Directed Explicit Model Checking with HSF-SPIN

Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue

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

Abstract. We present the explicit state model checker HSF-SPIN which is based on the model checker SPIN and its Promela modeling language. HSF-SPIN incorporates directed search algorithms for checking safety and a large class of LTL-specified liveness properties. We start off from the A* algorithm and define heuristics to accelerate the search into the direction of a specified failure situation. Next we propose an improved nested depth-first search algorithm that exploits the structure of Promela Never-Claims. As a result of both improvements, counterexamples will be shorter and the explored part of the state space will be smaller than with classical approaches, allowing to analyze larger state spaces. We evaluate the impact of the new heuristics and algorithms on a set of protocol models, some of which are real-world industrial protocols.

1 Introduction

Model Checking [3] is a formal analysis technique that has been developed to automatically validate 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 [21] 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 [2] and [12]. 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). For the description of the state machine model
we use the language Promela [17], and for the validation of Promela models we
use the model checker SPIN\(^1\) [16].

The use of model checking in system design has the great advantage over the
use of deductive formal verification techniques that 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 store it 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
addressed. First, it is desirable to make sure that the length of a search until
a property violation is found 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 an
approach to Directed Model Checking in our paper.

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 val-
idation 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 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**
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. We improve the heuristic estimate by taking the structure of the temporal property into account.

**Related Work.** In earlier work on the use of directed search in model checking the authors apply best-first exploration to protocol validation [20]. They are interested in typical safety properties of protocols, namely unspecified reception, absence of deadlock and absence of channel overflow. In the heuristics they therefore 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. While the approach in [20] is limited to the detection of deadlocks, channel overflows and unspecified reception in protocols with asynchronous communication, the approach in this paper is more general and handles a larger range of errors and communication types. While the labelings used in [20] are merely stochastic measures that will not yield optimal solutions, the heuristics we propose are lower bound estimators and hence allow us to find optimal solutions.

The authors of [30] use BDD-based symbolic search within the Mur$\phi$ 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 [26], where a BDD-based version of the A$^*$ algorithm [11] for the $\mu$cke 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 [25] 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 overapproximation 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) [5]. 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 [27] is inferred from an LTL property, whereas we work on the basis of Büchi automata. This work 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.

In our paper we will use a number of protocols as benchmarks. These include Lynch’s protocol, the alternating bit protocol, Barlett’s protocol, an erroneous solution for mutual exclusion (mutex)\(^2\), the optical telegraph protocol [17], an elevator model\(^3\), a deadlock solution to Dijkstra’s dining philosopher problem, and a model of a concurrent program that solves the stable marriage problem [22]. Real-World examples that we use include the Basic Call processing protocol [23], a model of a relay circuit [28], the Group Address Registration Protocol GARP [24], the CORBA GIOP protocol [18], and the telephony model POTS [19]\(^4\).

Precursory Work. The precursor [10] to this paper considers safety property analysis for simple protocols. In the current paper we extend on this work by refining the safety heuristics, by providing an approach to validating LTL-specified safety properties, and by experimenting with a larger set of protocols.

Structure of Paper. In Section 2 we review automata-based model checking. Section 3 discusses the analysis of safety properties in directed model checking and describes the use of the A* algorithm for this purpose. In Section 4 we discuss liveness property analysis. We present approaches to improve search strategies for validation of LTL properties. In Section 5 we discuss how to devise informative heuristic estimates in communication protocols. The new protocol validator HSF-SPIN is presented in Section 6. Experimental results for various protocols are discussed in Section 7. We conclude in Section 8.

2 Automata-Based Model Checking

In this Section we review the automata theoretic framework for explicit state model checking. Since we model infinite behaviors the appropriate formalization for words on the alphabet of transitions sequences are Büchi-Automata. They inherit the structure of finite state automata but with a different acceptance condition. A run (infinite path) in a Büchi-Automaton is accepting if the set of states that appear infinitely often in the run has a non-empty intersection with the set of accepting states. The language \(L(A)\) of a Büchi-Automaton \(A\) consists of all accepting runs. The expressiveness of Büchi-Automata includes LTL.

\(^2\) Available from netlib.bell-labs.com/netlib/spin
\(^3\) Available from www.inf.ethz.ch/personal/biere/teaching/mctools/elsim.html
\(^4\) The Promela sources and further information about these models can be obtained from www.informatik.uni-freiburg.de/~lafuente/models/models.html
Formally, LTL specification $F(M)$ according to a Kripke Model $M$ are defined as follows: All predicates $a$ are in $F(M)$ and if $f$ and $g$ are in $F(M)$, so are $\neg f, f \lor g, f \land g, X f, F f, G f, f U g$. In LTL, temporal modalities are expressed through the operators $\Box$ for globally ($G$) and $\Diamond$ for eventually ($F$).

