# Protocol Verification with Heuristic Search

**Stefan Edelkamp** and **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 an approach to reconcile explicit state model checking and heuristic directed search. We provide experimental evidence that the model checking problem for concurrent systems, such as communications protocols, can be solved more efficiently, since finding a state violating a property can be understood as a directed search problem. In our work we combine the expressive power and implementation efficiency of the SPIN model checker with the HSF heuristic search workbench, yielding the HSF-SPIN tool that we have implemented. We start off from the A* algorithm and some of its derivatives and define heuristics for various system properties that guide the search so that it finds error states faster. In this paper we focus on safety properties and provide heuristics for invariant and assertion violation and deadlock detection. We provide experimental results for applying HSF-SPIN to two toy protocols and one real world protocol, the CORBA GIOP protocol.

## Introduction

Concurrent software systems, such as communication protocols, are inherently difficult to debug. This is due to the non-deterministic structure of concurrent systems which causes a combinatorial explosion of the system's global state space. Even if a thorough development methodology is applied, software design documents as well as actual code typically contain numerous residual faults. Classical software quality assurance measures such as code inspections, simulations and testing fail to find the majority of faults in real-world sized protocols and telecommunication software models since they usually only cover a small fraction of the entire system's state space. Model checking (Clarke, Grumberg, & Peled 2000) has been proven an efficient method to find residual design and code faults. In this paper the focus is on improving the efficiency of error detection in communication protocols through model checking based on heuristic search techniques.

In model checking, concurrent systems are modeled as sets of communicating finite state machines (CFSMs) that communicate either via synchronous or asynchronous message passing, or via shared variables. The global state transition graph is that portion of the cross product of the CFSMs that is reachable from the initial system state. System properties are often expressed in terms of assertions, dead- and life-lock predicates, temporal logic formulae or property automata. The model checker evaluates the validity of the properties over the model by interpreting its global state transition graph as a Kripke structure. Safety properties can be validated through a simple depth-first search on the system's state space, for liveness properties a two-fold nested depth-first search is necessary (Clarke, Grumberg, & Peled 2000). When property violations are detected, the model checker will return a witness which consists of a trace of events leading from the initial state into the violating state. The witness can easily be constructed from the contents of the search stack at the moment when the property violation has been observed.

Model checking techniques fall into two different approaches: The first one, called *symbolic* model checking (McMillan 1998), uses binary decision diagrams (Bryant 1985) to represent the state set. The second approach represents states *explicitly* and applies reduction techniques such as partial order reduction (Peled 1998) and partial search (Holzmann 1990) to traverse the global state graph.

In a practical software engineering setting model checking is typically applied in two different ways. In *complete* model verification a given state transition model is completely explored and the desired properties are verified. However, the size of the state graph is often so large that it is impractical to traverse all states. In this situation, only *partial* searches are performed and hence the properties are only partially validated. Partial validation has two benefits. Residual design errors are typically located rather quickly. However, should the state space search not terminate within a reasonable time frame, or should memory bounds be exceeded, then one can at least conclude that the probability of residual design errors has been greatly reduced. This increases the overall confidence in the correctness of the model or code that has been scrutinized.

Various verification tools for concurrent systems based on model checking have been developed, most notably the SPIN validator (Holzmann 1990; 1997)[1]. SPIN parses the concurrent protocol specification language *Promela* and applies possibly bounded depth-first search variants like *Supertrace* to examine various beams in the search trees (Holzmann 1987). It allows to apply partial order reduction to reduce the state space (Peled 1998) and bit-state hashing (Holzmann 1990) to compress the state description of several hundred bits down to only one or two to build hash tables with over $2^{30}$ entries. Bit-state hashing algorithms are not complete, since the hash function does not disambiguate all synonyms. Moreover, due to the depth-first traversal that SPIN employs, the length of a witness is not minimal. However, since the density of errors in the cross product state space is rather high compared to that of single agent search problems (that usually only have one target state), SPIN succeeds in finding bugs efficiently, even though they are often located at large search depths. We chose SPIN as a basis for our implementation and provide an interface for AI search algorithms for which there is ample evidence of their capability to deal with huge problem spaces. Our paper shows that using the power of heuristic search algorithms we are capable of finding errors much faster, hence requiring less search steps as well as less memory which allows us to analyze more complex problems.

Central to our work is the A* (Hart, Nilsson, & Raphael 1968) search algorithm which is a variant of Dijkstra's single-source shortest-path algorithm. A* uses edges that are re-weighted by an estimator function that takes additional search information into account (Cormen, Leiserson, & Rivest 1990). Current heuristic search techniques such as iterative deepening (Korf 1985), transposition tables (Reinefeld & Marsland 1994), finite state machine pruning (Taylor & Korf 1993), relevance cuts (Junghanns 1999), symbolic representation (Edelkamp & Reffel 1998) and memory-based heuristics (Hernádvögyi & Holte 2000) efficiently solve problems with $10^{20}$ states and beyond, which compares with performance gains in symbolic model checking (Burch *et al.* 1992).

