Directed Explicit Model Checking with HSF-SPIN

Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue

Institut für Informatik
Albert-Ludwigs-Universität
Georges-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 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 [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 [22] 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

[1] For the purpose of this paper we use the word "validation" to denote the experimental approach to establishing the correctness of a piece of software, e.g., by testing or model checking, while we use the word "verification" to denote the use of formal theorem proving techniques for the same purpose.

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[2] [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 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 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.

In particular, 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 new algorithm performs less transitions during state space search and finds shorter counterexam-

[2] Available from `http://netlib.bell-labs.com/netlib/spin`.

ples than with the classical nested-depth first search. Given the Promela *Never Claim* A of the LTL-formula, the necessary static analysis can be performed in linear time with respect to the number of states in A. Automatically inferring the heuristic estimate by analyzing the specified formula for the failure turns out to support the more general setting as well. We improve the heuristic estimate by taking the structure of the temporal property into account.

Related Work. In earlier work work on the use of directed search in model checking the authors apply best-first exploration to protocol validation [21]. 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 to *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. However, the approach in [21] 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. We propose the use of various search strategies. Also, while the labelings used in [21] 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 [28] use BDD-based symbolic search with the Murϕ validation tool [28]. The best first search procedure incorporates symbolic information based on the Hamming distance between two states. This work has been improved in [26], where a BDD-based version of the A* algorithm [11] 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.

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)[3], the optical telegraph protocol [17], an elevator model[4], a deadlock solution to Dijkstra's dining philosopher problem, and a model of a concurrent program that solves the stable marriage problem [23]. Real-World examples that we use include the Basic Call processing protocol [24], a model of a relay circuit [27], the Group Address Registration Protocol GARP [25], the CORBA GIOP protocol [18], and the telephony model POTS [19][5].

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.

[3] Available from `http://netlib.bell-labs.com/netlib/spin`

[4] Available from `http://www.inf.ethz.ch/personal/biere/teaching/mctools/elsim.html`

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

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 improved 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 of applying HSF-SPIN to various protocol examples 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 $L(\mathcal{A})$ of a Büchi-Automaton $\mathcal{A}$ consists of all runs that are accepting. The expressive power of Büchi-Automata includes LTL.

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 $\neg f, f \vee g, f \wedge g, X\ f, F\ f, G\ f$, and $f\ U\ g$. In LTL, temporal modalities are expressed through the operators $\Box$ for *globally* (G) and $\Diamond$ for *eventually* (F).

In automata-based Model Checking we we construct the Büchi-Automaton $\mathcal{A}$ and the automaton $\mathcal{B}$ that represents the system M. $\mathcal{A}$ 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 $\mathcal{B}$ satisfies $\mathcal{A}$ when $L(\mathcal{B}) \subseteq L(\mathcal{A})$. 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 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 $\Diamond(p \wedge \ \Box\neg\ q)$. The Büchi-Automaton and the corresponding Promela Never-Claim for the negated response property are illustrated in Figure 6.

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 $\mathcal{A}$ and $\mathcal{B}$: Assume that $\mathcal{A}$ is in state s and $\mathcal{B}$ is in state t. $\mathcal{B}$ can perform a transition out of t if $\mathcal{A}$ 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 a cycle through at

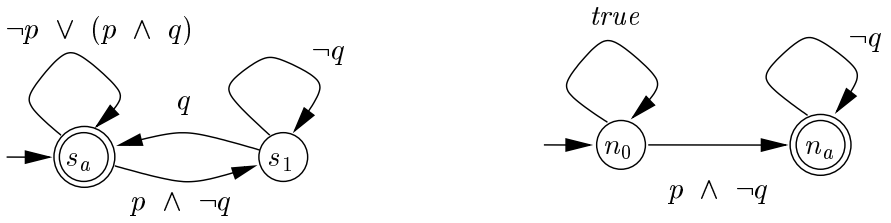

Fig. 1. Büchi-Automaton for response property (left) and for its negation (right).

least one accepting state of $\mathcal{A}$. $L(\mathcal{B}) \cap \overline{L(\mathcal{A})}$ 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.

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 suffer 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.