In automata-based Model Checking we construct the Büchi-Automaton $A$ and the automaton $B$ that represents the system $M$. $A$ is sometimes obtained by translating an LTL formula into a Büchi Automaton. While this translation 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 system $B$ satisfies $A$ when $L(B) \subseteq L(A)$. This is equivalent to $L(B) \cap \overline{L(A)} = \emptyset$, where $\overline{L(A)}$ denotes the complement of $L(A)$. Note that Büchi-Automata are closed under complementation. In practice, $\overline{L(A)}$ can be computed more efficiently by deriving a Büchi-Automaton from the negated formula. Therefore, in the SPIN validation tool LTL formulae are first negated, and then translated into a Never Claim (automaton) that represent the negated formula. As an example we consider the commonly used response property which states that whenever a certain request event $p$ occurs a response event $q$ will eventually occur. Response properties are specified in LTL as $\Box(p \rightarrow \Diamond q)$ and the negation is $\Diamond(\neg p \land \Box \neg q)$. The Büchi-Automaton and the corresponding Promela Never-Claim for the negated response property are illustrated in Figure 1.

![Büchi-Automaton for response property (left) and for its negation (right).](image)

The emptiness of $L(B) \cap \overline{L(A)}$ is determined using an on-the-fly algorithm based on the synchronous product of $A$ and $B$: Assume that $A$ is in state $s$ and $B$ is in state $t$. $B$ can perform a transition out of $t$ if $A$ has a successor state $s'$ of $s$ such that the label of the edge from $s$ to $s'$ represents a proposition satisfied in $t$. A run of the synchronous product is accepting if it contains at least one accepting state of $A$. $L(B) \cap \overline{L(A)}$ 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.
3 Searching for Safety Property Violations

The detection of a safety error consists of finding a state in which some property is violated. Typically, the algorithms used for this purpose are depth-first and breadth-first searches. Depth-first search is memory efficient, but not very fast in finding target states. We describe how heuristic search algorithms can be used instead in order to accelerate the exploration.

Heuristic search algorithms take additional search information in form of a evaluation function into account that returns a number purporting to describe 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 suffers from the same defects as depth-first search – it is not optimal and may be stuck in dead-ends or local minima.

Breadth-first search (BFS), on the other hand, is complete and optimal but very inefficient. Therefore, A* [13] combines both approaches for a new evaluation function by summing the generating path length $g(u)$ and the estimated cost of the cheapest path $h(u)$ to the goal yielding the estimated cost $f(u) = g(u) + h(u)$ of the cheapest solution through $u$. If $h(u)$ is a lower bound then A* is optimal. Table 1 depicts the implementation of A* to search safety violations, where $g(u)$ is the length of the traversed path to $u$ and $h(u)$ is the estimate from $u$ to a failure state.

\[
A^*(s) \\
\text{Open} \leftarrow \{(s, h(s))\}; \quad \text{Closed} \leftarrow \{} \\
\text{while} (\text{Open} \neq \{\}) \\
\quad u \leftarrow \text{Deletemin}(\text{Open}); \quad \text{Insert}(\text{Closed}, u) \\
\quad \text{if} (\text{failure}(u)) \text{ exit Safety Property Violated} \\
\quad \text{for all} \ v \ \text{in} \ P(u) \\
\quad \quad f'(v) \leftarrow f(u) + 1 + h(v) - h(u) \\
\quad \quad \text{if} (\text{Search}(\text{Open}, v)) \\
\quad \quad \quad \text{if} (f'(v) < f(v)) \\
\quad \quad \quad \quad \text{DecreaseKey}(\text{Open}, v, f'(v)) \\
\quad \quad \text{else if} (\text{Search}(\text{Closed}, v)) \\
\quad \quad \quad \text{if} (f'(v) < f(v)) \\
\quad \quad \quad \quad \text{Delete}(\text{Closed}, v); \quad \text{Insert}(\text{Open}, (v, f'(v))) \\
\quad \quad \text{else} \ \text{Insert}(\text{Open}, (v, f'(v))) \\
\]

Table 1. The A* Algorithm Searching for Violations of Safety Properties.

Similar to Dijkstra’s single source shortest path exploration [7], starting with the initial state, A* extracts states from the priority queue Open until a failure state is found. In a uniform-cost graph with integral lower-bound estimate the $f$-values are integer and bounded by a constant, such that the states can be kept in
doubly-linked lists stored in buckets according to their priorities [6]. Therefore, given a node reference Insert and Close can be executed in constant time while the operation DeleteMin increases the bucket index for the next node to be expanded. If the differences of the priorities of successive nodes are bounded by a constant, DeleteMin runs in O(1). 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 priority queue nodes (re-opening).

Figure 2 depicts the impact of heuristic search in a grid graph with all edge costs being 1. If \( h \equiv 0 \), A* reduces to Dijkstra’s algorithm, which in case of uniform graphs further collapses to BFS. Therefore, starting with \( s \) all nodes shown are added to the (priority) queue until the goal node \( t \) is expanded. If we use \( h(u) \) as the Euclidian distance \( \|u - t\|_2 \) to state \( t \), then only the nodes in the hatched region are ever removed from the priority queue.