In our work we handle safety errors such as violations of system invariants, assertion violations and deadlocks. Since the state space reduction due to heuristic search algorithms strongly depends on the quality of the estimate we have developed different heuristics for these purposes based on communication queue load, process activeness and distances to dangerous process states.

The structure of our paper is as follows: First we take a closer look at communication protocols and their representations in Promela. Then we address the architecture of the new protocol validator HSF-SPIN. Afterwards we present different known heuristic search strategies and present a new search paradigm based on the combination of partial and heuristic search. We then discuss the automatic generation of heuristic estimates for safety properties in communication protocols. The impact of our heuristics on three protocol examples is evaluated. Finally, we discuss the results, draw conclusions and indicate current and future work.

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

## Protocol Modeling with Promela

Promela is a concurrent modeling language for the description of extended communicating finite state machine models. A Promela model consists of a set of concurrent processes, known as "proctypes" in Promela parlance. Proctypes can be dynamically instantiated or deleted. Inside a proctype, execution is strictly sequential, and control flow is specified using Dijkstra-like guarded commands, a goto/labeling mechanism and inlined procedure calls. Inter proctype communication is either by shared variables, by synchronous rendez-vous message passing, or by buffered asynchronous message passing. The semantics of Promela is not fully formalized and the SPIN tool's interpretation is often used as reference, in particular the interpretation provided by SPIN's state space exploration component. Concurrency in SPIN is interpreted as interleaved state sequences. Since Promela was designed to be used in a model checker special attention was paid to ascertaining the finiteness of the resulting model. The basic data types all represent finite domains, and the Promela typing mechanism allows only finite types to be derived. Asynchronous communication is via buffered channels with bounded, finite capacity. The number of concurrently executing proctype instances is not bounded in the language definition, but all SPIN implementations limit the number to a platform-dependent constant.

There are various ways of specifying and validating state properties in Promela and SPIN. SPIN has built-in routines checking a Promela model for dead- and livelocks. To distinguish desired end states from deadlock states and in order to identify progress states for livelock analysis, the user has to apply well-defined label names to states in the model. Promela also permits the use of assertions in the code which can be used for pre and post condition-style assertional specification. Finally, to specify temporal properties, particularly liveness properties, Promela has a format to represent Büchi automata called *never claims.* SPIN provides a translation from linear time temporal logic (LTL) formulae into never claims, which greatly facilitates property specification with temporal logic.

We exemplify Promela with the well-known dining philosopher problem posed by Dijkstra. We are confronted with $n$ philosophers sitting around a table and trying to eat between meditations. Each philosopher has his own plate of spaghetti and there is a fork between each plate for a total of $n$ forks. Two forks (a left and a right one) are necessary and sufficient to start eating since the spaghetti are very slippery. The life of a philosopher consists of alternate periods of lunch and meditation time. The first obvious solution for a

```
#define MAX_PHILOSOPHERS 8
mtype={fork}
#define left forks[my_id]
#define right forks[(my_id+1)%MAX_PHILOSOPHERS]
chan forks[MAX_PHILOSOPHERS] = [1] of {bit};
proctype philosopher(int my_id)
{       do
        ::left?fork -> /* try to get left fork */
                right?fork; /* try to get right fork */
                /* eat... */
                left!fork; right!fork /* release forks */
                /* meditation... */
        od
}
init
{       byte philosophers=MAX_PHILOSOPHERS;
        atomic {
            do
                ::philosophers>0 ->
                        philosophers--;
                        run philosopher(philosophers);
                        forks[philosophers]!fork
                ::philosophers==0 ->
                        break
            od
        }
}
```

Table 1: The dining philosophers in Promela.

protocol to get and release the forks is implemented in Table 1. To get the forks the philosopher tries to take first the left fork and then tries to take the other. After eating the philosopher can quitely release both forks. This solution has a an obvious deadlock: If each philosopher takes only his left fork there is no further progress possible for requiring the other one.

## The Validator HSF-SPIN

HSF-SPIN merges the SPIN model checker and the Heuristic Search Framework (HSF) (Edelkamp 1999) for single-agent exploration problems. The goal of this merge was for HSF-SPIN to inherit most of the efficiency and functionality of Holzmann's original source of SPIN as well as the sophisticated search capabilities of HSF. In HSF-SPIN we modified the static representation of states to allow dynamic state vector allocation, since the predefined maximum size of the vector is too pessimistic in general. 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 verification of a model with HSF-SPIN is done in two phases: first the generation of an analyzer of the model, and second the verification 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. The result is an analyzer which can be invoked with a wide range of parameters: kind of error to be detected, applied algorithm, heuristic function to be used, etc. HSF-SPIN allows textual simulation to interactively traverse the state space which greatly facilitates in explaining witnesses that have been found.

