Protocol Verification with Heuristic Search: First Results

Stefan Edelkamp Alberto Lluch Lafuente
Stefan Leue
Institut für Informatik
Albert-Ludwigs-Universität
Georges-Köhler-Allee
D-79110 Freiburg
eMail: {edelkamp,lafuente,leue}@informatik.uni-freiburg.de

October 9, 2000

Abstract

In this paper we present the new protocol validator HSF-SPIN that incorporates directed search for checking safety properties. Promela specifications are parsed into c++-code to serve as a general single-state problem with start, goal, expand and estimator function to be solved with different heuristic search algorithms. The basic idea is to accelerate the search process into the direction of a specified goal situation, which in the case of protocol verification is the set of errors. The explored part will be smaller than with classical approaches, allowing larger validations. If the estimate is a lower bound, optimal solutions will be found.

The estimates exploit the representation of the protocol as communicating processes with queues and finite state machines. We evaluate the impact of various heuristics in reducing the search tree on scalable benchmark protocols. We also present a new search paradigm based on the combination of partial and heuristic search.

1 Introduction

Concurrent software systems [36] are erroneous and inherently complex to debug. Therefore, formal verification techniques [32] have been developed to automatically detect faults in the program specification. In this paper the focus is on the verification of communication protocols in searching assertion violations and deadlocks [45]. A communication protocol [19] itself is viewed as a collection of processes with one finite state machines for each party of the protocol.

Communication between two processes are either realized with queues or with global variables. Sending or receiving a message is an event that causes a state transition, which in turn corresponds to a line of code in a given implementation. The state space is represented as the cross product of the FSMs plus the information for the communication load.

Formal verification techniques would be useless if we always design correct protocols. However, in practice even involved work in protocol specification usually contains several bugs. This is due to the non-deterministic structure of concurrent communication processes that inherently call for a combinatorial explosion. Therefore, simulating protocol execution on real-world data usually covers only a small fraction of the entire system.

Critical safety and liveness properties can be addressed as temporal formulae based on a Kripke structure such that protocol verification is in fact a model checking problem. Model checking techniques branches into two different approaches: The first one, called symbolic model checking [37], uses binary decision diagrams (BDDs [5]) to efficiently represent huge sets of states. The second one represents states explicitly and applies various single state space techniques such as partial order reduction [41] and partial search [25] to efficiently traverse the search tree of the product automaton. The latter origins in linear time logic (LTL [40]), while the former is rooted in branching time logic (CTL [38]). However, the differences in the temporal logics are not crucial. Both are contained in CTL* which in turn is part of the μ-calculus [48]. When interested in counterexamples both single state and symbolic exploration apply. Various verification tools for concurrent systems have been developed in the last decade, most notably the SPIN validator written by Holzmann at Bell Labs. The single state space approach in the tool opposes symbolic and theorem proving systems like PVS [39], VIS [3], SMV [35], μcke [1], and *bounded model checking* [2].

SPIN parses the expressive concurrent Promela protocol specification language and (usually) applies the supertrace algorithm and sequential hashing to examine various beams in the search trees up to a certain threshold depth [24]. It applies partial order reduction [41] and bit-state hashing[25] compresses the state description of several hundred bits down to only a few bits to build a hash table with 2^{20} entries and more. The search algorithm is not complete since not all synonyms are disambiguated. Moreover, through depth-first traversal, the length of a witness of an error state is not minimal. However, SPIN succeeds in finding bugs in large depths. This is due to the fact that the density of errors in the cross product state space is quite high compared to single agent search problems, which usually have only one final state.

We choose SPIN as a basis for our implementation and provide an interface for AI search methods that has ever since addressed huge problem spaces. The fundamental search algorithm A* [20] is a variant of Dijkstra's single source shortest path algorithms with re-weighted edges according to additional search information in form of estimation functions [10]. Current heuristic search techniques such as iterative deepening [27], transposition tables [43], finite state ma-

chine pruning [46], relevance cuts [26], symbolic representation [16] and memory-based heuristics [21] effectively solve problems with 10^{20} states and more, which compares with performance gains in symbolic model checking [6].

