently, re-weighting cannot introduce negatively weighted cycles so that the problem remains (optimally) solvable. One can show that given a lower bound estimate (admissible heuristic) the solution returned by the A* algorithm with re-opening is indeed a shortest one [13]. The main argument is that there is always a correctly estimated node on an optimal path in the set *Open*. This node has to be considered before expanding any non-optimal goal node.

Figure 6 depicts the impact of heuristic search in a grid graph. If h is the trivial constant zero function, A* reduces to Dijkstra's algorithm, which in case of uniform graphs further collapses to BFS. Therefore, starting with s all depicted nodes shown are generated until the goal node t is expanded. If we use $h(u)$ as the Euclidean distance to node t, then only the nodes in the hatched region are ever removed from the *Open* set.

Algorithm A* has one severe drawback. Once the space resources for storing all expanded and generated nodes are exhausted, no further progress can be made. Therefore, the iterative deepening variant of A*, IDA* [21] for short, counterbalances time for space. It traverses the tree expansion of the problem graph instead of the problem graph itself. To guarantee completeness and optimality, IDA* expands all nodes in consecutive bounded depth-first search traversals with an increasing threshold value for each possible merit value. Since the average number of successors is often larger than one the tree expansion grows exponentially with increasing depth. Therefore, the last iteration in IDA* usually dominates the search effort.

Due to the depth-first structure of IDA*, duplicate state expansions may

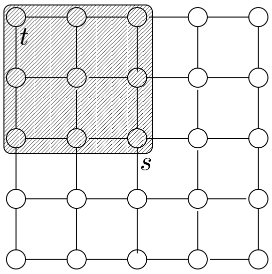

Figure 6: The effect of heuristic search in a grid graph.

not be detected, resulting in redundancy. Therefore, similar to depth-first and best-first search as long as memory is available, all generated nodes are kept in a transposition table. To allow dynamic updates of node information, for each node in the table the shortest generating path length and the corresponding predecessor are also maintained.

To improve duplicate detection, IDA* can be combined with bit-state hashing [17] which hashes an entire state vector into a single bit wide table. The bit position indicates whether the state has been reached before, or not. In single bit-state hashing, a hash function h_1 maps a state S to position $h_1(S)$; S is stored by setting the bit $h_1(S)$ and searched by querying $h_1(S)$. Double bit-state hashing often improves state space coverage by applying a second hash-function h_2. A state S is stored in setting $h_1(S)$ and $h_2(S)$ and detected as a duplicate if both bits are set.

Bit-state hashing implies that a retrieved node might be an unexpected synonym, since there is no way to distinguish a real duplicate from a false one. Therefore, re-opening such a node inside IDA* is dangerous since the information of generating path length and predecessor path length might be false. Subsequently, we avoid reopening in Partial IDA*. Note that the advantage of Partial IDA* compared to A* is that it can track the solution path on the recursion stack which means that no predecessor link is needed. Reopening in IDA* will not be encountered when the heuristic function is consistent. In this case the priorities $f = g + h$ increase on any generating path, since $f(u) = g(u)+h(u) \leq g(u)+h(u)+1+h(v)-h(u) = g(u)+1+h(v) = g(v)+h(v) = f(v)$ for all edges (u, v) in the tree expansion of the problem graph. Most practical heuristics satisfy this criterion. The negative impact of partial state space coverage due to bitstate search is reduced by repeating the search with restarts on different hash functions.

4 Search Heuristics for Safety Properties

In this Section we introduce search heuristics used by our tool HSF-SPIN in the analysis of safety properties for Promela models. We use S to denote a

f	$H_f(S)$	$\overline{H}_f(S)$
$true$	0	∞
$false$	∞	0
a	**if** a **then** 0 **else** 1	**if** a **then** 1 **else** 0
$\neg g$	$\overline{H}_g(S)$	$H_g(S)$
$g \vee h$	$\min\{H_g(S), H_h(S)\}$	$\overline{H}_f(S) + \overline{H}_g(S)$
$g \wedge h$	$H_g(S) + H_h(S)$	$\min\{\overline{H}_g(S), \overline{H}_h(S)\}$

Figure 7: Definition of H_f for Boolean expressions f.

global system state of the model. In S we have a set $\mathcal{P}(S) \subseteq \{P_i \mid i \geq 0\}$ of currently active processes. For the sake of simplicity we assume a fixed number of processes and write $\mathcal{P}$ instead of $\mathcal{P}(S)$. For a process P_i we use pc_i to refer to the current local control state. T_i denotes the set of transitions within the proctype instance P_i and S_i denotes the set of local states of P_i.

Violation of Invariants System invariants are state predicates that hold over every global system state S. When searching for invariant violations it is helpful to estimate the number of system transitions until a state is reached where the invariant is violated. Given a logical global state predicate f, let $H_f(S)$ be an estimation of the number of transitions necessary until a state S' is reached where f holds, starting from state S. Similarly, let $\overline{H}_f(S)$ denote the number of transitions necessary until f is violated, which is helpful when validating negations of state predicates. Let a be a Boolean variable, and g and h logical predicates. We give a recursive definition of H_f as a function of f, with the first part of the definition given in Figure 7.

In the definition of $H_{g\wedge h}$ and $\overline{H}_{g\vee h}$, the use of *plus* $(+)$ suggests that g and h are independent, which may not be true. Consequently, the estimate is not necessarily a lower bound, affecting the optimality condition for A*. Since it is our goal to obtain short but not necessarily optimal paths, we tolerate these inadequacies. To obtain lower bounds, we may replace *plus* $(+)$ with max.

Formulae describing system invariants may contain other terms, such as relational operators and Boolean functions over queues. We extend the definition of H_f and $\overline{H}_f$ as shown in Figure 8. The function $q?[t]$ refers to the expression that is true when the message at the head of queue q is tagged with a message of type t. All other functions are self-explaining. The symbol $\otimes$ represents relational operators $(=, \neq, \leq, <, >, \geq)$.

Note that the estimate is coarse but nevertheless very effective in practice. It is possible to refine these definitions for specific cases. For instance, $H_{a=b}$ can be defined as $a - b$ in case $a \geq b$ and a is only ever decremented and b is only ever incremented. However, we have not pursued these refinements any further.

Another statement that typically appears in system invariants is the *at* predicate which expresses that a process P with a process id *pid* of a given proctype

f	$H_f(S)$
$full(q)$	$capacity(q) - length(q)$
$empty(q)$	$length(q)$
$q?[t]$	length of minimal prefix of q without t (+1 if q lacks message tagged with t)
$a \otimes b$	if $a \otimes b$ then 0, else 1
f	$\overline{H}_f(S)$
$full(q)$	if $full(q)$ then 1, else 0
$empty(q)$	if $empty(q)$ then 1, else 0
$q?[t]$	if $head(q) \neq t$ then 0, else maximal prefix of t's
$a \otimes b$	if $a \otimes b$ then 1, else 0

Figure 8: Definition of H_f for Boolean queue expressions and relational operators in f.

f	$H_f(S)$	$\overline{H}_f(S)$
$i@s$	$D_i(pc_i, s)$	if $pc_i = s$ 1, else 0

Figure 9: Definition of H_f for control state predicates in f.

PT is in its local control state s[4]. We will write this as $i@s$, with $s \in S_i$. The corresponding definition is given in Figure 9. We use pc_i to express the local state of process P_i in the current global state S. The value $D_i(u, v)$ is the minimal number of transitions necessary for the finite state machine P_i to reach state u starting from state v, where $u, v \in S_i$. The matrix D_i can be efficiently pre-computed with the all-pairs shortest-path algorithm of Floyd/Warshall in $O(|S_i|^3)$ time [6]. Note that $|S_i|$ is small in comparison to the overall search space.