![Diagram](image)

**Fig. 2.** The Effect of Heuristic Search in a Grid Graph.

Weightening scales the influence of the heuristic estimate such that the combined merit function \( f \) of the generating path length \( g \) and the heuristic estimate \( h \) is given by \( f(u) = \alpha g(u) + (1 - \alpha) h(u) \) for all states \( u \) and \( \alpha \in [0, 1] \). In case \( \alpha < 0.5 \), optimality of the search algorithms is affected, for \( \alpha = 0 \) we exibit BF, and for \( \alpha = 1 \) we simulate BFS.

## 4 Searching for Liveness Property Violations

Liveness properties refer to paths of the state transition graph and the detection of liveness property violations entails searching for cycles in the state graph. This is typically achieved by a nested depth-first search (Nested-DFS) that can be implemented with two stacks as shown in Figure 3 (cf. [3]).

One feature of this algorithm is that a state, once flagged will not be considered further on. For the correctness of the algorithm the post-order traversal of the search tree is crucial, such that the secondary depth-first traversal only encounters nodes that have already been visited in the main search routine. Therefore in the application of heuristic methods for the first traversal of Nested-DFS, we are restricted to move ordering techniques: using a heuristic function for establishing the order in which the successors of a state will be explored. However,
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($s'$)

Fig. 3. Nested-Depth-First-Search

the second search can be improved by directed cycle detection search. Since we are aiming for those states in the first stack we can use heuristics to perform a directed search for the cycle-closing states. 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 visited list. To address this drawback we developed a single pass DFS algorithm applicable to a large set of practical property specifications.

4.1 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 [4]. In the Never-Claim of the example in 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. Therefore, accepting cycles in the Never-Claim exist only in the second SCC. Accepting cycles in the synchronous product automaton are composed of states in which the Never-Claim is always in state $n_a$ (second SCC). A cycle is found if a state is encountered on the stack. Moreover, if the local state of the never claim in the found global state belongs to the first SCC, the established cycle is not accepting, and if it belongs to the second SCC it is an accepting one.

In order to generalize the 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. Moreover, 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 restriction on the SCC partitioning given by the above rules can be relaxed according to the following classification of the SCCs.

- We call an SCC \textit{accepting} if at least one of its states is accepting, and \textit{non-accepting} (N-SCC) otherwise.
- We call an accepting SCC \textit{fully accepting} (F-SCC) if all of its cycles contain at least one accepting state.
- We call an accepting SCC \textit{partially accepting} (P-SCC) if there is at least one cycle that does not contain an accepting state.

If the Never-Claim contains no partially accepting SCC, then acceptance 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. If this state is reached the never claim is said to be violated; a \textit{bad} sequence is found. We indicate the presence of endstates with the letter \( S \). Bad sequences are tackled similarly to safety properties by standard heuristic search.

The classification of patterns in property specifications [8] reveals that a database of 555 LTL properties partitions into \textit{Absence} (85/555), \textit{Universality} (119/555), \textit{Existence} (27/555), \textit{Response} (245/555), \textit{Precedence} (26/555), and \textit{Others} (53/555). Using this pattern scheme and the modifiers \textit{Globally}, \textit{Before}, \textit{After}, \textit{Between}, and \textit{Until} we obtain a partitioning into SCCs according to Table 2.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Globally</th>
<th>Before</th>
<th>After</th>
<th>Between</th>
<th>Until</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absence</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
</tr>
<tr>
<td>Universality</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
</tr>
<tr>
<td>Existence</td>
<td>( F )</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
</tr>
<tr>
<td>Response</td>
<td>( N+F )</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
</tr>
<tr>
<td>Precedence</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
<td>( S+N )</td>
</tr>
</tbody>
</table>

\textbf{Table 2.} SCC Classification for LTL-Specification Patterns. \( S \) indicates the presence of endstates in the never claim, while \( N, P, F \) indicate the presence of at least one N-SCC, P-SCC and F-SCC respectively.

### 4.2 Improved Nested Depth-First-Search

In this section we present an improvement of the Nested-DFS algorithm called \textit{Improved-Nested-DFS}. It finds acceptance cycles without nested search for all problems which partition into N- or F-components. The algorithm reduces the number of transitions required for full validation of liveness properties. 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 Improved-Nested-DFS algorithm is given in Fig. 5. 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 algorithms 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. Figure 4 depicts the different cases of cycles detected in the search. The correctness of Improved-Nested-DFS follows from 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.

![Search Path](image)

**Fig. 4.** Visualization of the Different Cases in Improved-Nested-DFS.

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. In contrast to the heuristic directed search the improved nested depth-first search algorithm accelerates the search for full validation. The ease of implementation suggests to Improved-Nested-DFS to the SPIN validation tool.