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

```
A*(s)
  Open ← {(s, h(s))}; Closed← {}
  while (Open ≠ ∅)
    u ← Deletemin(Open); Insert(Closed,u)
    if (failure(u)) exit Safety Property Violated
    for all v in Γ(u)
      f'(v) ← 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.

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 *Delete* 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 Euclidean distance $\|u - t\|_2$ to state t, then only the nodes in the hatched region are ever removed from the priority queue.

Weighting 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 exhibit BF, and for $\alpha = 1$ we simulate BFS.

4 Searching for Liveness Property Violations

4.1 Nested-Depth-First Search

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.

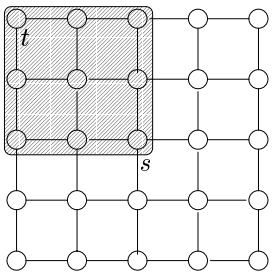

Fig. 2. The Effect of Heuristic Search in a Grid Graph.

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. [4]).

```
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

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, 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 which will be applicable to a large set of practical property specifications. It will be described in the sequel of this Section.

4.2 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. SCCs can be computed in linear time [5]. In the Never-Claim of the example in Figure 6 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 encounterd 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 by non-accepting states or only by accepting ones. 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 structure can be relaxed according to the following classification of the SCCs. We call a SCC *accepting*, if at least one of its states is accepting. Otherwise it is *non-accepting* (N-SCC). We further distinguish between *full* acceptance (F-SCC) and *partial* acceptance (P-SCC). Full acceptance is given if no cycle in the SCC contains only non-accepting states. 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 *bad* sequence is found. We indicate the presence of endstates with the letter S. Bad sequences are tackled similarily 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 *Absence* (85/555), *Universality* (119/555), *Existence* (27/555), *Response* (245/555), *Precedence(26/555)*, and *Others (53/555)*. Using this pattern scheme and the pattern modifiers *Globally, Before, After, Between*, and *Until* we obtain a partitioning into SSCs according to Fig 2. We derived these classifications from analysing the never claims generated by SPIN using the `spin -f` command.

Pattern	Globally	Before	After	Between	Until
Absence	S+N	S+N	S+N	S+N	S+N+P
Universality	S+N	S+N	S+N	S+N+F	S+N+P
Existence	F	S+P	N+F	S+N+P	S+N+P
Response	F	S+N+P+F	N+F	S+N+P+F	S+N+P+F
Precedence	S+N+P	S+N	P	S+N	S+N+P

Table 2. Strongly-Connected Component Classification for LTL-Specification Patterns.

4.3 Improved Nested Depth-First-Search

In this section we present an improvement of the Nested-DFS algorithm called *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 Figure 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 *expanded*. 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 a endstate in the Never Claim is reached the algorithm terminates imediately. Figure 4 depicts the different cases of cycles detected in the search. The correctness of Improved-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.

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. This suggests to add the SCCs pre-computed classification and the Improved-Nested-DFS to the SPIN validation tool.

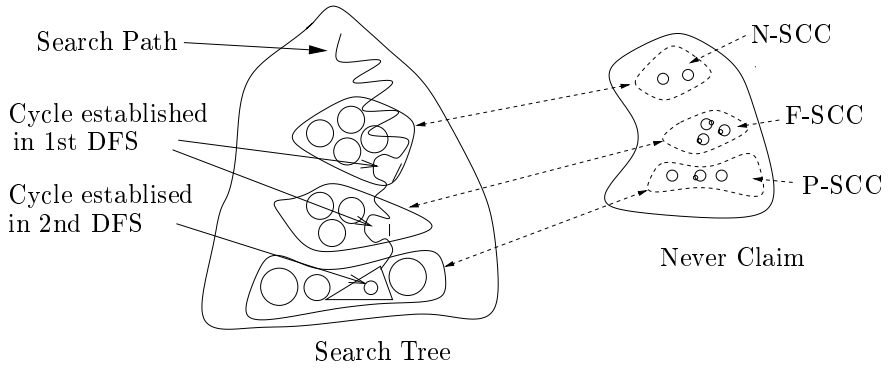

Fig. 4. Visualization of the Different Cases in Improved-Nested-DFS.

4.4 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 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* + DFS* 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-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.

```
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')
```

Fig. 5. Improved Nested Depth-First Search.

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 [5] with respect to the size of the number of states in the finite state representation of P_i.