We briefly present HSF[2], an efficient `c++` workbench to define and solve solitaire games like the $(n^2-1)$-Puzzle, Sokoban, and Rubik's Cube. HSF further includes efficient implementations for a generalization to the $(n^2-1)$-Puzzle and some other challenging domains. Its predecessor (Müller, Eckerle, & Ottmann 1996) was effectively applied in teaching but the implementation of the heuristic search algorithms were too slow for practical application purposes. HSF implements classical and extended AI search strategies and includes different automata-based duplicate pruning schemes. Since such duplicate elimination is supposed to support only very regular problems, which are typically not present in concurrent system models, we decided to omit this feature from the system. A state in HSF is a quadruple: the packed state description for compactly and uniquely memorizing problem states and the $f$-value as the sum of generating path length and the estimate to a goal state. For tracking the solution the predecessor and the transition from the predecessor to the current state are also stored. The lists of expanded and generated states are kept in efficient priority queue structures and hash tables. Conflicts are resolved by chaining.

The software architecture of HSF is designed to combine the workbench design pattern of object-oriented programming (Gamma *et al.* 1994) with efficiency aspects found in special-purposed heuristic puzzle solvers. HSF provides an interface for a client-server Java visualization system (Hipke & Schuierer 1999).

## Heuristic Search Algorithms

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. In this section we describe how heuristic search algorithms can be used instead in order to accelerate the exploration.

**A*** accommodates the information of the *heuristic* $h(u)$, which estimates the minimum cost of a path from

[2]Available from `http://www.informatik.uni-freiburg.de/~edelkamp`.

node $u$ to a target node. A heuristic $h$ is called *consistent* if and only if $w(u,v) - h(u) + h(v) \geq 0$ for all $u$ and $v$, where $w(u,v)$ denotes the weight along the edge from $u$ to $v$. It is called *optimistic* if it is a lower bound function. We characterize A* on the basis of Dijkstra's algorithm to find shortest paths in (positively) weighted graphs from a *start node* $s$ to a set of *goal nodes* $T$ (Cormen, Leiserson, & Rivest 1990). Dijkstra's algorithm uses a priority queue maintaining the set of currently reached yet unexplored nodes. If $f(u)$ denotes the total weight of the currently best explored path from $s$ to some node $u$ the algorithm always selects a node with minimum $f$-value for expansion, updates its successors' $f$-values, and transfers it to the set of visited nodes with established minimum cost path. A* can be cast as a search through a re-weighted graph. More precisely, the edge weights $w$ are replaced by adding the heuristic difference. By this transformation, negative weights may be introduced, since not all heuristics are consistent. 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). On every path from $s$ to $u$ the accumulated weights in the two graph structures differ by $h(s)$ and $h(u)$ only. Consequently, re-weighting cannot lead to negatively weighted cycles so that the problem remains (optimally) solvable. One can show that given a optimistic heuristic the solution returned by the algorithm is indeed a shortest one.

**Weighted A*** , WA* for short, can help when the estimates are too weak. The combined merit function $f$ of the generating path length and the heuristic estimate is weighted with two parameters $w_g$ and $w_h$, where $w_g$ influences the generated path length $g$ and $w_h$ scales the heuristic estimate $h$, i.e. $f = w_g g + w_h h$. For $w_g = w_h = 1$ WA* reduces to A*, for $w_g = 0, w_h \neq 0$ we have best-first search, and for $w_g \neq 0, w_h = 0$ the algorithm reduces to breadth-first search.

**Iterative deepening A*** , IDA* (Korf 1985) for short, expands all nodes in consecutive bounded depth-first search traversal until the next horizon value has been reached. As long as main memory can be allocated, transposition tables memorize states and updates to state merits.

## Partial Search Algorithms

Invoking partial search implies that a retrieved node might be an unexpected synonym, since there is no way to distinguish a real duplicate from a false one. Therefore, reopening a node is very dangerous; the information of generating path length and predecessor path length might be false. Subsequently, we omit reopening in partial methods. Note that reopening will not be encountered when the heuristic function is consistent, since this implies that the priorities $f = g + h$ are monotonically increasing: $f(u) = g(u) + h(u) \leq g(v)+w(u,v)+h(v)-w(u,v) = f(v)$. Fortunately, most heuristics satisfy this criterion. The lack of state space coverage in partial search is compensated for by repeating the search with restarts on different hash functions. We implemented a set of universal hash functions (Cormen, Leiserson, & Rivest 1990) from which the current one is randomly chosen.

**Partial A*** , the bit-state version of A*, omits the packed state description for each fully expanded node, which is the list of visited nodes that are not contained in the priority queue. For further compaction we separate the visited list from the set of horizon nodes such that for the latter only the executed move and link to the predecessor of a node remain. These items are necessary since the access to the priority queue is almost impossible to predict and through the uncorrelated access on the horizon once the goal is found, the solution path cannot be generated.

**Partial IDA*** , like IDA*, can track the solution path on the recursion stack no predecessor link is needed. Therefore, the transposition table in IDA* is represented by a large bit-vector. A hash function maps a state $S$ to position $BT(S)$. A state $S$ is stored by setting the bit at $BT(S)$ and searched by querying $BT(S)$.

Unfortunately, the bit-state transposition table is initialized in each iteration of IDA*, since neither the predecessor nor the $f$-value are present to distinguish the current iteration from the previous ones. Since the bit-vector is large this seems to be the only bottleneck of this approach. One solution to this problems is to dynamically access the bit-vector depending on the number of expansions of the previous iteration and the branching factor, i.e. the ratio of nodes expanded in two successive iterations.