The state space reduction in heuristic search algorithms depends on the quality of the heuristic estimate. We have developed different heuristics for these purposes based on communication queue load, process activeness and distances to dangerous FSM states from current ones.

The article is structured as follows: First we take a closer look on communication protocols and their representations in Promela. Then we address automated protocol verification in the new validator HSF-SPIN and implementation issues of HSF with its general problem interface. Afterwards we present different search heuristic search algorithms and different heuristics and their impact is evaluated on two simple but scalable protocols.

As an outlook we present the conception of a protocol workbench, that integrates protocol design, parsing, validation and visualization. We use different visualization front-ends for the verification tool such as VIP, XSPIN and VEGA.

2 Communication Protocols

2.1 Concurrent Systems

Concurrent systems have been formalized as finite state systems $\mathcal{F}$ together with a finite set of propositions and a state labeling function L [41]. $\mathcal{F}$ is characterized by a finite set of states S, a set of deterministic transactions T, and an initial state s_0. The mapping T is partial such that an action t can be executed in s only if the set of preconditions $pre(a)$ are fulfilled. The sets T, S and L can be used to define formal specification languages for transition, state, and propositional sequences to determine which of the generated labeled paths satisfy a given formula. Concurrent systems can formalize the combined behavior of a finite number of interacting sequential processes. For protocol verification we prefer labeled transactions to refer to the process statement that is represented by the transaction [22]. Process numbers denote the sequential process that contains a specific statement.

2.2 Protocol Specification with Promela

A protocol in Promela is defined by a set of processes. A process consists of statements on a set of local and global variables. Furthermore, sending and receiving messages via communication queues are supported. Promela provides a suitable specification language for concurrent systems.

To exemplify the notation we use the well-known dining philosopher problem posed by Dijkstra in 1965. We are confronted with n philosophers sitting around a table and trying to have a lunch time between meditations. Each philosopher