5.2 The Formula-Based Heuristic

The formula-based heuristic 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

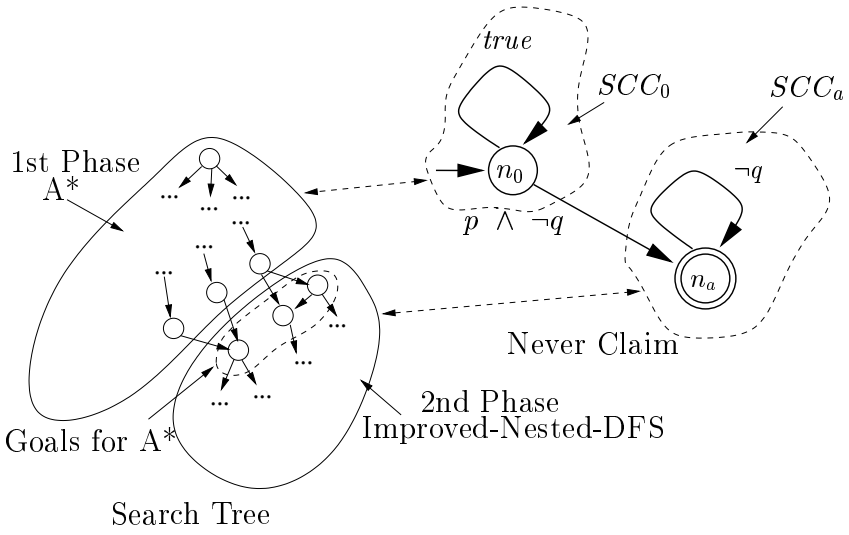

Fig. 6. Visualization of A* and Improved-Nested-DFS for a response property.

safety properties: The function $q?[t]$ is read as *message at head of queue q tagged with t*. Another statement is the $i@s$ predicate which denotes that a process with a process id i of a given proctype is in its local control state s.

In the definition of $H_{g \wedge h}$ we can replace *plus* $(+)$ with *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 $H_{x=y}$ can be fixed as $x - y$.

Heuristics for Safety Properties

Invariants. System invariants are state predicates that are required to hold over every reachable system state S. To obtain a heuristic 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 heuristic 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 $f = (i@u) \wedge \neg a$ 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

f	$H_f(S)$
$true$	0
$false$	∞
a	if a then 0 else 1
$\neg g$	$\overline{H}_g(S)$
$g \vee h$	$\min\{H_g(S), H_h(S)\}$
$g \wedge h$	$H_g(S) + H_h(S)$
$full(q)$	$capacity(q) - length(q)$
$empty(q)$	$length(q)$
$q?[t]$	minimal prefix of q without t (+1 if q contains no message tagged with t)
$x \otimes y$	if $x \otimes y$ then 0, else 1
$i@s$	$D_i(pc_i, s)$

Table 3. The formula-based heuristic: a denotes a Boolean variable and g and h are logical predicates, t is a transition, q a queue. The symbol $\otimes$ represents relational operators $(=, \neq, \leq, \leq, \geq, \geq)$ for natural numbers x and y.

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 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 heuristic 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 $\neg\ empty(q)$.
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 $\neg full(q)$.
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

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

where the predicate $blocked(i, pc_i(S'), S')$ is defined as

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

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 $blocked(i, u, S)$ is valid for some $u \in C_i$, with C_i being the set of dangerous states of P_i. We define $blocked(i, S)$ as a predicate for process P_i to be blocked in system state S,i.e., $blocked(i, S) = \bigvee_{u \in C_i} blocked(i, S, u)$ and approximate the deadlock formula with $deadlock' = \bigwedge_{P_i \in P} blocked(i, S)$.

Heuristics for the Violation of Liveness Properties For the validation of LTL specifications we need a heuristic for accelerating the search into the direction of accepting SCCs (P-SCCs and F-SCCs) in the Never Claim. This can be achieved by declaring all accepting states as dangerous and by using the local distance table to derive an estimate. An alternative is to collect all incoming transition labels for the accepting SCCs and build a formula-based heuristic on the disjunction of that labeling. For the example of the response property we devise the heuristic $H_{(p \wedge \neg q) \wedge never@n_a}$.

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 heuristic 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 heuristic estimate to improve the inference mechanism. Another possibility is to concretize deadlock occurences 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, weighing 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 parameterized 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 parameterized 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.