## Heuristics for Errors in Protocols

In this section we introduce some search heuristics to be used in the detection of errors in models written in Promela. We consider the violation of system invariants, the violation of assertions, and deadlocks. We use $S$ to denote global system states. In $S$ we have a set $P$ of currently active processes (proctype instances) $P_0, P_1, P_2, \ldots$. For a process $P_i$ with $pc_i$ we denote the current local state (program counter), and $T_i$ is the set of transitions within the proctype instance $P_i$.

**Violation of Invariants** System invariants are state predicates that hold over every system state $S$. To obtain estimator functions it is necessary to estimate the number of system transitions until a state is reached where the invariant does not hold. Given a logical predicate $f$ let $H_f(S)$ an estimation of the number of transitions necessary until a state $S'$ is reached where $f(S')$ holds, starting from state $S$. Similarly, we fix $\overline{H}_f(S)$ as the number of transitions necessary until $f$ is violated.

A first approach to recursively define these functions is as given in the following table ($a$ denotes a Boolean variable and $g$ and $h$ are logical predicates):

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

In the definition of $H_{g \wedge h}$ and $\overline{H}_{g \vee h}$, we can replace *plus* (+) with *max* if we want a lower bound. Note that in some cases the proposed definition is not optimistic, e.g., when repeated terms appear in $g$ and $h$.

System invariants may contain other terms such as relational operators and Boolean functions over queues. Thus $H_f$ and $\overline{H}_f$ can be fixed as follows:

| $f$ | $H_f(S)$ |
|---|---|
| $full(q)$ | $capacity(q) - length(q)$ |
| $empty(q)$ | $length(q)$ |
| $q?[t]$ | minimal prefix of $q$ without $t$ (+1 if $q$ contains no message tagged with $t$) |
| $a \otimes b$ | if $a \otimes b$ then 0, else 1 |
| $f$ | $\overline{H}_f(S)$ |
| $full(q)$ | if $full(q)$ then 1, else 0 |
| $empty(q)$ | if $empty(q)$ then 1, else 0 |
| $q?[t]$ | if $head(q) \neq t$ then 0, else maximal prefix of $t$'s |
| $a \otimes b$ | if $a \otimes b$ then 1, else 0 |

The function $q?[t]$ is read as *message at head of queue $q$ tagged with $t$*. All other functions are self-explaining. The expressions of the right sides are applied to state $S$. For example, $capacity(q)$ should be interpreted as *capacity of queue $q$ in state $S$*.

We have used the symbol $\otimes$ for representing relational operators $(=, \neq, \leq, \leq, \geq, \geq)$. We could refine $H_{a \otimes b}$ and $\overline{H}_{a \otimes b}$. Supposing that in the model, variables are only decremented or incremented, we could define $H_{a=b}$ as $a - b$. However, variables are usually used in assignments (typically in read operations). A possible idea is to perform an analysis of the model, extracting the necessary information to decide how to optimistically define the estimate functions.

Another statement that typically appears in system invariants is the *at* predicate which is used to express that a process $P$ with a process id *pid* of a given proctype $PT$ is in its local control state $s$. In Promela this is expressed as `PT[pid]@s`. We will write this as $i@s$, with $s \in S_i$. We extend our estimates as follows:

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

We use $pc_i$ to express the state of process $P_i$ in state $S$. The value $D_i(u, v)$ is the minimal number of transitions necessary for the finite state machine $P_i$ to reach state $u$ starting from state $v$, where $u, v \in S_i$. The matrix $D_i$ can be pre-computed with the all-pairs shortest-path algorithm of Floyd/Warshall in $O(|S_i|^3)$ time (Cormen, Leiserson, & Rivest 1990). Note that $|S_i|$ is small in comparison to the overall search space.

**Violations of Assertions** The Promela statement `assert` allows to label the model with logical assertions. Given that an assertion $a$ labels a transition $(u, v)$, with $u, v \in S_i$, then we say $a$ is violated if the formula $f = (i@u) \wedge \neg a$ is satisfied. We denote $f$ as *assertion*$(i, u, a)$ to distinguish assertions in our heuristic functions. The next step is to extend our functions $H$ and $\overline{H}$. One possible solution is:

| $f$ | $H_f(S)$ | $\overline{H}_f(S)$ |
|---|---|---|
| $assertion(P_i, u, a)$ | $H_{i@u}(S) + \overline{H}_a(S)$ | not defined |

As above, when aiming at optimal counterexamples we can use *max* instead of *plus* (+). Note that there is no meaningful interpretation for $\overline{H}$ in this case.

**Deadlock Detection** In concurrent systems, a deadlock occurs if at least a subset of processes and resources is in a cyclic wait situation. In Promela, $S$ is a deadlock state if there is no possible transition from $S$ to a successor state $S'$ and at least one of the processes of the system is not in a *valid endstate*. Hence, 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. The following table describes executability conditions for a significant portion of Promela statements:

**untagged receive operation** (`q?x`, **with** `x` **variable**) not executable if the queue is empty.

**tagged receive operations** (`q?t`, **with** `t` **tag**) not executable if the head of the queue is a message tagged with a different tag than `t`.