Violations of Assertions The Promela statement `assert` allows to label the model with logical assertions. Given that an assertion a labels a transition (u, v), with $u, v \in T_i$, then we say a is violated if the formula $f = (i@u) \wedge \neg a$ is satisfied.

Deadlock Detection In concurrent systems, a deadlock occurs if at least a subset of processes and resources is in a cyclic wait situation. In Promela, S is a deadlock state if there is no possible transition from S to a successor state S' and at least one of the processes of the system is not in a *valid end-state*[5]. Hence, no process has a statement that is executable. In Promela, there

[4]In Promela this is expressed as `PT[pid]@s`.

[5]In Promela, a local control state can be labeled as `end` to indicate that it is a valid end state, i.e., that the system may terminate if the process is in that state.

$label(t)$	$executable(t,S)$
`q?x`, q asynchronous channel	$\neg empty(q)$
`q?t`, q asynchronous channel	$q?[t]$
`q!m`, q asynchronous channel	$\neg full(q)$
`q?x,q!x`, q synchronous channel	$false$
`condition c`	c

Figure 10: Function *executable*, where `x` is a variable, and `t` is a tag.

are statements that are always executable, amongst others assignments, `else` statements, and `run` statements used to start processes. For other statements, such as send or receive operations or statements that involve the evaluation of a guard, executability depends on the current state of the system. For example, a send operation `q!m` is only executable if the queue `q` is not full. The following enumeration describes executability conditions for a significant portion of Promela statements:

1. Asynchronous untagged receive operations (`q?x`, with `x` variable) are not executable if the queue is empty. The corresponding formula is $\neg\, empty(q)$.

2. Asynchronous tagged receive operations (`q?t`, with `t` tag) are not executable if the head of the queue is a message tagged with a different tag than `t` yielding the formula $\neg\, q?[t]$.

3. Asynchronous send operations (`q!m`) are not executable if `q` is full indicated by the predicate $\neg full(q)$.

4. Synchronous operations over a queue (rendezvous) block the process executing the operation when no other process is also blocked on the queue trying to execute an inverse operation. For simplicity we consider that synchronous operations always block the process, i.e. is executability is defined by the predicate $false$.

5. Conditions (Boolean expressions) are not executable if the value of the condition is false corresponding to the term c.

The Boolean function *executable*, ranging over tuples of Promela statements and global system states, is summarized in Figure 10.

We now propose estimator functions for the number of transitions necessary from the current state to reach a deadlock state.

Active Processes In a deadlock state, all processes are blocked. The active process heuristics uses the number of active or non-blocked processes in a given state S:

$$H_{ap}(S) = \sum_{P_i \in \mathcal{P} \wedge active(i,S)} 1$$

where $active(i,S)$ is defined as

$$active(i,S) \equiv \bigvee_{t=(pc_i,v)\in T_i} executable(t)$$

Given that the range of *active* is $[0..|\mathcal{P}|]$, the active processes heuristic may not be very informative for protocols involving a small number of processes.

Formula Characterization In a deadlock state S', all process are blocked, i.e.,

$$deadlock \equiv \bigwedge_{P_i\in\mathcal{P}} blocked(i, pc_i(S'), S')$$

The predicate $blocked(i, pc_i(S'), S')$ is defined as: in the system state S' the process P_i is blocked in its local state $u = pc_i(S')$, which is formalized as:

$$blocked(i,u,S) \equiv (i@u) \wedge \bigwedge_{t=(u,v)\in T_i} \neg\ executable(t,S).$$

Since we do not know the set of states in which the system will go into a deadlock in advance, we cannot compute $H_{deadlock}$ at exploration time and therefore resort to an approximation. A first approximation step is to determine in which states a process *can* block. In this approximation, a process P_i is "blocked" if $blocked(i,u,S)$ is valid for *some* $u \in C_i$, with C_i being the set of dangerous states of P_i. A state is dangerous if all of its outgoing transitions have the potential of not being enabled, i.e., no outgoing transition has an enabling (or executable) condition equivalent to the predicate *true*. Note that we classify valid and invalid end states as dangerous by default. We define the approximation *blocked'*(i,S) as a predicate for process P_i to be blocked in system state S:

$$blocked'(i,S) = \bigvee_{u\in C_i} blocked(i,u,S)$$

and approximate the deadlock formula with

$$deadlock' = \bigwedge_{P_i\in\mathcal{P}} blocked'(i,S).$$

Obviously, $H_{deadlock'}$ has to be computed at exploration time, depending on a characterization of control states as dangerous. With this definition all transitions corresponding to read and send operations as well as conditional expressions are considered dangerous. Additionally, to further guide the search the designer can characterize selected control states as dangerous by inserting labels in the Promela model.

5 The HSF-SPIN Tool Set

We chose SPIN as a basis for HSF-SPIN. It inherits most of the efficiency and functionality from the original source of SPIN as well as the sophisticated search capabilities from the Heuristic Search Framework (HSF) [14]. HSF-SPIN uses a large subset of Promela as modeling language. HSF-SPIN possesses a refined state description of SPIN to incorporate solution length information, transition labels and predecessors for solution extraction. It provides an interface consisting of a node expansion function, initial and goal specification. In order to direct the search, we implemented different heuristic estimates. HSF-SPIN writes SPIN-compatible trail information that can be visualized in the XSPIN interface. As when working with SPIN, the validation of a model with HSF-SPIN is done in two phases: first the generation of an analyzer of the model, and second the validation run. The protocol analyzer is generated with the program `hsf-spin` which is a modification of the SPIN analyzer generator. By executing `hsf-spin -a <model>` several `c++` files are generated. These files are part of the source of the model checker for the given model. They are compiled and linked with the rest of the implementation, incorporating, for example, data structures, search algorithms, heuristic estimates, statistics and solution generation. HSF-SPIN also supports *bit-state hashing* by implementing Partial IDA*. HSF-SPIN can be invoked with different parameters: kind of error to be detected, property to be validated, algorithm to be applied, heuristic function to be used, weighting of the heuristic estimator. HSF-SPIN allows textual simulation to interactively traverse the state space which greatly facilitates in explaining witnesses that have been found.

HSF-SPIN is still a prototype. Therefore, its performance in terms of time and space cannot compete with SPIN. For example, an exhaustive exploration of the state space generated by the GIOP protocol parameterized with 2 clients and 2 servers is performed by SPIN (without partial order reduction) in 226 seconds with a memory consumption of 236 MB, while our tool requires 341 seconds and about 441 MB of space. Further experiments show that SPIN achieves a speedup of about factor 3 in comparison to HSF-SPIN.

6 Safety Property Validation Experiments

The experiments have been performed with SPIN version 3.3.10 and HSF-SPIN version 1.0 and were executed on a SUN workstation, UltraSPARC-II CPU with 248 Mhz under Solaris 5.7. If nothing else is stated the depth bound is set to 10,000 and no compression technique is used. In the case of deadlock detection in HSF-SPIN, H_{ap} is the estimation function used, unless indicated otherwise. In all other cases the formula based heuristic H_f is used. When comparing to SPIN it should be noted that this model checker was invoked with partial order reduction enabled.

6.1 Shorter Trails and Computational Effort

The first set of experiments is intended to show that A* always finds shorter trails compared to DFS while requiring less computational effort than BFS, and that in some cases A* performs better than DFS. By computational effort

GIOP	BFS	DFS	A*	BF	SPIN
s	40,847	218	31,066	117	326
e	37,266	218	27,061	65	326
t	151,671	327	108,971	126	364
l	58	134	58	65	134
Philosophers	BFS	DFS	A*	BF	SPIN
s	3,678	1,341	67	493	1,341
e	2,875	1,341	17	225	1,341
t	15,775	1,772	73	622	1,772
l	18	1,362	34	66	1,362
Optical	BFS	DFS	A*	BF	SPIN
s	148,591	20	83	83	20
e	110,722	20	14	14	20
t	621,216	20	83	13	20
l	38	44	38	38	44
Marriers	BFS	DFS	A*	BF	SPIN
a	9,459	10,588	9,208	7,154	2,530
e	9,004	10,588	8,335	4,124	2,530
t	24,064	29,069	22,298	9,710	3,116
l	50	72	50	61	72

Table 2: Deadlock detection in various protocols.

we mean the conjunction of the number of states stored, the number of states expanded and the number of transition performed. An additional objective is to show that BF can require less computational effort than A*, but that BF often delivers sub-optimal solutions.