has each own meal (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. There is a need for a protocol to get and release the forks. The first obvious solution is following. 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 its left fork there is no further progress possible for requiring the other one. There are of course deadlock-free and fair solutions to this solution of the dining philosopher solution is given in Table 1.

```
#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 philosopher problem specification in Promela.

The problem has been implemented in a pure message-passing form. The

set of forks is represented as an array `forks` of queues of size 1. The only type of message is to access or release a fork. For a philosopher i, `forks[i]` represents the left fork and `forks[(i+1)%MAX_PHILOSOPHERS)]` represents the right one. A philosopher is modeled as a process (`proctype` in Promela) that performs an endless loop. To grab a fork a philosopher tries to receive it from the corresponding queue while releasing a fork corresponds to send a message. Receive and send operations are expressed as $queue?message$ and $queue!message$, respectively. Like all other Promela statements, these operations can block a process. The `init` process, implemented as an `atomic` statement, places all forks on the table and *runs* the philosophers.

3 The Validator HSF-SPIN

HSF-SPIN merges the SPIN validation tool to inherit most of the efficiency and functionality of Holzmann's original source and with the Heuristic Search Framework (HSF [13]) for single-agent exploration problems.

For this aim we refined the state description to incorporate solution length information, transition labels and predecessors for solution extraction. We newly implemented universal hashing, and provide an interface consisting of a node expansion function, initial and goal specification and, in order to direct the search, we realized different heuristic estimates. HSF-SPIN also writes trail information to be visualized in the XSPIN interface. Since SPIN is a widely distributed software package[1] we briefly present the HSF architecture.

3.1 HSF

HSF is an efficient `c++` workbench to define and to solve challenging single agent games like the (n^2-1)-Puzzle [30], Sokoban [26], and Rubik's Cube [28]. It further includes implementations for a generalization to the (n^2-1)-Puzzle and some other intriguing tile and cube domains.

HSF provides classical and extended AI search strategies and includes different FSM duplicate pruning schemes. Since duplicate elimination is supposed to support only well-structured problems for our proposes we omit this feature from the system. It is designed to implement both new state space problems and new search algorithms, combining the workbench design pattern of object-oriented programming [17] with efficiency aspects found in implementations of A* and IDA* for specific puzzles.

We characterize the A* algorithm in an unusual but concise way 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 [10]. 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

[1] `http://netlib.bell-labs.com/netlib/spin/whatispin.html`

node u the algorithm always selects a node from the priority queue (the search horizon) 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.

Algorithm A* accommodates the information of the *heuristic* $h(u)$, which estimates the minimum cost of a path from node u to a goal node in T. A heuristic h is called *consistent* if and only if $w(u,v) - h(u) + h(v) \geq 0$ for all u and v. It is called *optimistic* if it is a lower bound function. A* can be cast as a search through a re-weighted graph. More precisely, the edge weights w are replaced by new weights $\hat{w}$ by adding the heuristic difference: $\hat{w}(u,v) = w(u,v) - h(u) + h(v)$. At each instant of time in the re-weighted Dijkstra algorithm, the merit f of a node u is the sum of the new weights along the currently cheapest path explored by the algorithm.

By this transformation, negative weights can be introduced. Nodes that have already been expanded might be encountered on a shorter path. Thus, contrary to Dijkstra's algorithm, A* deals with them by possibly re-inserting nodes from set of already expanded nodes into the set of horison nodes (re-opening).

On every path p from s to u the accumulated weights in the two graph structures differ by $h(s)$ and $h(u)$ only, i.e., $w(p) = \hat{w}(p) - h(u) + h(s)$. 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.

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 tracing the solution path we additionally store the predecessor and the transition from the predecessor to the state. The lists of expanded and generated states are stored in a table and a hash function spreads packed states to the available table indices. Hash conflicts are resolved by chaining.

The directed exploration algorithms use the pre-specified abstract class interface of a single agent search problem as presented in Table 2.

Similarily, all search methods and state space problems are specified by virtual methods. For the former this is the method `search`. In A* (cf. Table 3) this function takes the initial state and expands one states one after another until the priority queue gets empty or a goal state is found. The updates in the queue and the visited list are very much like Dijkstra's single source shortest path algorithm [12] with the difference that the priorities in the queue are ordered according to the sum of the current path length and the heuristic estimate.

The implementation of the priority queue as used in A* is as follows [11]. Since both the generating path length and the heuristic estimate are integer values and the priorities are bounded by a constant, the states can be kept in doubly-linked lists stored in buckets according to their priorities. Therefore, given a node reference `insert` and `close` can be executed in constant time while the operation `deletemin` increases the bucket index for the next node to be expanded. If the differences of the priorities/estimates of successive nodes

```
(01)              class SearchProblem {
(02)                public:
(03)                  SearchProblem();
(04)                  virtual heuristic (State* S) = 0;
(05)                  virtual goal (State* S) = 0;
(06)                  virtual expand(State* S, State** Succs) = 0;
(07)                  virtual input (State* S) = 0;
(08)                  virtual vega(State* S) = 0;
(09)              }
```

Table 2: Abstract problem representation in HSF. `input` defines the initial situation and `goal` determines if the exploration is completed. `expand` generates the set of successors and `heuristic` approximates the distance to the goal. Last but not least `vega` defines the interface to the Java visualization system.