### 4.3 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 evaluation function to focus the
Improved-Nested-DFS($s$)
hash($s$)
for all successors $s'$ of $s$ do
if $s'$ in Improved-Nested-DFS-Stack and F-SCC(neverstate($s'$))
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($s'$)

Fig. 5. Improved Nested Depth-First Search.

search 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, $A^*+DFS$ 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 encountered, 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-component lead to pruning of the sub-searches and are inserted back into the Open-List (priority queue) of A*, which contains all horizon nodes with a neverstate in the corresponding N-SCCs. Therefore $A^* + Improved-Nested-DFS$ continues with directed search, if cycle detection in the F- and P-component components fails. As in the naive approach, cycle detection search itself might be accelerated with an evaluation function heading back to the states where it was started.

Figure 6 visualizes this strategy for our simple example. The Never Claim corresponds to a response property. It 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.

5 Heuristics for Errors in Protocols

In this section we introduce search heuristics to be used in the detection of errors in models written in Promela. We start off with precompiling techniques that help to efficiently compute different heuristic estimates.
5.1 Precompiling State Distance Tables

We now discuss how to calculate heuristic estimates through a precompilation step. We assume that a transition system $T = (T_1, \ldots, T_k)$ is given with $T_i$ being the set of transitions within the process $P_i$. We use $S$ to denote global system states. In $S$ we have a set $P$ of currently active processes $P_1, \ldots, P_k$. We write $pc_i$ to denote the current control state for process $P_i$. The information we infer is the Local State Distance Table $D$ that is defined for each process type. The value $D_i(u, v)$ fixes the minimal number of transitions necessary to reach the local state $u \in S_i$ starting from the local state $v \in S_i$ in the finite state machine representation for $P_i$. The matrix $D_i$ is determined cubic time [4] with respect to the size of the number of states in the finite state representation of $P_i$.

5.2 The Formula-Based Heuristics

The formula-based heuristics assumes a logical description $f$ of the failure to be searched. Given $f$ and starting from $S$, $H_f(S)$ is the estimation of the number of transitions necessary until a state $S'$ is reached where $f(S')$ holds. Similarly, $\overline{H}_f(S)$ is the minimum number of transitions that have to be taken until $f$ is violated. Table 3 depicts the distance measure $H_f(S)$ of the failure formula that we used. The estimator $\overline{H}_f(S)$ is defined analogously.

We allow formulae to contain other terms such as relational operators and Boolean functions over queues, since they often appear in failure specifications of safety properties: The function $q?t$ is read as message at head of queue $q$ tagged with $t$. Another statement is the $i\theta s$ predicate which denotes that a process with a process id $i$ of a given proctype is in its local control state $s$. 

---

**Fig. 6.** Visualization of $A^*$ and Improved-Nested-DFS for a response property.
\[
\begin{array}{l|c}
  f & H_f(S) \\
  \text{true} & 0 \\
  \text{false} & \infty \\
  a & \text{if } a \text{ then } 0 \text{ else } 1 \\
  \neg g & \Pi_g(S) \\
  g \lor h & \min\{H_g(S), H_h(S)\} \\
  g \land h & H_g(S) + H_h(S) \\
  \text{full}(q) & \text{capacity}(q) - \text{length}(q) \\
  \text{empty}(q) & \text{length}(q) \\
  q[t] & \text{minimal prefix of } q \text{ without } t \text{ (+1 if } q \text{ contains no message tagged with } t) \\
  a \oplus b & \text{if } a \oplus b \text{ then } 0, \text{ else } 1 \\
  i \ominus s & D_i(pc_i, s) \\
\end{array}
\]

Table 3. The formula-based heuristics: \(a\) denotes a Boolean variable and \(g\) and \(h\) are logical predicates, \(t\) is a transition, \(q\) a queue. The symbol \(\oplus\) represents relational operators (=, \(\neq\), \(\leq\), \(\leq\), \(\geq\), \(\geq\)).

In the definition of \(H_{g \land h}\) and \(\Pi_{g \lor h}\), we can replace \(\text{plus} \; (+)\) with \(\text{max}\) if we want a lower bound. In some cases the proposed definition is not optimistic, e.g., when repeated terms appear in \(g\) and \(h\). The estimate can be improved based on a refined analysis of the domain. For example suppose that variables are only decremented or incremented, then \(H_{a \oplus b}\) can be fixed as \(a - b\).

**Heuristics for Safety Properties**

**Invariants.** System invariants are state predicates that are required to hold over every reachable system state \(S\). To obtain a heuristics it is necessary to estimate the number of system transitions until a state is reached where the invariant does not hold. Therefore, the formula for the heuristics is derived from invariant.

**Assertions.** Promela allows to specify logical assertions. Given that an assertion \(a\) labels a transition \((u, v)\), with \(u, v \in S_i\), then we say \(a\) is violated if the formula \(f = (i @ u) \land \neg a\) is satisfied. According to \(f\) the estimate \(H_f\) for assertion violation can now be derived.