	HSF-SPIN					SPIN
GARP	BFS	DFS	A*,H_{ap}	A*,$H_f + U$	Best-First,H_{ap}	DFS
Expanded States	834	62	1,145	53	33	56
Generated States	2,799	70	3,417	194	60	64
Witness Length	16	50	16	18	28	58
Philosophers ($p = 8$)						
Expanded States	1,801	1,365	41	69	249	1,365
Generated States	10,336	1,797	97	69	646	1,797
Witness Length	34	1,362	34	34	66	1,362
Snoopy						
Expanded States	37,191	5,823	32,341	6,872	152	1,243
Generated States	131,475	7,406	110,156	24,766	299	1,646
Witness Length	40	4,676	40	40	40	1,113
Telegraph ($p = 6$)						
Expanded States	75,759	44	38	366	38	44
Generated States	445,434	45	108	1,897	108	45
Witness Length	38	44	38	38	38	44
Marriers ($p = 4$)						
Expanded States	403,311	294,549	333,529	284,856	6,281	36,340
Generated States	1,429,380	1,088,364	1,176,336	996,603	16,595	47,221
Witness Length	62	112	62	62	112	112
GIOP ($u = 1, s = 2$)						
Expanded States	49,679	247	38,834	27,753	315	338
Generated States	168,833	357	126,789	89,491	504	377
Witness Length	61	136	61	61	83	136
Basic Call ($p = 2$)						
Expanded States	80,137	115	4,170	36	57	117
Generated States	199,117	136	8,785	60	89	140
Witness Length	30	96	30	30	42	96

Table 4. Detection of Deadlocks in Various Protocols.

		HSF-SPIN					SPIN
p		BFS	DFS	A*,H_{ap}	A*,$H_f + U$	Best-First,H_{ap}	DFS
2	Expanded States	10	12	10	10	10	12
	Generated States	12	14	12	12	12	14
	Witness Length	10	10	10	10	10	10
3	Expanded States	18	19	16	14	32	19
	Generated States	30	22	22	19	52	22
	Witness Length	14	18	14	14	14	18
4	Expanded States	33	57	21	21	69	57
	Generated States	77	75	33	27	155	75
	Witness Length	18	54	18	18	26	54
8	Expanded States	1,801	1,365	41	69	249	1,365
	Generated States	10,336	1,797	97	69	646	1,797
	Witness Length	34	1,362	34	34	66	1,362
12	Expanded States	-	278,097	61	50	539	278,097
	Generated States	-	46,435	193	127	1,468	46,435
	Witness Length	-	9,998	50	50	98	9,998
16	Expanded States	-	-	81	66	941	-
	Generated States	-	-	321	201	2,626	-
	Witness Length	-	-	66	66	130	-

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

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.

Model	Invariant
Elevator	$\Box(\neg \mathit{opened} \vee \mathit{stopped})$
POTS	$\neg\Diamond(P_1@s_1 \wedge P_2@s_2 \wedge P_3@s_3 \wedge P_4@s_4)$

The search for the violation is performed with $H_{\neg i}$ 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 configurated 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 are 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

	HSF-SPIN				SPIN
Elevator	BFS	DFS	A*	Best-First	DFS
Expanded States	228,479	310	227,868	16,955	305
Generated States	1,046,983	388	1,045,061	53,871	363
Witness Length	205	521	205	493	521
POTS					
Expanded States	31,792	1,465,103	228	65	2,012,345
Generated States	55,402	4,460,586	471	129	2,962,232
Witness Length	67	1,203	67	67	872

Table 6. Detection of Invariant Violations

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.

The POTS protocol violates a more complicated invariant. The formula f used for the heuristic estimate H_f is the negation of the invariant. Therfore, 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. Additionally we suspect that the discrepancies between SPIN's and HSF-SPINs DFS-exploration can be treated back to the differences in the node expansion for atomic regions.

7.3 Experiments on Detecting Assertion Violations

We have a small group of models containing errors such as violation of assertions summarize as follows.

Model	Assertion
Lynch's Protocol	$i = last_i + 1$
Barlett	$mr = (lmr + 1)\%\max$
Mutex	$in = 1$
Relay	$(k_{14_1} = (s_{1_1} \wedge \neg k_{12_1})) \wedge$
	$(k_{12_1} = (dienstv \wedge (\neg s_{1_1} \vee k_{12_1}))) \wedge$
	$(k_{14_2} = (s_{1_2} \wedge \neg k_{12_2})) \wedge$
	$(k_{12_2} = (dienstv \wedge (\neg s_{1_2} \vee k_{12_2}))) \wedge$
	$(dienstv = (k_{14_1} \vee k_{14_2}))) <= \neg(k_{14_1} \wedge k_{14_2})$
GARP	$false$

Table 7 depicts experimental results with these protocols. The data shows 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 with the exception of the Relay protocol, where the number of expanded nodes in A* exceeds the number found in SPIN by at most three times.