```
(01)          State* search(State* S) {
(02)            ht.insert(S);
(03)            pq.insert(S,sp.heuristic(S));
(04)            while (!pq.empty()) {
(05)                u = pq.deletemin();
(06)                if (sp.goal(u)) return u;
(07)                sp.expand(u, succs);
(08)                for(v = succs; v; v = v->next) {
(09)                  found = ht.search(v->ps);
(10)                  if (!found) {
(11)                    pq.insert(v, v->f);
(12)                    ht.insert(v);
(13)                  }
(14)                  else {
(15)                    if (found->f > v->f) {
(16)                      pq.close(found, found->f);
(17)                      pq.insert(v, v->f);
(18)                      ht.insert(v)
(19)                    }
(20)                  }
(21)                }
(22)              }
(23)            }
```

Table 3: A* as implemented in HSF: `S`,`u`,`v` and `found` are states, `ht` is the hash table, `pq` is the priority queue and `sp` corresponds to the search problem. Reopening assumes that `pq.close` is trivial if `found` is not in `pq` and the a value is always inserted to the front of the hash table list.

are bounded by a constant, `deletemin` runs in $O(1)$, too.

The iterative deepening variant of A* (cf. Table 4) expands all nodes in the search until the next horizon value has been reached. Note that Line (16) can be refined to "`if (found->f <= v->f && f->pred != v->pred)`", such that synonymous nodes with the same f-value are only expanded if they are encountered on the same path.

Since interfaces in HSF were designed to serve single-agent search problems, the core 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. Separating the search algorithm from the problem definition is probably the most important contribution. Now we can implement new search algorithms, hash functions and solution length approximations without accessing SPIN-source.

3.2 Search Algorithms

We assume an uniformly weighted problem graph, such that each state transition contributes 1 to the overall solution length. HSF-SPIN allows textual simulation to interactively search the space. On each expansion the user is prompted which successor state to take. This allows to debug an error once it has been found.

In HSF-SPIN we modified the static representation of states in HSF to allow dynamic state vector allocation, since the predefined maximum size of the vector is too pessimistic in general. We concentrate on the changes according to the combination of partial and heuristic search.

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, since the information of generating path length and predecessor path length might be totally wrong. Subsequently, we omit reopening in both A* and IDA*. In A* "`pq.insert(v,v->f)`" in Line (17) is changed to "`pq.insert(found,v->f)`" such that "`ht.insert(v)`" in Line (18) is dropped. In IDA* we simply omit to reconsider the node found in the hash table (Line (16) and (18) in the implementation of Table 4). This further accelerates the search and no memory for storing the f-value is required. 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: $f(u) = g(u) + h(u) \leq g(v) - 1 + h(v) + 1 = f(v)$. Fortunately, most heuristic functions satisfy this criterion.

The lack of state space coverage in partial search is usually compensated by repeating the search with restarts on different hash functions. We implemented a set of universal hash functions [10] from which the current one is randomly chosen.