For each kind of safety error we use a representative set of protocol models. Deadlock detection is performed using the CORBA GIOP protocol [19] with a configuration of 2 clients and 1 server, an 8-philosophers configuration of the dining philosophers problem, a model of an optical telegraph protocol [17] with 6 stations and a model of a concurrent program that solves the stable marriage problem [25] with a configuration of 3 suitors. Assertion violation detection experiments are carried out with Lynch's protocol, with a model of a relay circuit and with a faulty solution for the mutual exclusion problem (mutex)[6]. Invariant violation is evaluated using the POTS telephony model [20][7] and using an elevator protocol[8]. For the POTS model, the invariant described in Section 2.4 was used. In the elevator model, the invariant was of the form $\Box(\neg \mathit{opened} \vee \mathit{stopped})$.

Tables 2, 3 and 4 depict the results of error detection in these protocols

[6]Available from `netlib.bell-labs.com/netlib/spin`

[7]The Promela sources and further information about these models can be obtained from `www.informatik.uni- freiburg.de/~lafuente/models/models.html`

[8]Derived from `www.inf.ethz.ch/personal/biere/teaching /mctools/elsim.html`

Relay	BFS	DFS	A*	BF	SPIN
s	905	342	738	162	342
e	707	342	663	48	342
t	2,701	718	2,262	263	870
l	12	190	12	28	190
Lynch	BFS	DFS	A*	BF	SPIN
s	80	48	73	49	46
e	77	48	70	46	46
t	94	49	87	59	48
l	29	46	29	29	46
Mutex	BFS	DFS	A*	BF	SPIN
s	363	202	38	39	202
e	344	202	21	24	202
t	688	363	42	48	363
l	15	54	15	15	54

Table 3: Detection of assertion violations in various protocols.

with various search strategies. For each protocol, the number of stored states (s), expanded states (e), transitions performed (t) and the length of the error trail (l) is shown. Similar to SPIN, we count a sequence of atomic steps as one unique transition. The number of expansion steps in SPIN is the number of stored states. Columns 2 to 5 correspond to different search strategies of HSF-SPIN, namely breadth-first search (BFS), depth-first search (DFS), A* and best-first search (BF). The last column corresponds to the exploration with SPIN's depth-first search (SPIN).

In all examples BFS and A* provide optimal counterexamples. Compared to BFS the A* algorithm requires less computational effort. The reduction in the number of expansions, states and transitions varies from example to

POTS	BFS	DFS	A*	BF	SPIN
s	24,546	-	6,654	781	148,049
e	17,632	-	3,657	209	148,049
t	99,125	-	18,742	1,067	425,597
l	81	-	81	83	2,765
Elevator	BFS	DFS	A*	BF	SPIN
s	38,662	279	38,598	2,753	292
e	38,564	279	38,506	2,297	292
t	160,364	356	160,208	5,960	348
l	203	510	203	421	510

Table 4: Detection of invariant violations in various protocols.

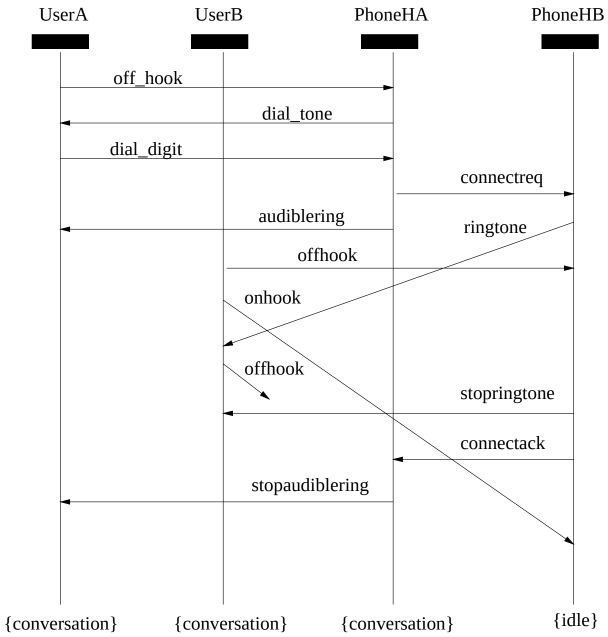

Figure 11: POTS example, error trail generated by HSF SPIN using A* and H_f.

example. This is mainly due to the quality of the heuristic estimate. For example, in the case of invariant violation detection for the elevator protocol, the savings in trail length achieved by A* are rather weak. This can be attributed to the integer range [0..2] of the heuristic estimation function which is very small considering that the optimal solution has 203 steps. On the other hand, while detecting the violation of the invariant of the POTS protocol the heuristic function returns estimates in the range [0..42]. With this range, the estimate function allows for a much more differentiated successor selection in A* which results in a much more informed search leading to a strong reduction in the computational effort required to detect the error. As can be expected, DFS finds error trails significantly larger than the optimal one(s). For example, the trail provided by SPIN's DFS for the invariant violation in the POTS protocol is about 20 times larger than the optimal trail generated by HSF-SPIN visualized in Figure 11. This trail is even superior to the manually generated short trail in Figure 4. However, HSF-SPIN happens to find a different target state than the one found by SPIN and this target state also corresponds to a different race condition than the one found by SPIN. Nevertheless, this race condition can also be traced back to a lack of synchronization between the `UserB` and `PhoneHB` processes. While in most cases DFS performs better than A* in terms of computational effort, in the philosophers problem and in the POTS protocol the performance of A* is superior to that of DFS. The reason for this lies in the particular structure of these problems. For both problems it is necessary that there is a sequence of actions in which every process performs one or a few steps in order to get closer to the target state. DFS, however, will try to first explore all possibilities for one process, before it includes the behavior of other processes. As a consequence, DFS will require more computational effort to reach a target state than A*. It should also be explained why HSF-SPIN runs out of memory in the POTS example, while the DFS in SPIN finds a counterexample. This is due to the fact that the implementation of DFS in SPIN is more efficient, and that we employed partial order reduction. Finally, the experiments highlight that although BF often requires less computational effort than A*, the established error trails are not optimal.

6.2 Heuristic Estimates

In the previous section we have noted the important influence of the heuristic estimate function on the performance of A*. Now we analyze different heuristic functions proposed for deadlock detection. In particular we compare the heuristic based on the number of active processes H_{ap} with formula based heuristic H_f combined with the proposed method for automatically inferring the deadlock formula f. With $H_f + U$ we denote that the user explicitly defines dangerous states. In the example we chose an "optimal" labeling, i.e., exactly those states are labeled as dangerous so that the resulting global control state is a deadlock state.

In our experiments we use the deadlock solution to the philosophers problem, the optical telegraph protocol, the marriers problem and the GIOP protocol. All models have been configured as in the previous set of experiments. Table 5 visualizes the number of expansions required to find the deadlock state and the range of values (r) that the heuristic estimate function is defined over. In all cases the optimal solution is being found.