**send operations** (`q!m`) not executable if `q` is full.

**conditions (boolean expressions)** not executable if the value of the condition is false.

From the above characterization we derive the boolean function *executable*, ranging over tuples of Promela statements and global system states, as follows:

| $label(t)$ | $executable(t, S)$ |
|---|---|
| `q?x`, `x` variable | $\neg empty(q)$ |
| `q?t`, `t` tag | $q?[t]$ |
| `q!m` | $\neg full(q)$ |
| condition `c` | $c$ |

We now turn to the problem of estimating the number of transitions necessary to reach a deadlock state.

**Active Processes** As mentioned above, in a deadlock state all processes are blocked. A naive approach to deriving an estimator function is is to count the number of active (or non-blocked) processes in the current state $S$:

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

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

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

The problem with this approach is that it is not very informative. In typical two-side protocols in which only two processes appear (sender and receiver for example) the range of values of this estimate is very small.

**Deadlocks as Formulae** In a deadlock state $S'$ all process are blocked, i.e.,

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

The predicate $blocked(i, pc_i(S'), S')$ is defined as: in the system state $S'$ the process $P_i$ is blocked in its local state $pc_i(S')$ (which is the program counter of $P_i$ in the deadlock state). This predicate is defined as

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

Given the deadlock formula we would like to implement deadlock detection using a directed search strategy with $H_{deadlock}$ as heuristic estimate function. Unfortunately, since we do not know the set of states in which the system will go into a deadlock in advance, we cannot compute the formula at exploration time.

**Approximating the Deadlock Formula** A first approach is to determine in which states a process can block. We call such states *dangerous*.

A process $P_i$ is blocked if $blocked(i, u, S)$ is valid for some $u \in C_i$, with $C_i$ being the set of dangerous states of $P_i$. We define $blocked(i, S)$ as a predicate for process $P_i$ to be blocked in system state $S$:

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

Therefore, we approximate the deadlock formula with

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

The search for deadlocks can be now performed with $H_{deadlock'}$ as an estimate.

Without designer intervention, all the 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.

## Experimental Results

All experimental results were produced on a SUN Workstation, UltraSPARC-II with 248 MHz. The SPIN comparison values were generated with version 3.3 of the validator. If nothing else is stated, the parameters while experimenting with SPIN and HSF-SPIN are as follows:

| Parameter | SPIN | HSF-SPIN |
|---|---|---|
| Partial Order Reduction | yes | no |
| Supertrace mode | no | no |
| Memory Limit | 512 MB | 512 MB |
| Depth bound | 10,000 | 10,000 |

We will present experimental results on the detection of deadlocks in three scalable protocols using the A* algorithm with different heuristic functions. The heuristic estimates we use are $H_{ap}$ and $H_{deadlock'}$. As indicated above, the approximation of the deadlock formula can be improved with hand-coded labels, which reduces the set of dangerous states (option $+U$). On the automatic side we experiment with dangerous states according to send operations (option $+S$), receive operations (option +R) or conditions (option $+C$).

Table 2 depicts experimental data for different instances of the dining philosophers problem. We tested A* with the following heuristic estimates: $H_{ap}$ is the activeness based heuristic, while $H_f$ is the formula based heuristic. When performing a breadth-first search (column $H = 0$) the number of expanded states grows exponentially, such that the 12-philosophers protocol is already intractable. SPIN is capable of solving all instances of the protocol with less than 16 philosophers, but offers long solution trails, e.g. 1,362 steps in the 8-philosophers problem, compared to the optimal number of 34. The results achieved by A* are superior: optimal solutions are found in all cases and the number of expanded states scales linearly. Moreover, for some heuristics the number of expansions matches the solution depth.

| $p$ | | $H = 0$ | $H_{ap}$ | $H_f + SRC$ | $H_f + R$ | SPIN |
|---|---|---|---|---|---|---|
| 2 | exp | 13 | 10 | 10 | 10 | 14 |
| | len | 10 | 10 | 10 | 10 | 10 |
| 3 | exp | 23 | 16 | 16 | 14 | 22 |
| | len | 14 | 14 | 14 | 14 | 18 |
| 4 | exp | 54 | 21 | 21 | 18 | 75 |
| | len | 18 | 18 | 18 | 18 | 54 |
| 8 | exp | 2,899 | 41 | 41 | 34 | 1,797 |
| | len | 34 | 34 | 34 | 34 | 1,362 |
| 12 | exp | - | 61 | 61 | 50 | 278,097 |
| | len | - | 50 | 50 | 50 | 9,998 |
| 16 | exp | - | 81 | 81 | 68 | - |
| | len | - | 68 | 68 | 68 | - |
| 25 | exp | - | 126 | 126 | 102 | - |
| | len | - | 102 | 102 | 102 | - |

Table 2: Number of expanded states and solution lengths achieved by A* in the dining philosophers protocol ($p$=number of philosophers).