**Deadlocks.** \(S\) is a deadlock state if there is no transition starting from \(S\) and at least one of the processes of the system is not in a valid endstate, i.e., no process has a statement that is executable. In Promela, there are statements that are always executable: assignments, else statements, run statements (used to start processes), etc. 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. A naive approach to the derivation of an estimator function is to count the number of active (or non-blocked) processes in the current state \(S\). We call this estimator \(H_{ap}\). It turns out that best-first search using this estimator is quite effective in practice. For the formula based heuristics
$H_f$ we can devise conditions for executability for a significant portion of Promela statements:

1. Untagged receive operation ($q?x$, with $x$ variable) are not executable if the queue is empty. The corresponding formula is $\neg \text{empty}(q)$.
2. 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. Send operations ($q!m$) are not executable if $q$ is full indicated by the predicate $\neg \text{full}(q)$.
4. Conditions (boolean expressions) are not executable if the value of the condition is false corresponding to the term $c$.

We now turn to the problem of estimating the number of transitions necessary to reach a deadlock state. The deadlock in state $S'$ can be formalized as the conjunct

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

where the predicate $\text{blocked}(i, pc_i(S'), S')$ is defined as

$$\text{blocked}(i, u, S) \equiv (i@u) \land \bigwedge_{t=\{u,v\} \in \mathcal{T}_i} \neg \text{executable}(t, S).$$

Unfortunately, we do not know the set of states in which the system deadlocks such that we cannot compute the formula at exploration time. A possible solution to this problem is to approximate the deadlock formula. First we determine in which states a process can block and call such states dangerous. Therefore, we consider a process $P_i$ to be blocked if $\text{blocked}(i, u, S)$ is valid for some $u \in C_i$, with $C_i$ being the set of dangerous states of $P_i$. We define $\text{blocked}(i, S)$ as a predicate for process $P_i$ to be blocked in system state $S$, i.e., $\text{blocked}(i, S) = \bigvee_{u \in C_i} \text{blocked}(i, S, u)$ and approximate the deadlock formula with $\text{deadlock}' = \bigwedge_{P_i \in P} \text{blocked}(i, S)$.

**Heuristics for the Violation of Liveness Properties** For the validation of LTL specifications we need a heuristics for accelerating the search into the direction of the accepting state in the Never Claim. This can be achieved by declaring all accepting states as dangerous and use the local distance table to derive an estimate. An alternative is to collect all incoming transition labels for the accepting states and build a formula-based heuristics on the disjunction of that labeling. For the example of the response property we devise the heuristics $H_p^{\land \neg q}$.

During the second phase of the nested depth-first search we need cycle-detection search algorithms. Since we know which accepting state to search for we can refine $H_f(S)$ for the given state $S$ as

$$f = \bigwedge_{P_i \in P} i@pc_i(S)$$
Designer Devised Heuristics The designer of the protocol can support the search for failures by devising a more accurate heuristics than the automatically inferred one. In HSF-SPIN, there are several options. First of all, the designer can alter the recursive tabularized definition of the heuristics estimate to improve the inference mechanism. Another possibility is to concretize deadlock occurrences in the Promela code. Without designer intervention, all reads, sends and conditions are considered dangerous. Additionally, the designer can explicitly define which states of the processes are dangerous by including Promela labels with prefix danger into the protocol specification.

6 The Model Checker HSF-SPIN

We chose SPIN as a basis for HSF-SPIN. It inherits most of the efficiency and functionality of Holzmann's original source of SPIN as well as the sophisticated search capabilities of the Heuristic Search Framework (HSF) [9]. HSF-SPIN uses Promela as its modeling language. We refined the state description of SPIN to incorporate solution length information, transition labels and predecessors for solution extraction. We newly implemented universal hashing, and provided an interface consisting of a node expansion function, initial and goal specification. In order to direct the search, we realized different heuristic estimates. HSF-SPIN also writes trail information to 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 basically 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 have to be 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 partial search by implementing sequential bit-state hashing [14]. Especially for the IDA* algorithm, bit-state hashing supports the search for various beams in the search trees. Although the hash function does not disambiguate all synonyms and the length of a witness is often minimal [10].

The result is an model checker that 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, weightening 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.

7 Experimental Results

All experimental results were produced on a SUN workstation, UltraSPARC-II CPU with 248 Mhz. If nothing else is stated, the parameters while experimenting with SPIN (3.3.10) and HSF-SPIN are a depth bound of 10,000 and a memory limit of 512 MB. Supertrace is not used, but partial order reduction is used in
SPIN. We list our experimental results in terms of expanded states and witness path length, i.e., the length of the counterexample. SPIN does not give the number of expanded states. We calculate it as the number of stored states plus one; in SPIN all stored states except the error state are expanded due to the depth first search traversal. Note that we apply SPIN with partial order reduction, while HSF-SPIN does not yet include this feature.