The results show that when applying the inferred deadlock heuristic H_f user intervention improves the results in most cases. It is not easy to compare the inferred heuristic H_f with H_{ap}. H_{ap} seems to perform worse than $H_f + U$ except in the optical telegraph protocol. In the optical telegraph protocol the estimate H_{ap} works well, since the number of processes in the model is quite high. In the case of the GIOP protocol and the marriers model the number of processes is rather small and H_{ap} produces poor reductions in the number of expanded states. It should be emphasized that the quality of $H_f + U$ highly depends on the quality of the designers labeling of dangerous states. In summary, the experiments indicate the influence of the quality of the heuristic estimate function.

6.3 Finding Errors where DFS fails

A further objective of the *directed model checking* approach is to detect errors in models where classical depth-first exploration fails due to the exhaustion of memory resources.

Philosophers	no	H_{ap}	H_f	$H_f + U$
e	2,875	17	17	10
r	0..0	0..8	0..10	0..16
Optical	no	H_{ap}	H_f	$H_f + U$
e	110,722	14	342	342
r	0..0	0..12	0..14	0..12
Marriers	no	H_{ap}	H_f	$H_f + U$
e	493,840	432,483	462,235	192,902
r	0..0	0..4	0..25	0..25
GIOP	no	H_{ap}	H_f	$H_f + U$
e	37,266	27,061	28,067	24,859
r	0..0	0..6	0..12	0..25

Table 5: Deadlock detection with A* and different heuristic functions.

We perform a set of experiments with a scalable deadlock solution to the dining philosophers problem. We let the experiments run without time limitations, but with a hard memory constraint of 512 MB. Contrary to other experiments, we allow SPIN to apply bitstate hashing compression in order to emphasize the benefits of directed search.

Table 6 shows results on deadlock detection in the philosophers model with a growing number of philosophers. The first column depicts the number of philosophers in the model. The labeling of the other columns is obvious.

While A* and BF seem to scale linearly with respect to the increase of p, BFS and DFS do not. HSF-SPIN's BFS and DFS exploration are not able to find the deadlock situation in configurations with more than 13 philosophers. SPIN can go a little further, but fails in configurations with more than 15 philosophers. The results show that there are models in which A* is able to detect errors and in which depth-first search even if combined with reduction and compression techniques fails.

6.4 IDA* and Bitstate Hashing

We now show that for given memory and time constraints, IDA* in combination with bitstate hashing is able to detect errors in problems in which A* and IDA* fail. Once the priority queue is full, A* will run out of memory and once the transposition table is full, duplicate states will force IDA* to run out of time. We use the GIOP protocol with a deadlock error configured with 3 clients and one server as an experimental setup. We set the space limit to 256 MB and the time limit to 120 minutes. Both hash table sizes in A* and IDA* have been set to the given memory bound. Table 7 depicts the number of expansions performed by A*, IDA*, and IDA* combined with (double) bitstate hashing. To obtain the data in the table we modify A* to print a snapshot of the expansions every time the search depth increases, while for the last two methods, the number of

p		BFS	DFS	A*	BF	SPIN
2	s	9	6	6	6	6
	e	7	6	6	4	6
	t	10	7	4	6	7
	l	10	10	10	10	10
3	s	19	19	12	26	19
	e	14	10	7	23	10
	t	29	12	13	43	12
	l	14	18	14	14	18
4	s	56	45	19	70	45
	e	42	45	9	57	45
	t	116	62	21	142	116
	l	18	54	18	26	54
8	s	3,768	1,341	67	493	1,341
	e	2,875	1,341	17	225	1,341
	t	15,775	1,772	73	622	1,772
	l	34	1,362	34	66	1,362
14	s	-	-	199	1,660	2,164,280
	e	-	-	29	1,963	2,164,280
	t	-	-	211	684	27,050,400
	l	-	-	58	114	9,998
16	s	-	-	259	2,201	-
	e	-	-	33	893	-
	t	-	-	273	2,578	-
	l	-	-	66	130	-

Table 6: Deadlock detection in the dining philosophers problem.

Depth	A*	IDA*	IDA*+bitstate
58	150,344	146,625	146,625
59	168,191	164,982	164,982
60	184,872	184,383	184,383
61	-	206,145	206,145
62	-	-	229,626
63	-	-	255,411
64	-	-	282,444
65	-	-	311,340
66	-	-	341,562
67	-	-	373,422
68	-	-	407,310
69	-	-	442,941
70	-	-	goal found

Table 7: Deadlock detection in the GIOP protocol under memory constraints.

state expansions corresponds to the number of nodes in the current iteration. The results show that only the combination of IDA* and bitstate hashing is able to find the error in the protocol. IDA* exceeds the time and A* exceeds the space limit.

A duplicate is a state with different generating paths. Duplicates occur frequently in typical protocols. As long as the visited lists of A* and IDA* are not full, all duplicate states are detected. When the memory bound is reached, A* aborts since it is unable to allocate further states for the open and closed lists. IDA* bypasses the problem by exploring the tree expansion of the underlying graph and possibly re-exploring state space subtree structures. In some cases, there are too many duplicates such that after the transposition table is full and IDA* fails to complete the next iteration within the given time limit. However, IDA* with bitstate hashing prunes off duplicates optimistically, storing only a finger print (signature) of each state. This reduces the space requirements by some orders of magnitudes (about 3 in the example case), so that duplicates can be detected even in large search depths. The loss of states by false positives is marginal: in the example no state is wrongly identified with double bitstate hashing until the depth is reached in which IDA* gives up.

7 Liveness Property Validation

One feature of the Nested-DFS algorithm described in Section 2 is that a state, once *flagged* will not be expanded again during the cycle detection. For the correctness of the algorithm the post-order traversal of the search tree is crucial, such that the second depth-first traversal only encounters nodes that have already been visited in the main search routine. The second search can be improved by directed cycle detection search. Since we are aiming for states that have been placed on the Nested-DFS stack by the first traversal we can use heuristics to perform a directed search for the cycle-closing states. Unfortunately, in each of our benchmark examples, there is at most one accepting cycle, so that there is nothing to improve. The disadvantage of a pre-ordered nested search approach (search the acceptance state in the never claim and, once encountered, search for a cycle) is its quadratic worst-case time and linear memory overhead, since the second search has to be invoked with a newly initialized list of flagged states. To address this drawback, we developed an improvement of the nested depth-first search algorithm that exploits the structure of the never claim. This algorithm is applicable to a large set of practical property specifications, and can be combined with heuristic techniques for more efficient search performance.

7.1 Classification of Never Claims

Strongly connected components (SCC) partition a directed graph into groups such that there is no cycle combining two components. A subset of nodes in a directed graph is strongly connected if for all nodes u and v there is a path from u to v and a path from v to u. SCCs are maximal in this sense and can be computed in linear time [6].

To illustrate how SCCs can help improve the Nested-DFS algorithm, consider the never claim of Figure 1. We find two strongly connected components: the first is formed by n_0 and the second by n_a. Furthermore, there is no path from the second SCC to the first. Accepting cycles in the synchronous product automaton are composed of states in which the never claim is always in the second SCC. We can conclude that if the local never claim state corresponding to a cycle-closing synchronous product automaton state belongs to the second SCC the cycle is accepting. If, however, it belongs to the first SCC, it is not accepting.

In order to generalize this observation suppose that we have pre-computed all SCCs of a given never claim. Due to the synchronicity of the product of the model automaton and the never claim a cycle in the synchronous product is generated by a cycle in exactly one SCC. If the cycle is accepting, so is the corresponding cycle in the SCC of the never claim. Suppose that each SCC is either composed only of non-accepting states or only of accepting states. Then global accepting cycles only contain accepting states, while non-accepting cycles only contain non-accepting states. Therefore, a single depth-first search can be used to detect accepting cycles: if a state s is found in the stack, then the established cycle is accepting if and only if s itself is accepting.

The partitioning rules for SCCs given above can be relaxed according to the following classification of SCCs:

- We call an SCC *accepting* if at least one of its states is accepting, and *non-accepting* (N) otherwise.