Table 3 refers to experiments with the optical telegraph protocol and includes experimental results with best-first search (BF). The breadth-first search algorithm achieves optimal solutions, but it is unable to solve instances with more than 6 stations. A* finds optimal solutions, but in the most cases it does not scale well. With the formula-based heuristic A* is not capable of solving problems with more than 9 stations. Surprisingly, with an even number of stations A* with $H_{ap}$ is optimal, but with an odd number of stations the number of expanded states grows exponentially. SPIN scales linearly offering near-to-optimal solutions. This is due to the traversal order of depth-first search, which in this case is close to optimal. The best-first search algorithm with $H_{ap}$ the number of expanded nodes and the length of the solution trail match.

| $p$ | | $H=0$ | $H_{ap}$ | $H_f+U$ | $BF$ | SPIN |
|---|---|---|---|---|---|---|
| 2 | exp | 42 | 14 | 15 | 14 | 17 |
| | len | 14 | 14 | 14 | 14 | 16 |
| 3 | exp | 296 | 96 | 73 | 21 | 24 |
| | len | 21 | 21 | 21 | 21 | 23 |
| 4 | exp | 1,772 | 26 | 48 | 26 | 31 |
| | len | 26 | 26 | 26 | 26 | 30 |
| 5 | exp | 17,562 | 1,451 | 839 | 33 | 38 |
| | len | 33 | 33 | 33 | 33 | 37 |
| 6 | exp | 110,746 | 38 | 366 | 38 | 45 |
| | len | 38 | 38 | 38 | 38 | 44 |
| 7 | exp | - | 11,630 | 11,542 | 45 | 52 |
| | len | - | 45 | 45 | 45 | 51 |
| 8 | exp | - | 50 | 4,779 | 50 | 59 |
| | len | - | 50 | 50 | 50 | 58 |
| 9 | exp | - | - | - | 57 | 66 |
| | len | - | - | - | 57 | 65 |
| 10 | exp | - | 62 | 68,052 | 62 | 73 |
| | len | - | 62 | 62 | 62 | 72 |
| 11 | exp | - | - | - | 69 | 80 |
| | len | - | - | - | 69 | 79 |
| 12 | exp | - | 74 | - | 74 | 87 |
| | len | - | 74 | - | 74 | 86 |

Table 3: Number of expanded nodes and solution lengths achieved by A* the optical telegraph protocol ($p$=number of stations). In best-first search, $H_{ap}$ has been used as an estimate.

Tables 4 and 5 show experimental results for a *real-world* protocol: the CORBA General Inter Orb Protocol (Kamel & Leue 2000) (GIOP for brevity). The algorithm evaluated in Table 4 is best-first search. The solution lengths are substantially better than the solution lengths offered by SPIN except in the simplest configuration. Note that in some cases, SPIN's solutions are about 4 times larger. Table 5 shows the effect of applying WA* with $H_{ap}$ and different values of $w_h$ in the GIOP protocol. With larger values of $w_h$ WA* is able to solve larger problems, but with longer counterexamples. The extreme case is best-first search ($w_g = 0$): it scales best in the number of expanded nodes but one pays a price in terms of solution lengths.

| $p,q$ | | $H=0$ | $H_f+SRC$ | $H_f+SR$ | $H_f+U$ | SPIN |
|---|---|---|---|---|---|---|
| 1,1 | exp | 3,571 | 992 | 1,342 | 128 | 308 |
| | len | 54 | 61 | 61 | 55 | 60 |
| 1,2 | exp | 53,547 | 108 | 749 | 123 | 376 |
| | len | 60 | 61 | 95 | 61 | 135 |
| 1,3 | exp | - | 670 | 1,173 | 158 | 446 |
| | len | - | 79 | 86 | 79 | 210 |
| 2,4 | exp | - | 1,232 | 1,198 | 682 | 517 |
| | len | - | 113 | 97 | 82 | 391 |
| ... | ... | ... | ... | ... | ... | ... |

Table 4: Number of expanded states and solution lengths achieved by best-first in the GIOP protocol ($p$=clients,$q$=servers).

| $p,q$ | | $w_h=1$ | $w_h=4$ | $w_h=16$ | $w_h=32$ | $w_g=0$ |
|---|---|---|---|---|---|---|
| 1,1 | exp | 2,944 | 1,530 | 404 | 337 | 228 |
| | len | 54 | 54 | 54 | 54 | 72 |
| 1,2 | exp | 38,833 | 15,565 | 1478 | 1,234 | 314 |
| | len | 60 | 60 | 60 | 60 | 82 |
| 1,3 | exp | - | - | 22,843 | 9,342 | 452 |
| | len | - | - | 70 | 78 | 111 |
| 2,4 | exp | - | - | - | 47,634 | 416 |
| | len | - | - | - | 81 | 112 |
| 2,5 | exp | - | - | - | - | 421 |
| | len | - | - | - | - | 117 |

Table 5: Number of expanded nodes and counterexample length achieved by WA* with $H_{ap}$ and different values of $w_g$ and $w_h$ in the GIOP protocol ($p$=clients, $q$=servers). If $w_g$ or $w_h$ are not cited then their value is set to 1.