7.1 Experiments on Detecting Deadlocks

This section is dedicated to experiments with protocols that contain deadlocks. Table 4 depicts experimental results with these protocols. For parametrized protocols, we have used the largest configuration that a breadth-first search (BFS) can solve. We experimented with two heuristics for deadlock detection: \( H_{ap} \) and \( H_f + U \): \( H_{ap} \) is the weak heuristics, counting the number of active processes; and \( H_f + U \) is the formula based heuristics, where the deadlock formula is inferred from the user designated dangerous states. In A*, \( H_f + U \) seem to perform better than \( H_{ap} \). On the other hand, with best-first search the results achieved for both heuristics are similar. Therefore, we give the results with \( H_{ap} \) for BF only.

BFS and A* find optimal solutions, while BF finds optimal or near to optimal solutions in most cases. To the contrary, the depth-first search (DFS) traversal in HSF-SPIN and in SPIN generally provide solutions far from the optimum. The most significant cases are the Dining Philosophers and the Snoopy protocol. SPIN finds counterexamples of length larger than 1,000, while the optimal solution is about 30 times smaller. In some cases, A* expands almost as many nodes as BFS, which indicates a less-informed heuristic estimate. This weakness is compensated in best-first searches, in which the number of expanded nodes is smaller than in other search strategies for most cases.

In [10] we analyzed the scalability of the search strategies. Evidently, BFS does not scale. A* and DFS also tend to struggle when the protocols are parametrized with higher values. However, best-first search seems to be very stable: in most cases it scales linearly with the parameter tuned, offering near-to-optimal solutions. Table 5 depicts some experimental results with the deadlock solution to the dining philosophers problem. These results show that directed search can find errors in protocols, where undirected search techniques are not able to find them. In the presented case SPIN fails to find a deadlock for large configurations of the philosophers problem.

7.2 Experiments on Detecting Violation of System Invariants

This Section is dedicated to experiments of models with system invariants. In the following table we summarize the models and the invariant that they violate. Note that we simplified the denotation of invariant for better understanding.

<table>
<thead>
<tr>
<th>Model</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elevator</td>
<td>( \Box \neg (\neg opened \lor stopped) )</td>
</tr>
<tr>
<td>POTS</td>
<td>( \neg \Box (P_1@s_1 \land P_2@s_2 \land P_3@s_3 \land P_4@s_4) )</td>
</tr>
<tr>
<td>Protocol</td>
<td>HSF-SPIN</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>GARP</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>BFS</td>
</tr>
<tr>
<td>Expanded States</td>
<td>834</td>
</tr>
<tr>
<td>Generated States</td>
<td>2,799</td>
</tr>
<tr>
<td>Witness Length</td>
<td>16</td>
</tr>
<tr>
<td><strong>Philosophers ($p = 8$)</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>1,801</td>
</tr>
<tr>
<td>Generated States</td>
<td>10,336</td>
</tr>
<tr>
<td>Witness Length</td>
<td>34</td>
</tr>
<tr>
<td><strong>Snoopy</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>37,191</td>
</tr>
<tr>
<td>Generated States</td>
<td>131,475</td>
</tr>
<tr>
<td>Witness Length</td>
<td>40</td>
</tr>
<tr>
<td><strong>Telegraph ($p = 6$)</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>75,759</td>
</tr>
<tr>
<td>Generated States</td>
<td>445,434</td>
</tr>
<tr>
<td>Witness Length</td>
<td>38</td>
</tr>
<tr>
<td><strong>Marriage ($p = 4$)</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>403,311</td>
</tr>
<tr>
<td>Generated States</td>
<td>1,429,380</td>
</tr>
<tr>
<td>Witness Length</td>
<td>62</td>
</tr>
<tr>
<td><strong>GIOP ($u = 1, s = 2$)</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>49,679</td>
</tr>
<tr>
<td>Generated States</td>
<td>168,833</td>
</tr>
<tr>
<td>Witness Length</td>
<td>61</td>
</tr>
<tr>
<td><strong>Basic Call ($p = 2$)</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>80,137</td>
</tr>
<tr>
<td>Generated States</td>
<td>199,117</td>
</tr>
<tr>
<td>Witness Length</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 4. Detection of Deadlocks in Various Protocols.

The search for the violation is performed with $H_{-i}$ as heuristic estimate, where $i$ is the system invariant. Table 6 depicts the results of experiments with two models: an Elevator model, and the model of a Public Old Telephon System (POTS). The latter is not scalable, and the former has been configured with 3 floors. For the Elevator model, the meaning of the invariant is self explaining. For the POTS model, the invariant describes the fact that not all processes are in a conversation state. As explained in [19], we use this invariant to test whether a given POTS model is capable of establishing a phone conversation at all.