- We call an accepting SCC *fully accepting* (F) if all of its cycles contain at least one accepting state.

	G	B	A	B	U
A	S,N	S,N	S,N	S,N	S,N,P
U	S,N	S,N	S,N	S,N,P,F	S,N,P
E	F	S,P,N	N,F	S,N,P	S,N,F
R	N,F	S,N,P,F	N,F	S,N,P,F	S,N,P,F
P	S,N,P	S,N	N,P	S,N	S,N,P

Table 8: SCC classification for LTL specification ppatterns.

- We call an accepting SCC *partially accepting* (P) if there is at least one cycle that does not contain an accepting state.

If the never claim contains no partially accepting SCC, then accepting cycle detection for the global state space can be performed by a single depth-first search: if a state is found in the stack, then it is accepting, if the never state belong to an accepting SCC. A special case occurs if the never claim has an endstate. When this state is reached, the never claim is said to be violated and a *bad* sequence has been found. Bad sequences are tackled similarly to safety properties by standard heuristic search.

The classification of patterns in property specifications proposed in [8] reveals that an empirically collected database of 555 practically used LTL properties partitions into *Absence (A)* (85/555), *Universality (U)* (119/555), *Existence (E)* (27/555), *Response (R)* (245/555), *Precedence (P)* (26/555), and *Others (53/555)*. Using this pattern scheme and the scope modifiers *Globally (G), Before (B), After (A), Between (B)*, and *Until (U)* we obtain a partitioning into SSCs according to Table 8. We indicate the presence of endstates with the letter S. Since the specification patterns are given using LTL formulae, we derive the equivalent never claims using the SPIN built-in LTL to never claim converter. Our approach is particularly useful for never claims that only contain N and F components, as for instance the Response pattern with a global scope. Given the prevalence of the Response pattern we conclude that our improvement of the Nested-DFS algorithm is applicable to a large set of practical problems.

7.2 Improved Nested Depth-First Search

We now present the *Improved-Nested-DFS* (INDFS) algorithm that is based on the above ideas, given in Figure 13. In this Figure, SCC(s) is the SCC of state s, F-SCC(s) determines if the SCC of state s is of type F (fully accepting), P-SCC(s) determines if the SCC of the state is of type P (partially accepting) and *neverstate*(s) denotes the local state of the never claim in the global state s.

The algorithm finds acceptance cycles without nested search for all problems which partition into N- or F-components. Except for P-SCCs it avoids the post-order traversal. For P-SCCs we guarantee that the second cycle detection traversal is restricted to the strongly connected component of the seed. The

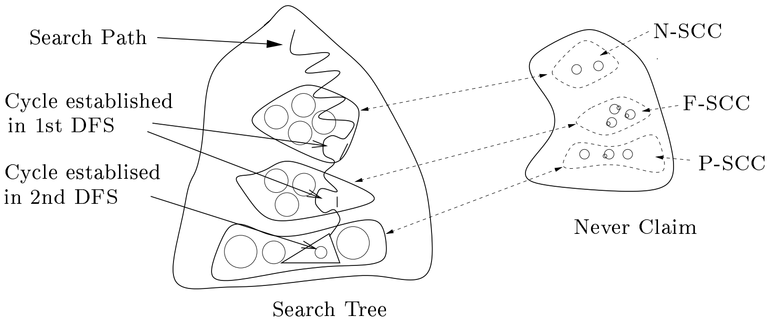

Figure 12: Visualization of the different cases in Improved-Nested-DFS.

algorithm considers the successors of a node in depth-first manner and marks all visited nodes with the label *hash*. If a successor s' is already contained in the stack, a cycle C is found. If C corresponds to a cycle in a F-SCC of the *neverstate* of s', it is an accepting one. Cycles for the P-SCCs parts in the never claim are found as in Nested-DFS, with the exception that the successors of a node are pruned which *neverstates* are outside the component. If an endstate in the never claim is reached the algorithm terminates immediately. A detailed proof of the correctness of INDFS is given in Section 7.4.

Figure 12 depicts the different cases of cycles detected in the search. The main idea for the correctness of Improved-Nested-DFS is based on the fact that all cycles in the state-transition graphs correspond to cycles in the never claim. Therefore, if there is no cycle combining two components in the never claim, so there is none in the overall search space.

As mentioned above, the strongly connected components can be computed in time linear to the size of the Never Claim, a number which is very small in practice. Partitioning the SCCs in *non-accepting*, *partially accepting* and *fully accepting* can also be achieved in linear time by a variant of Nested-DFS in the never claim.

7.3 A* and Improved-Nested-DFS

So far we have not considered heuristic search for Improved-Nested-DFS. Once more, we consider the example of *Response* properties to be validated. In a first phase, states are explored by A*. The heuristic estimation function can easily be designed to reach the accepting cycles in the SCCs faster, since all states that we are aiming at are accepting. This approach generalizes to a hybrid algorithm A* and Improved-Nested-DFS, *A*+INDFS* for short, that alternates between heuristic search in N-SCCs, single-pass searches in F-SCCs, and Nested-Search in P-SCCs. If a P- or S-component is encounterd, Improved Nested-DFS is invoked and searches for cycles. The heuristic estimate respects the combination of all F-SCCs and P-SCCs, since accepting cycles are present in either of the two components. The nodes at the horizon of a F- and P-

Improved-Nested-DFS(s)
hash(s)
for all successors s' of s **do**
  if s' in *Improved-Nested-DFS*-Stack **and**
    F-SCC(*neverstate*(s')) **then**
      exit *LTL-Property violated*
  if s' not in the hash table **then Nested-DFS**(s')
if *accept*(s) **and** *P-SCC*(*neverstate*(s)) **then**
  Improved-Detect-Cycle(s)

Improved-Detect-Cycle(s)
flag(s)
for all successors s' of s **do**
  if s' on $Improved-Nested-DFS$-Stack **then**
    exit *LTL-Property violated*
  else if s' not flagged **and**
    $SCC(neverstate(s)) = SCC(neverstate(s'))$ **then**
      Improved-Detect-Cycle(s')

Figure 13: Improved Nested Depth-First Search.

component lead to pruning of the sub-searches and are inserted back into the *Open*-List of A*, which contains all horizon nodes with a neverstate in the corresponding N-SCCs. Therefore *A* + INDFS* continues with directed search, if cycle detection in the F- and P-components fails. Cycle detection search itself can be accelerated with an estimation function heading back to the states where it was started.

Figure 14 visualizes this strategy for a response property. The never claim has the following SCCs: SCC_0 which is a N-SCC, and SCC_a which is F-SCC. The state space can be seen as divided in two partitions, each one composed of states where the never claim is a state belonging to one of the SCCs. In a first phase, A* is used for directing the search to states of the partition corresponding to SCC_a. Once a goal state is found, the second phase begins, where the search for accepting cycles is performed by Improved-Nested-DFS.

7.4 Correctness of INDFS

The nested depth-first search algorithm in model checking validates the emptiness of Büchi automata. It searches accepting cycles in the problem graph that represents the state-space of a Büchi automaton and reports non-emptiness if and only if there exists at least one accepting cycle in the graph. A correctness proof of this algorithm can be found in [4]. Our improvement to the nested depth-first search algorithm is depicted in Figure 13. To prove the correctness of the algorithm we start with some definitions.