Table 6 depicts the results of searching the GIOP protocol with 1 server and 3 clients. Given 1 GByte of main memory A* can solve this problem with 545,141 node expansions. The minimal counterexample length is 70. With a limit of 60 MByte A* runs out of memory. The same bound yields a full transposition table in IDA* at depth 50 and a drastic increase in the number of expanded nodes, such that the time limit of one hour is exceeded in a few iterations. Bit-state hashing solves the problem in the optimal number of iterations according to the given time and memory bound. In the last iteration 5,752,690 elements were stored. The prior call of IDA* helps to avoid initialization time for the bit-vector but the difference in the number of expansion are very small, even though we provided twice as much memory.

## Related Work

There are two approaches that apply guided and heuristic search techniques to symbolic model checking:

*Validation with Guided Search of the State Space* uses BDD-based symbolic search of the Mur$\phi$ validation tool (Yang & Dill 1998). The best first search procedure incorporates symbolic information based on the Ham-

| len | IDA*+TT | A* | IDA*+BT | IDA*+BT+TT |
|---|---|---|---|---|
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 47 | 29,027 | 29,027 | 29,020 | 29,027 |
| 48 | 34,439 | 34,439 | 34,432 | 34,439 |
| 49 | 40,508 | 40,508 | 40,499 | 40,508 |
| 50 | 46,991 | - | 46,948 | 46,948 |
| 51 | 71,944 | - | 54,206 | 54,206 |
| 52 | 265,365 | - | 62,408 | 62,408 |
| 53 | 1,500,812 | - | 71,872 | 71,872 |
| 54 | - | - | 83,264 | 83,264 |
| 55 | - | - | 97,303 | 97,303 |
| 56 | - | - | 114,184 | 114,184 |
| 57 | - | - | 133,755 | 133,755 |
| 58 | - | - | 155,772 | 155,772 |
| 59 | - | - | 179,759 | 179,759 |
| 60 | - | - | 205,124 | 205,124 |
| 61 | - | - | 231,400 | 231,400 |
| 62 | - | - | 258,487 | 258,487 |
| 63 | - | - | 286,695 | 286,695 |
| 64 | - | - | 316,571 | 316,571 |
| 65 | - | - | 347,998 | 347,998 |
| 66 | - | - | 381,171 | 381,171 |
| 67 | - | - | 416,194 | 416,194 |
| 68 | - | - | 454,319 | 454,319 |
| 69 | - | - | 495,364 | 495,364 |
| 70 | - | - | 539,555 | 539,555 |

Table 6: The number of expansions with respect to different algorithms and increasing search depth in the GIOP protocol (1 server and 3 clients) with the $H_{ap}$ heuristic. TT abbreviates a transposition table and BT abbreviates a bit-state hash table.

ming distance between two states. The other considered options *target enlargement* (error state is searched backwards with bounded depth), *Tracks* (approximation of pre-images) and *Guideposts* (explicit hints) are not solution length approximations.

*Directed Model Checking* extends the $\mu$cke model checker (Biere 1997) with a symbolic variant of the A* algorithm (Reffel & Edelkamp 1999) in finding bugs of scalable hardware circuits like the tree-arbiter and the DME. The authors applied a backward, so-called refinement search to infer the estimate: The error specification is broadcasted within the circuit until a given refinement depth is reached.

For explicit state model checking the use of best-first exploration for protocol verification has been investigated in the PROVAT strategy (Lin, Chu, & Liu 1988). The system assumes that the two only available protocol operations are *send* and *receive*. The authors describe two different kinds of heuristics: global heuristics and move ordering heuristics. Both approaches split into three parts for different kinds of errors: unspecified reception, deadlocks and channel overflow. Global heuristics correspond to the ordering of the priority queue and in case of deadlock detection, the estimate is the weighted sum of the number of states that try to receive message from empty queues and the number of states that try to send a message to a full queue. For ordering the moves *receive* operations are considered first with ties broken in favor of the shortest queue. We have re-implemented the PROVAT strategy, but in the experiments the automatic inferred heuristics $H_{ap}$ and $H_{deadlock'}$ scale exponentially better.

Very recently symbolic model checking techniques have been reconciled with AI *planning* strategies. It has been argued that model checking and planning are identical (Giunchiglia & Traverso 1999). Communication protocols specify non-deterministic planning problems with resources. However, a single-state verification planner has not been developed although it has recently been shown that some standard LTL specifications can be parsed into conditional operators with quantified effects (Rintanen 2000). Our approach provides another bridge for the gap between AI-planning and verification. Promela can be considered an input language for non-deterministic planning problem in which channels and shared variables represent resources. A planning goal is encoded as a predicate that violates a certain assertion such that directed search methods apply.

## Conclusion and Future Work

We have shown that the efficiency of protocol validation based on model checking can greatly benefit from directed and heuristic search. Our approach centers around variants of the A* algorithm. We discussed weighted A*, iterative deepening and partial search variants of these algorithms. Next we introduced weighting functions to be used in heuristics for invariant verification, assertion violation and deadlock detection. We described the architecture of the HSF-SPIN tool which implements the heuristics that we described. We finally discussed the validation of two toy protocols and one real world protocol.