	HSF-SPIN				SPIN
Lynch	BFS	DFS	A*	Best-First	DFS
Expanded States	79	50	72	63	47
Generated States	96	52	89	79	50
Witness Length	29	46	29	29	46
Barlett					
Expanded States	82	348	61	26	262
Generated States	99	383	76	33	289
Witness Length	20	246	20	20	251
Mutex					
Expanded States	349	202	150	24	202
Generated States	699	363	300	48	363
Witness Length	15	54	15	15	54
Relay					
Expanded States	707	342	665	151	341
Generated States	2,701	719	2,292	1,069	870
Witness Length	12	190	12	120	190
GARP					
Expanded States	17,798	1,040	18,968	4,727	150
Generated States	53,001	2,818	56,406	13,107	187
Witness Length	29	54	29	39	55

Table 7. Detection of Assertion 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.

Model	LTL formula
Alternating Bit	$\Box(p \rightarrow ((\Diamond q) \vee (\Diamond q))$
Elevator	$\Box(p \rightarrow \Diamond(q \wedge r))$
GIOP	$\Box(p \rightarrow \Diamond(q \wedge r))$

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 configurated 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 heurisitc 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.

	HSF-SPIN			SPIN
Alternating Bit	Nested-DFS	Improved-Nested-DFS	A*+DFS	DFS
Expanded States	33	32	11	24
Generated States	37	36	12	32
Witness Length	64	64	22	46
Elevator				
Expanded States	309	251	217,810	253
Generated States	381	288	1,276,391	401
Witness Length	405	391	377	405
GIOP				
Expanded States	404,799	404,619	113	53,812
Generated States	1,957,563	1,957,390	1,158	107,987
Witness Length	430	158	158	430

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. In particular we want to study if the classification of the SCCs of the never claim is inherent to the specification pattern or if it depends on the algorithm used for the translation from the LTL formula. We also 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.

One of our central research aims is develop an integrated protocol definition and validation system for Promela protocols that integrates HSF-SPIN and visualization front-ends for protocol design and visualized error traces, such as VIP [20] and VEGA [3].

References

1. A. Biere. μcke - efficient μ-calculus model checking. In *Computer Aided Verification*, pages 468–471, 1997.
2. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, Apr 1983.
3. C. A. Bröcker and S. Schuierer. Vega—a user-centered approach to the distributed visualization of geometric algorithms. Technical Report 117, Institut für Informatik, University of Freiburg, 1998.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
6. R. Dial. Shortest path forest with topological ordering. *Communications of the ACM*, pages 632–633, 1969.
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. *Data Structures and Learning Algorithms in State Space Search*. PhD thesis, University of Freiburg, 1999. Infix.
10. S. Edelkamp, A. L. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI Symposium on Model-based Validation of Intelligence*, 2001.
11. S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, pages 81–92, 1998.
12. M. G. Gouda. Protocol verification made simple: a tutorial. *Computer Networks and ISDN Systems*, 25(9):969–980, 1993.
13. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Trans. on SSC*, 4:100, 1968.
14. G. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, November 1998. extended and revised version of Proc. PSTV95, pp. 301-314.
15. G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The Spin Verification System*, pages 23–32. American Mathematical Society, 1996.
16. G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Protocol Specification, Testing, and Verification*, 1987.
17. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.
18. 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.
19. M. Kamel and S. 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.
20. S. Leue and G. Holzmann. v-Promela: A visual, object-oriented language for Spin. In *IEEE International Symposium on Object-oriented Real-time Distributed Computing*, 1999.
21. F. J. Lin, P. M. Chu, and M. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *ACM*, pages 126–135, 1988.
22. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.
23. D. McVitie and L. Wilson. The stable marriage problem. *Communications of the ACM*, 1971.
24. A. Miller and M. Calder. Analysing a basic call protocol using promela/xspin. In *International SPIN Workshop*, 1998.
25. T. Nakatani. Verification of group address registration protocol using promela and spin. In *International SPIN Workshop*, 1997.
26. F. Reffel and S. Edelkamp. Error detection with directed symbolic model checking. In *World Congress on Formal Methods*, pages 195–211. Springer, 1999.
27. P. van Eijk. Verifying relay circuits using state machines. In *International SPIN Workshop*.
28. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *DAC*, pages 599–604, 1998.