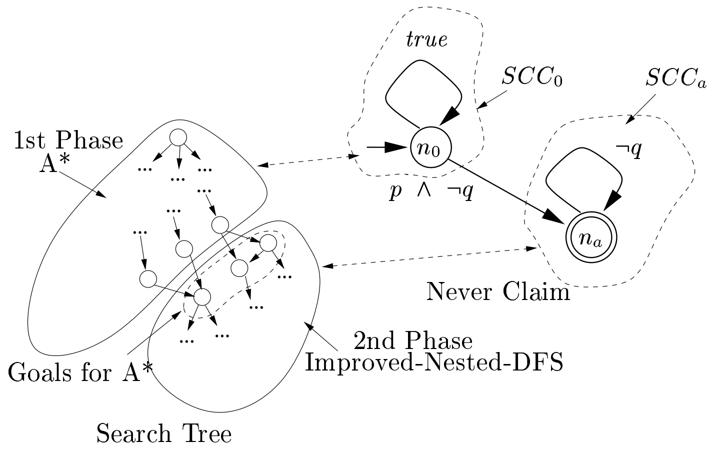

Figure 14: Visualization of A* and Improved-Nested-DFS for a response property.

Definition 1 *A Büchi automaton is a five tuple $\langle \Sigma, Q, \delta, Q_0, F \rangle$, where Σ is a finite alphabet, Q is the finite set of states, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $Q_0 \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of accepting states.*

Definition 2 *A* run *of a Büchi automaton over an infinite word $v \in \Sigma^*$ is a mapping $\rho : \{0, 1, .., \infty\} \mapsto Q$ such that a) the first state is an initial state, that is, $\rho(0) \in Q_0$, and b) moving from the i-th state $\rho(i)$ to the $(i+1)$-st state $\rho(i+1)$ upon reading the i-th input letter $v(i)$ is consistent with the transition relation, that is, for all $i \geq 0$ we have $(\rho(i), v(i), \rho(i)+1) \in \delta$.*

Definition 3 *Let* $\inf(\rho)$ *be the set of states that appear infinitely often in a run ρ (when treating the run as an infinite path). A run ρ of a Büchi automaton B over an infinite word is said to be* accepting *if and only if some accepting state appears infinitely often in ρ, that is, when $\inf(\rho) \cap F \neq \emptyset$. The* language $\mathcal{L}(B)$ accepted by the Büchi automaton B *is then the set of infinite words, over which all runs of B are accepting.*

Let M be a finite state automaton representing the model, and let N be the *never claim.* For this construction the automaton M is interpreted as a Büchi automaton in which *all* states are accepting.

Definition 4 *Let $M = \langle \Sigma, Q_M, \delta_M, Q_0^M, F_M \rangle$ be a Büchi automaton which states are all accepting, that is $Q_M = F_M$, and let $N = \langle \Sigma, Q_N, \delta_N, Q_0^N, F_N \rangle$ be another Büchi automaton. The* synchronous product *$M \otimes N$ of M and N is defined as: $M \otimes N = \langle \Sigma, Q, \delta, Q_0, F \rangle$, where $Q = Q_M \times Q_N$, $Q_0 = Q_0^M \times Q_0^N$, $F = F_M \times F_N = Q_M \times F_N$, and $((s_M, s_N), a, (s'_M, s'_N)) \in \delta'$ if and only if $(s_M, a, s'_M) \in \delta_M$ and $(s_N, a, s'_N) \in \delta_N$.*

Büchi automata can be represented as directed graphs: the set of vertices is Q and the edges are labeled by the transition relation δ. Runs of the automaton over an infinite word correspond to infinite paths in the graph, and accepting runs to infinite paths containing infinite accepting cycles. An accepting cycle is defined as a cycle in which at least one state is accepting.

Definition 5 *A strongly connected component (SCC) of a directed graph is a maximal set of vertices, such that each vertex in the set is reachable from each other vertex of the set.*

It is not difficult to show that pairwise reachability is an equivalence relation such that the set of nodes can be partitioned into equivalence classes of strongly connected components. An important consequence of the definition of SCCs is that all vertices of a cycle belong to the same SCC. In the following we write $scc(s)$ to denote the SCC to which a state s belongs.

Let Q be the set of states of $M \otimes N$. We define a partition function π from Q onto $\{0, \ldots, k\}$ in such a manner that two states belong to the same partition if and only if the state component of N belongs to the same SCC in the state transition graph of N. More precisely, if $s = (s_M, s_N)$ and $i = scc(s_N)$ then $\pi((s_M, s_N)) = i$. Obviously, π defines a partition of equivalence classes $P_0, \ldots P_k$ of Q, where $P_i = \{s \in Q \mid \pi(s) = i\}$, $i \in \{0, \ldots, k\}$.

Definition 6 *A strongly connected component is called* non-accepting *if none of its states is accepting,* full-accepting *each cycle formed by states of the SCC is accepting, and* partial-accepting *otherwise.*

Definition 7 *An equivalence class P_i of $M \otimes N$ is* non-accepting, full-accepting *or* partial-accepting *if the corresponding strongly connected component i in N is* non-accepting, full-accepting *or* partial-accepting*, respectively.*

Lemma 1 *If there is a cycle C in Q, then π partitions the states in Q in such a manner that all states of the cycle belong to the same equivalence class in Q, i.e., $C \subseteq P_i$ for one $i \in \{0, \ldots, k\}$.*

Proof: Let C be a cycle in state transition graph of Q, that is $C = (s_0, s_1, \ldots, s_n)$ with $s_n = s_0$ and $(s_i, a, s_{i+1}) \in \delta$ for all $i \in \{0, \ldots, n-1\}$. Therefore, since $s_i = (s_i^M, s_i^N)$ and $s_i^N \in N$, $i \in \{0, \ldots, n\}$, a cycle $C_N = (s_0^N, s_1^N, .., s_n^N = s_0^N)$ exists with $(s_i^N, a, s_{i+1}^N) \in \delta_N$ for all $i \in \{0, \ldots, n\}$. Hence, for all $s_i = (s_i^M, s_i^N)$ and $s_j = (s_j^M, s_j^N)$ in C we have $scc(s_i^N) = scc(s_j^N)$. This implies $\pi(s_i) = \pi(s_j)$ for all $s_i, s_j \in C$ such that all states of C belong to the same equivalence class.

■

Lemma 2 *A cycle C in $M \otimes N$ is accepting if and only if the corresponding cycle in N is accepting.*

This is easy to see, since as defined, a state $s = (s_M, s_N)$ of $M \otimes N$ is accepting if and only if s_N is an accepting state of N.

Lemma 3 *All cycles in a* non-accepting *component are non-accepting, all cycles in a* fully-accepting *component are accepting. In* partial-accepting *components, there can be accepting and non-accepting cycles.*

Lemma 3 is inmediately deduced from Definitions 6 and 7, and Lemma 2.

The following lemma is a well-known property of depth-first search and is essential for proving the correctness of our algorithm.

Lemma 4 *Let s be a vertex that does not appear on any cycle. Then the depth first search algorithm will backtrack from s only after all the nodes that are reachable from s have been explored and backtracked from.*

It is easy to see that this lemma still holds for the first search in both the original and in the improved nested depth first search algorithm.

Theorem 1 *The improved nested depth first search algorithm returns a counterexample for the emptiness of the checked automaton $M \otimes N$ exactly when $\mathcal{L}(M \otimes N)$ is not empty.*

Proof: We have to show I) that a counterexample returned by the algorithm corresponds in fact to an accepting run of the automaton, and II) that no accepting run is missed by the algorithm.

I) The first thing to show is that if the algorithm finds an accepting cycle, then the cycle is in fact accepting. The algorithm closes accepting cycles in the first and in the second search. When the algorithm closes cycles in the second search, it acts like the original algorithm. As shown in [4], cycles closed in the second search are accepting, since the second search is started from accepting states only. On the other hand cycles closed in the first search correspond only to cycles present in a *full-accepting* equivalence class, and as shown in Lemma 2, every cycle in a *full-accepting* component is accepting.