As the Elevator model violates a very simple invariant, the results show that A* performs like breadth-first search; an optimal solution is found, but the number of expanded nodes is almost the same. SPIN and our depth-first search algorithm (DFS) yield about same results. The number of expanded nodes is small compared to breadth-first search and best-first search expands more nodes than DFS for a better solution quality. However, best-first search does not approximate the solution quality. The cause of these unexpected bad performances of the heuristic search algorithms is the restricted range of the heuristic estimate: the integer range 0..2. The quality of the estimate and the efficiency of the heuristic search procedures for system invariants correlates with the amount of information that can be extracted from the invariant.
<table>
<thead>
<tr>
<th>$p$</th>
<th>Expanded States</th>
<th>Generated States</th>
<th>Witness Length</th>
<th>Expanded States</th>
<th>Generated States</th>
<th>Witness Length</th>
<th>Expanded States</th>
<th>Generated States</th>
<th>Witness Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10 barring 12</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>14</td>
<td>10</td>
<td>10</td>
<td>14</td>
<td>14</td>
<td>19</td>
<td>19</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>39</td>
<td>97</td>
<td>21</td>
<td>21</td>
<td>69</td>
<td>57</td>
<td>77</td>
<td>75</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>54</td>
<td>18</td>
<td>18</td>
<td>26</td>
<td>54</td>
<td>54</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>1,801</td>
<td>1,365</td>
<td>41</td>
<td>69</td>
<td>249</td>
<td>1,365</td>
<td>10,336</td>
<td>1,797</td>
<td>646</td>
</tr>
<tr>
<td></td>
<td>10,336</td>
<td>1,797</td>
<td>97</td>
<td>69</td>
<td>646</td>
<td>1,797</td>
<td>34</td>
<td>34</td>
<td>66</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>193</td>
<td>127</td>
<td>595</td>
<td>278,097</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>50</td>
<td>50</td>
<td>98</td>
<td>9,998</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>-</td>
<td>81</td>
<td>66</td>
<td>941</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>321</td>
<td>201</td>
<td>2,626</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>66</td>
<td>66</td>
<td>130</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5. Number of expanded states and solution lengths achieved by A* in the dining philosophers protocol ($p$=number of philosophers).

<table>
<thead>
<tr>
<th>Elevator</th>
<th>BFS</th>
<th>DFS</th>
<th>A*</th>
<th>Best-First</th>
<th>DFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expanded States</td>
<td>228,479</td>
<td>310</td>
<td>227,868</td>
<td>16,955</td>
<td>305</td>
</tr>
<tr>
<td>Generated States</td>
<td>1,046,983</td>
<td>388</td>
<td>1,045,061</td>
<td>53,871</td>
<td>363</td>
</tr>
<tr>
<td>Witness Length</td>
<td>205</td>
<td>521</td>
<td>205</td>
<td>493</td>
<td>521</td>
</tr>
</tbody>
</table>

Table 6. Detection of Invariant Violations

The POTS protocol violates a more complicated invariant. The formula $f$ used for the heuristic estimate $H_I$ is the negation of the invariant. The function $f$ is a conjunction of four statements about the local state of four different processes. The heuristic estimate exploits the information of the transition graph corresponding to each process. While SPIN has serious problems to find the violation of the invariant, A*'s performance is superior. It finds an optimal solution with a relatively small number of expanded nodes. Best-First search achieves even better results, since it still finds optimal solutions expanding less nodes.

7.3 Experiments on Detecting Assertion Violations

We have a small group of models containing errors such as violation of assertions summarize as follows.
Table 7 depicts experimental results with these protocols. The results show that directed search strategies in HSF-SPIN offer shorter counterexamples for assertion violations than SPIN. For the GARP Protocol the number of expanded states is considerably high, since the heuristic according to the assertion false is very weak. In all other cases, the number of expansions for heuristic search is by far smaller smaller than the corresponding number of expanded states in SPIN or exceeds it by at most three times.

<table>
<thead>
<tr>
<th>Model</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lynch’s Protocol</td>
<td>i = last_i + 1</td>
</tr>
<tr>
<td>Barlett</td>
<td>mr = (imr + 1) % max</td>
</tr>
<tr>
<td>Mutex</td>
<td>in = 1</td>
</tr>
<tr>
<td>Relay</td>
<td>(k_{141} = (s_1 &amp; \neg k_{121})) &amp; (k_{121} = (\text{dienstv} &amp; (\neg s_1 \lor k_{121})) \land (k_{142} = (s_2 &amp; \neg k_{122})) \land (k_{122} = (\text{dienstv} &amp; (\neg s_1 \lor k_{122})) \land (\text{dienstv} = (k_{141} \lor k_{142})) \leq (k_{141} &amp; k_{142}) \text{false})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HSF-SPIN</th>
<th>SPIN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lynch</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>79</td>
</tr>
<tr>
<td>Generated States</td>
<td>96</td>
</tr>
<tr>
<td>Witness Length</td>
<td>29</td>
</tr>
<tr>
<td><strong>Barlett</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>82</td>
</tr>
<tr>
<td>Generated States</td>
<td>99</td>
</tr>
<tr>
<td>Witness Length</td>
<td>20</td>
</tr>
<tr>
<td><strong>Mutex</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>349</td>
</tr>
<tr>
<td>Generated States</td>
<td>699</td>
</tr>
<tr>
<td>Witness Length</td>
<td>13</td>
</tr>
<tr>
<td><strong>Relay</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>707</td>
</tr>
<tr>
<td>Generated States</td>
<td>2,701</td>
</tr>
<tr>
<td>Witness Length</td>
<td>12</td>
</tr>
<tr>
<td><strong>GARP</strong></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>17,798</td>
</tr>
<tr>
<td>Generated States</td>
<td>53,001</td>
</tr>
<tr>
<td>Witness Length</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 7. Detection of Asserting Violations in Various Protocols.