Our experiments show that the search heuristics we proposed can greatly increase the perfomance of model checking with respect to the number of explored states, and in particular with respect to the length of error witnesses. In the experiments, best-first search seems to be the best choice, since it generally scales well in the number of expanded states compared to the solution length, being near to optimal in many cases. When aiming at optimality, A* is effective, but when working with large protocols it has some drawbacks in performance due to the weakness of our heuristics. The activeness based estimate $H_{ap}$ is very simple and not very informative. It works well on protocols with many simple processes, but not in protocols with few complicated processes, as our experiments with the GIOP example indicate. The problem of the formula based heuristic for deadlocks is that its quality depends on designer aid to determine the set of dangerous processes. We expect that future research will greatly increase the efficiency of our heuristics. For instance, the values in the state distance matrices currently do not adequately reflect the number of receive operations needed to consume the current number of elements in the queue. Establishing good estimates for the actual number of transpositions without necessarily encountering a combinatorial explosion is an important research topic in the near future.

This study focuses on algorithms for detecting safety property violations. Liveness properties refer to paths of the state transition graph. The detection of liveness property violations entails searching for cycles in the state graph. This is typically achieved by an algorithm called nested depth-first search. In future work we will investigate how nested-depth first search can be improved with directed cycle-detection search. It has already been shown that nested-depth first search and partial order reductions can coexist (Holzmann, Peled, & Yannakakis 1996). It is not yet clear what kind of heuristic search algorithms could be used to accelerate the detection of liveness errors. We suspect that the directed approach can contribute to cycle detection by minimizing the distance for the state we started from. We currently re-implement partial order reduction algorithms to achieve speed up through directed search.

Since interfaces in HSF were designed to serve single-agent search problems, the main effort of implementing the new protocol validator was to alter the SPIN code generator to produce c++-source for a single-state problem representation to be compiled and executed with different search algorithms. This allows to implement new search algorithms, hash functions and estimates without accessing SPIN-source.

For the long term we will 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.

**Acknowledgments** Stefan Edelkamp and Alberto L. Lafuente are supported by the DFG project *Heuristic Search and its Application in Protocol Validation.*

## References

Biere, A. 1997. $\mu$cke - efficient $\mu$-calculus model checking. In *Computer Aided Verification*, 468–471.

Bryant, R. E. 1985. Symbolic manipulation of boolean functions using a graphical representation. In *DAC*, 688–694.

Burch, J. R.; M.Clarke, E.; McMillian, K. L.; and Hwang, J. 1992. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation* 98(2):142–170.

Clarke, E.; Grumberg, O.; and Peled, D. 2000. *Model Checking.* MIT Press.

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

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In *KI*, 81–92.

Edelkamp, S. 1999. *Data Structures and Learning Algorithms in State Space Search.* Ph.D. Dissertation, University of Freiburg. Infix.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Massachusetts: Addison Wesley.

Giunchiglia, F., and Traverso, P. 1999. Planning as model checking. In *ECP*, 1–19.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for heuristic determination of minimum path cost. *IEEE Trans. on SSC* 4:100.

Hernádvögyi, I. T., and Holte, R. C. 2000. The automatic creation of memory-based search heuristics. Submitted to AIJ special issue on heuristic search.

Hipke, C. A., and Schuierer, S. 1999. Vega—a user-centered approach to the distributed visualization of geometric algorithms. Technical Report 117, University of Freiburg.

Holzmann, G.; Peled, D.; and Yannakakis, M. 1996. On nested depth first search. In *The Spin Verification System*, 23–32. American Mathematical Society.

Holzmann, G. J. 1987. On limits and possibilities of automated protocol analysis. In Rudin, H., and West, C., eds., *Proc. 6th Int. Conf on Protocol Specification, Testing, and Verification.*

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

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

Junghanns, A. 1999. *Pushing the Limits: New Developments in Single-Agent Search.* Ph.D. Dissertation, University of Alberta.

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

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

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

McMillan, K. L. 1998. Symbolic model checking. In Inan, M. K., and Kurshan, R. P., eds., *Verification of Digital and Hybrid Systems*, 117–137. Springer.

Müller, R.; Eckerle, J.; and Ottmann, T. 1996. An environment for experiments and simulations in heuristic search. In Swiridow, A. P.; Widmayer, P.; Oberhoff, W.-D.; and Unger, H., eds., *New Media for Education and Training in Computer Science*, 130 – 139. Infix.

Peled, D. 1998. Partial order reductions. In Inan, M. K., and Kurshan, R. P., eds., *Verification of Digital and Hybrid Systems*, 117–137. Springer.

Reffel, F., and Edelkamp, S. 1999. Error detection with directed symbolic model checking. In *FM*, 195–211. Springer.

Reinefeld, A., and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701–710.

Rintanen, J. 2000. Incorporation of temporal logic control into plan operators. In *ECAI*, 526–530.

Taylor, L. A., and Korf, R. E. 1993. Pruning duplicate nodes in depth-first search. In *AAAI*, 756–761.

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