II) The difficult case is to prove that if the algorithm finds no accepting cycle then $\mathcal{L}(M \otimes N)$ is in fact empty. As shown above, accepting cycles can exist only in *full-accepting* component or in *partial-accepting* component. There are two cases if the algorithm fails to find an existing accepting cycle: II_a) the cycle exists in a *full-accepting* component and is missed in the first search, or II_b) the cycle exists in a *partial-accepting* component and is missed in the second search.

II_a) Suppose that an accepting cycle exists in a *full-accepting* component and that the first search fails to find it. Let s be the first state visited by the depth first search that is reachable from itself and that belongs to a *full-accepting* component. The first search misses that cycle if in the moment in which the search is started from s, every path from s to itself contains a an already visited state. Let s' be the first such state. Then s' was visited by the depth-first search before s and is reachable from itself through the cycle $(s' \to \ldots \to s \to \ldots \to s')$, and s' belong to the same *full-accepting* component by Lemma 1, which contradicts our assumption.

II_b) Suppose now that an accepting cycle exists in a *partial-accepting* component and that the second search fails to find it. In this case a similar reasoning as in [4] can be done to show that this cannot be happen. Let s be the first accepting state belonging to a *partial-accepting* component from which the second search starts but fails to find a cycle even though one exists. In the moment in which the second search starts from s there is at least one flagged state on a cycle through s. Let r be the first such state, and let s' be the state from which the second search that flagged r was started. In the algorithm the second search remains in the same equivalence class from which the search was started. Therefore s, r and s' must belong to the same component. According to our assumptions, the second search from s' was started before the second search from s. There are two cases:

II'_b: The state s' is reachable from s. Then there is a cycle ($s' \rightarrow \ldots \rightarrow r \rightarrow \ldots \rightarrow s \rightarrow \ldots \rightarrow s'$) that could not have been found previously, otherwise the algorithm would already have terminated. By Lemma 3, the cycle belongs to a *partial-accepting* component. However this contradicts our assumption from which the second search missed a cycle belonging to a *partial-accepting* component.

II''_b: The state s' is not reachable from s. If s' appears on a cycle, then a cycle was missed before starting the second search at s and the cycle belongs to a *partial-accepting* component, since s and s' belong to the same component. According to the assumption, s is reachable from r and, subsequently, s is reachable from s'. Thus, if s' does not occur on a cycle, by Lemma 4 we must have discovered and backtracked from s in the first search before backtracking from s'. Hence, according to the algorithm, we must have started a second search from s before starting it from s'. This contradicts the assumptions. ∎

8 Liveness Property Experiments

In this Section we describe experimental results in the validation of liveness properties. The experimental setup is largely as described in Section 6. We compare two algorithms in this section, namely NDFS and INDFS. Both algorithms have been implemented in HSF-SPIN. We have also implemented INDFS inside SPIN, so that both tools have the same algorithmic capabilities. Since the results produced by both tools are very similar in terms of computational effort, we only give the values obtained by HSF-SPIN in this Section.

8.1 INDFS for Validating Correctness

This first set of experiments is intended to show the benefits of INDFS when validating liveness properties. In the worst case, INDFS performs as many transitions and expansions as NDFS, while in a best case situation INDFS can halve these values. The worst case occurs when the never claim contains no F-SCCs, while the best case occurs when the never claim contains exclusively

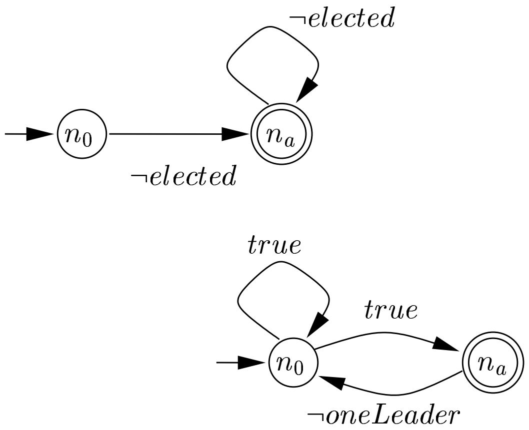

Figure 15: Never claims for $\Diamond\Box oneLeader$ (top-left) and $\Diamond elected$ (bottom-right).

$\Diamond\Box oneLeader$	NDFS	INDFS
s	4,779	4,779
e	9,556	9,556
t	42,307	42,307
$\Diamond elected$	NDFS	INDFS
s	2,380	2,380
e	14,086	7,044
t	4,759	2,380

Table 9: Checking correctness of two liveness properties in the leader election algorithm with NDFS and INDFS.

SCCs of this type. Note that all never claims are generated using SPIN's LTL-to-never-claim translation. We use a model of the leader election algorithm as test case. As a worst case we check the property $\Diamond\Box oneLeader$ for which the corresponding never claim is formed by a unique P-SCC. For the best case situation we used the property $\Diamond elected$ for which the corresponding never claim is formed by a unique F-SCC. Figure 15 illustrates the never claims that SPIN generates for each property.

Table 9 depicts the results of the experiments. The number of transitions and expansions is shown. The number of stored states is also included in the table to show that both algorithms explore exactly the same number of states. The results show that in the worst case situation both NDFS and INDFS perform the same. On the other hand, in the best case situation INDFS requires about half of the transitions and expansions that NDFS requires.

Elevator	NDFS	INDFS	A*+INDFS
s	192	187	29,407
e	229	187	28,309
t	280	215	130,211
l	320	311	297
GIOP	NDFS	INDFS	A*+INDFS
s	7,331	7,260	86
e	7,346	7,260	81
t	33,061	32,984	93
l	289	155	155

Table 10: Detection of violation of liveness properties in two protocols.

8.2 INDFS for Error Detection

Our objective now is to show that INDFS requires less computational effort and provides better error trails than NDFS. We also test the performance of the hybrid algorithm that combines NDFS with A*.

We first use a version of the GIOP model configured with 1 server and 3 clients and with a seeded error that causes the model to violates a response property stating that when a client sends a request, a reply will always be received. Second, we use a model of an elevator with 3 floors that violates the response property stating that whenever a request for the elevator exists in one level, the elevator will eventually stop at that level and open the door. Table 10 shows experimental results on detecting the violation of LTL formulae. NDFS is compared with INDFS and the algorithm that combines A* and NDFS (A*+INDFS).

The results show that INDFS provides small improvements over NDFS in all categories. However, only for the GIOP protocol the reduction is significant, INDFS almost halves the length of the error trail. The hybrid algorithm finds better solutions in all situations, but its computational effort varies drastically. While in the elevator experiment it requires about 15 times more state expansions than INDFS, in the GIOP experiment it performs 89 times less. The reason of this varying behavior is that A*+NDFS directs the search to the nearest full accepting component of the state space. This component may, however, be free of cycles. Only after this part of the state space is entirely explored the nested search returns control to A* which then directs the search into the next full accepting part. While in the case of the GIOP protocol the algorithm finds a component with a cycle early on in the search, in the elevator example the algorithm first explores parts of the state space that include accepting states, but no accepting cycles.

9 Related Work

In earlier work on the use of directed search in model checking the authors apply best-first exploration to protocol validation [22]. They are interested in typical safety properties of protocols, namely unspecified reception, absence of deadlock and absence of channel overflow. In the heuristics they use an estimate determined by identifying *send* and *receive* operations. In the analysis of the X.21 protocol they obtained savings in the number of expansion steps of about a factor of 30 in comparison with a typical depth first search strategy. We have incorporated this strategy in HSF-SPIN. The approach in [22] is limited to the detection of deadlocks, channel overflows and unspecified reception in protocols with asynchronous communication. To the contrary our approach is more general and handles a larger range of errors and communication types. While the measures in [22] are merely stochastic and will not yield optimal solutions, the heuristics we propose are in most cases lower bound estimators and hence allow us to find optimal solutions.