7.4 Experiments on Detecting Violation of LTL Properties

In the following table we summarize test cases for the detection of LTL property violations. Note that the error in the GIOP protocol has been seeded by explicit source code annotation.

<table>
<thead>
<tr>
<th>Model</th>
<th>LTL formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternating Bit</td>
<td>□(p → ((□q) ∨ (□q)))</td>
</tr>
<tr>
<td>Elevator</td>
<td>□(p → (□(q ∧ r)))</td>
</tr>
<tr>
<td>GIOP</td>
<td>□(p → (□(q ∧ r)))</td>
</tr>
</tbody>
</table>
The LTL properties of the Elevator and GIOP protocols correspond to the Response (Globally) pattern, the structure of the property in the alternating bit is similar such that the A*+DFS algorithm for response properties can be used.

Table 8 shows experimental results on detecting the violation of LTL formulae. We used a variant of the elevator model that includes a controller satisfying the previously discussed invariant but violates a response property. This protocol has been configured with 4 floors, while the GIOP protocol is configured with 1 server and 3 clients. Comparing the results of the new proposed Improved-Nested-DFS with those of the classical Nested-DFS, the new algorithm finds shorter solutions expanding a few states less. On the other side, the ad-hoc algorithm for response properties (A*+DFS) finds the shortest solution in all cases. In the Elevator protocol it expands about 1,000 times more states than the other algorithms, and in the GIOP example it expands about 1,000 times less states. In the elevator case we trace the anomaly back to the heuristic estimate which gave a poor range of values: [0..1]. Heuristic estimates can only improve a search strategy if they have very specific knowledge of the system. A small ranged heuristic function cannot achieve this. In the GIOP case the range of values was somewhat larger ([0..6]), and obviously this improves the effectiveness of the heuristic search. This observation calls for further refinements of the heuristic functions.

<table>
<thead>
<tr>
<th></th>
<th>HSF-SPIN</th>
<th>SPIN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Alternating Bit</td>
<td>Nested-DFS</td>
</tr>
<tr>
<td>Expanded States</td>
<td>33</td>
<td>32</td>
</tr>
<tr>
<td>Generated States</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>Witness Length</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Elevator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>309</td>
<td>251</td>
</tr>
<tr>
<td>Generated States</td>
<td>381</td>
<td>288</td>
</tr>
<tr>
<td>Witness Length</td>
<td>405</td>
<td>391</td>
</tr>
<tr>
<td>GIOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expanded States</td>
<td>404,799</td>
<td>404,619</td>
</tr>
<tr>
<td>Generated States</td>
<td>1,957,563</td>
<td>1,957,390</td>
</tr>
<tr>
<td>Witness Length</td>
<td>430</td>
<td>158</td>
</tr>
</tbody>
</table>

Table 8. Detection of Violation of Liveness Properties in Various Protocols.

We also performed full validation experiments with a version of the elevator protocol that satisfies the response property and observed that Improved-Nested-DFS executes less transitions (716,715) than classical Nested-DFS (979,336).

7.5 Performance of HSF-SPIN

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 parametrized 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 3 in comparison with HSF-SPIN.

8 Conclusion

In this paper we commenced by arguing that there is a need for improving the efficiency of model checking. It is desirable to obtain shorter error witnesses in order to more easily understand errors that the model checker reports. A reduction in the number of visited states during state space search is also desirable since this renders larger models executable. While in previous work the improvements were limited to safety properties, we now present an approach to improving the validation of a large class of non-safety properties. We view this as a step of developing HSF-SPIN into a full-fledged model checker.

The work centers around an algorithm for LTL property checking that is an improvement to nested depth first search. The algorithm exploits the structure of the Never Claim and heuristic estimates in order to find cycles faster. We argued that based on the translation of LTL formulae to Büchi Automata implemented in SPIN we can improve LTL property checking for a large class of specification patterns used in practice. Next we presented heuristics to be used in search algorithms for different classes of properties. We then presented HSF-SPIN, and illustrated its application to a number of protocol examples.

As future work we plan to analyze the proposed improvement of the nested depth-first search algorithm. We plan to perform further experiments to verify the reduction in the number of performed transitions by the new algorithm as well as refinements of the heuristic estimates. It has been shown that nested-depth first search and partial order reductions can coexist [15]. Therefore, we currently investigate how to reconcile partial order reduction and directed search.

Acknowledgements The authors would like to thank the anonymous referees for helpful references to related work.

References