The authors in [5] identify three phases in the verification process. In the *exploratory mode* the system designer tries to find a first error, in the *fault finding mode* s/he aims at meaningful counterexamples, while in the *maintenance mode* one does not expect errors at all. From this point of view, our approach concentrates on the first two modes. Moreover, the authors in [5] analyze which algorithm is best-suited for which mode. They use different variants of depth-first search, breadth-first search and A*. Some of the ideas for the heuristic estimates are similar to ours, but the authors do not elaborate on the specific heuristic estimates that they use. Contrary to us, they do not consider IDA* and restrict their work to safety properties. Our conclusions are slightly different. We agree that a shortest path algorithm is suitable for the fault finding mode, but we believe that directed search can also be useful in the first exploratory mode: even in this phase by guiding the search an error state can be found with less computational effort than with blind search strategies.

The authors of [30] use BDD-based symbolic search within the Murϕ validation tool. The best first search procedure they propose incorporates symbolic information based on the Hamming distance of two states. This approach has been improved in [27], where a BDD-based version of the A* algorithm [12] for the μcke model checker [1] is presented. The algorithm outperforms symbolic breadth-first search exploration for two scalable hardware circuits. The heuristic is determined in a static analysis prior to the search taking the actual circuit layout and the failure formula into account. The approach to symbolic guided search in CTL model checking documented in [2] applies 'hints' to avoid sections of the search space that are difficult to represent for BDDs. This permits splitting the fix-point iteration process used in symbolic exploration into two parts yielding under- and over-approximation of the transition relation, respectively. Benefits of this approach are simplification of the transition relation, avoidance of BDD blowup and a reduced amount of exploration for complicated systems. However, in contrast to our approach providing 'hints' requires user intervention. Also, this approach is not directly applicable to explicit state model checking, which is our focus.

Exploiting structural properties of the Büchi Automaton in explicit state mode checking has been considered in the literature in the context of weak alternating automata (WAA) [26]. WAA were invented to reason about temporal logics, generalize the transition function with boolean expressions of the successor set, and partition the automaton structure. The classification of the states of a WAA differs from ours, since the partitioning into disjoint sets of states that are either all accepting or all rejecting does not imply our partitioning. The simplification of Büchi automata proposed in [28] is inferred from an LTL property, whereas our INDFs algorithm is based on the analysis of the structure of Büchi automata. The work in [28] also considers a partitioning according to WAA-type weakness conditions and hence differs from the approach taken in our paper.

The approach taken in [29] addresses explicit CTL* model checking in SPIN using hesitant alternating automata (HAAs). The paper shows that the performance of the proposed 'LTL nonemptiness game' is in fact a reformulation and improvement of nested depth-first search. Both the partitioning and the context of HAA model checking are significantly different from our setting.

10 Conclusion und Outlook

We argued that in order to facilitate debugging the error trails or witnesses that a model checker generates minimizing their length is highly desirable. A reduction in the number of visited states during state space search is also desirable since this renders larger models tractable. Standard depth-first search algorithms used in explicit state model checkers like SPIN are rather efficient in terms of memory usage and computing time, but tend to produce lengthy counterexamples.

We introduced into heuristic search algorithms, and showed how to apply heuristic search to safety property validation. The experimental results showed that directed model checking with A* always returns shorter error trails than DFS, and that in most instances the trail length is optimal. Regarding computational effort the results were mixed: in some instances A* was superior to SPIN and DFS, but in many cases A* was not performing as well. It also became clear that the gain obtained through directed model checking is better the more differentiation the heuristic estimation function allows. We also observed that for the dining philosophers problem under constrained memory availability directed model checking was able to solve a problem that could not be solved by DFS. We expect that this effect is linked to the highly symmetric nature of the problem, and the high degree of coordination that is typical for this example.

Next we proposed an improvement of the nested depth-first search algorithm that exploits the structure of the never claim to be validated. The INDFS algorithms is applicable to the validation of liveness properties. We showed that INDFS, which is not a directed model checking algorithm, leads to modest improvements in terms of error trail length compared to NDFS. In further experiments we showed how the combined usage of A* and NDFS can lead to significant reductions in error trail length.

The incorporation of heuristic search strategies is based on the observation that standard state space exploration algorithms perform a search that is rather uninformed of the structure of the search problem. As raw as the heuristics that we propose may be, it is surprising to see them work rather well on many problems. We are not primarily interested in optimal solutions, which is why we can tolerate non-admissible heuristic estimates when better estimates are not available.

In concurrent work [11] we describe an approach to shorten existing error trails using refined state distance metrics as heuristic estimates. This approach has already been implemented in HSF-SPIN. For selected benchmark and industrial communication protocols experimental evidence is given that *trail-directed model checking* effectively shortcuts existing witness paths.

We are nevertheless interested in improving the quality of the heuristics so that our approach becomes applicable to an even larger set of problems. One approach assesses the fact that our assumption of independence in combining sub-formulae is rarely fulfilled in practice. The main idea in the not yet implemented approach of *directed stochastic model checking* is to derive a stochastic model for search prediction that takes correlations of propositions into consideration in order to direct the search.

Further unpublished work reconciles partial order reduction techniques with the directed model checking approach of HSF-SPIN. Both theoretically and empirically we show that A* and IDA* can be combined with partial order reduction methods. While the benefit of the application of partial order reduction to A* is limited, due to its similarity to DFS IDA* avails itself rather nicely to partial order reduction.

References

[1] Armin Biere. μcke - efficient μ-calculus model checking. In *Computer Aided Verification*, pages 468–471, 1997.

[2] R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Conference on Design Automation (DAC-00)*, pages 29–34. ACM/IEEE, 2000.

[3] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, Apr 1983.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[5] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite s tate verification. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 37–46. IEEE Computer Society, 2001.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[7] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*, 1999.

[9] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proc. of the 8th International SPIN Workshop on Model Checking Software*, Lecture Notes in Computer Science. Springer Verlag, May 2001.

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

[11] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Trail-directed model checking. In *Proc. of the Workshop on Software Model Checking*, Electrical Notes in Theoretical Computer Science. Elsevier, July 2001.

[12] S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, pages 81–92, 1998.

[13] S. Edelkamp and S. Schrödl. Localizing A*. In *National Conference on Artificial Intelligence (AAAI)*, pages 885–890, 2000.

[14] Stefan Edelkamp. *Data Structures and Learning Algorithms in State Space Search*. PhD thesis, University of Freiburg, 1999. Infix.

[15] M. G. Gouda. Protocol verification made simple: a tutorial. *Computer Networks and ISDN Systems*, 25(9):969–980, 1993.

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

[17] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[18] G. J. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[19] M. Kamel and S. Leue. Formalization and validation of the general inter-orb protocol (GIOP) using Promela and SPIN. In *Software Tools for Technology Transfer*, volume 2, pages 394–409, 2000.

[20] Moataz Kamel and Stefan Leue. Vip: A visual editor and compiler for v-promela. In *6th International Conference, TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 471–486. Springer, 2000.

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

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

[23] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[24] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.

[25] D.G. McVitie and L.B. Wilson. The stable marriage problem. *Communications of the ACM*, 1971.

[26] D. E. Muller, A. Saoudi, and P. E. Schnupp. Alternating automata. The weak monadic theory of the tree, and its complexity. In Laurent Kott, editor, *International Colloquium on Automata, Languages and Programming*, pages 275–283. Springer, 1986.

[27] F. Reffel and S. Edelkamp. Error detection with directed symbolic model checking. In *World Congress on Formal Methods*, pages 195–211. Springer, 1999.

[28] F. Somenzi and R. Bloem. Efficient buchi automata from LTL formulae. In *Computer Aided Verification*, 2000.

[29] W. Visser and H. Barringer. CTL* model checking for SPIN. *Software Tools for Technology Transfer*, 2(4), 2000.

[30] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *DAC*, pages 599–604, 1998.