```
(01)            State* search(State* S) {
(02)              thresh = sp.heuristic(S);
(03)              do {
(04)                fstar = maxf;
(05)                solution = step(thresh,S);
(06)                thresh = fstar;
(07)              } while (!solution);
(08)              return solution;
(09)            }
(10)            State* step(int thresh, State* u) {
(11)              sp.expand(u, succs);
(12)              for(v = succs; v; v = v->next) {
(13)                if (sp.goal(v)) return v;
(14)                found = tt.search(v->ps);
(15)                if (found) {
(16)                  if (found->f < v->f)
(17)                    continue;
(18)                  found->f = v->f;
(19)                }
(20)                else
(21)                  if (tt.size() < max)
(22)                    tt.insert(v);
(23)                if (v->f > thresh)  {
(24)                  if (v->f < fstar)
(25)                    fstar = v->f;
(26)                }
(27)                else {
(28)                  found = step(thresh,found);
(29)                  if (found) return found;
(30)                }
(31)              }
(32)            }
```

Table 4: IDA* with a transposition table in HSF: `S`,`solution`,`u`,`v` and `found` are states, `tt` is the hash table, `max` its capacity, `maxf` the maximum f-value, `thresh` and `fstar` the current and next threshold value and `sp` corresponds to the search problem.

3.3 A* with Bit-State Hashing

For the bit-state version of A* we omit 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 is necessary since the access to the priority queue is almost impossible to predict and through the uncorrelated access on the horizon once the goal (e.g. the deadlock) is found, the solution path (e.g. the witness) cannot be generated.

Bit-state hashing reduces the memory consumption of A* if the state description length is large and the ratio of horizon list and the visited list is small; two objectives met in validating large protocols.

3.4 IDA* with Bit-State Hashing

Since 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 $p(S)$. A state S is stored by setting the bit at $p(S)$ and searched by querying $p(S)$.

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 exanded in two successive iterations.

3.5 Heuristics for Deadlock Detection

For the sake of conciseness in the following we concentrate on heuristics for deadlock detection. A deadlock arises if all processes of the protocol are blocked, i.e., no transition can be executed. The only exception is if all processes are in a valid end-state.

Blind Search In this case the estimate h_0 is always zero and a trivial lower bound. Let g be the generating path length then A* with merit $f = g+h$ reduces to breadth-first search. A complete exploration is but the only possibility to verify that no error is contained in the system.

Non-Empty-Queues For this very simple heuristic we simply count the number of non-empty-queues in the current state S:

$$h_1(S) = \sum_{q \in Q,\ q \text{ not empty in } S} 1$$

with Q being the set of queues. This contributes to the observation that in several problems like the problem of the dining philosophers deadlocks were given when all communication queues indicate an underflow. An analogous deadlock situation for sending information into the channel is given if all queues are full and signal an overflow. This estimate is not a lower bound in general.

Active Processes An intuitive idea is to count the number of active (or non-blocked) processes in the current state S:

$$h_2(S) = \sum_{p \in P,\ p \text{ active in } S} 1$$

with P being the set of processes. This estimate is a lower bound for all protocols except for those in which rendezvous communication[2] is used, e.g. if after a rendezvous of two processes, both block. The problem with this approach is that it is not very informative.

Local Distances to a Dangerous State In this lower bound technique for each pair of states of each finite state machine M we pre-compile a matrix $D_M = (d_{ij})$, where d_{ij} denotes the minimal number of transitions from state i to state j in M. Each matrix D_M is computed with the all-pairs shortest-path algorithm of Floyd/Warshall in $O(|M|^3)$ time and $O(|M|^2)$ space [10]. Note that $|M|$ is very small in comparison to the overall search space.

To calculate the estimate in the search of the overall state space we first determine the set of dangerous states in each FSM. A dangerous state is a state from which all transitions are dangerous. A dangerous transition in turn is a transtition which is not always executable. For example, a transition representing an assignment is not dangerous but transitions representing operations over queues or conditions over variables are not always executable. If all transitions from a state are dangerous, it is possible that no transition is executable and that the corresponding process is blocked. The idea of the estimate h_3 is that in a deadlock, each FSM will be blocked in a dangerous state. Let P be the set of processes, $M(p)$ the FSM according to p, $s(p)$ be the current state in $M(p)$, and $C(p)$ the set of dangerous states in p, with a send or receive operation. Then

$$h_3(S) = \sum_{p \in P} \min_{g \in C(p)} d_{M(p)}(s(p), g).$$

The crucial point is that the minimum distances to the set of dangerous states in each FSM are independent (each FSM has to reach its dangerous state) such that the sum of the distances still serves as a lower bound estimate.

[2] In rendezvous communication two processes perform a transition at the same time

Local Information of Dangerous Queues A dangerous transition is a transition that cannot always be executed. For the sake of simplicity we work only with message-passing protocols, in which only send and receive operations can block a process. Therefore, only transitions representing such operations are considered as dangerous. A receive operation $queue?message$ can block if there is no message of the specified type in the queue. A send operation is executable only if the queue is not full. Let $|q|$ be the length of the queue and $q_{\max}$ be its maximum. To block a receive operation there is a need of $|q|$ steps. For a send operation $q_{\max} - |q|$ steps are necessary. Let $Q(p, s)$ be the set of queues that are accessed by process p in dangerous state s with either a receive $(-)$ or a send $(+)$ message then the refined heuristic h_4 is formalized as follows:

$$h_4(S) = \sum_{p \in P} \min_{g \in C(p)} \left\{ \sum_{q \in Q^-(p,g)} |q| + \sum_{q \in Q^+(p,g)} q_{\max} - |q| \right\}.$$

Combining Dangerous State Distance and Queue Information The following estimate h_5 includes the estimate information of h_3 and h_4.

$$h_5(S) = \sum_{p \in P} \left\{ d_{M(p)}(s(p), g) + \min_{g \in C(p)} \left\{ \sum_{q \in Q^-(p,g)} |q| + \sum_{q \in Q^+(p,g)} q_{\max} - |q| \right\} \right\}$$

Estimates h_4 and h_5 are not lower bounds, since when computing the local bounds in many processes the impact of a queue may be added several times.

Global Information The definition of dangerous state can be improved, including the requirement that the executability of a set of dangerous transitions in different FSM can always be executed: Not all combinations of dangerous states of each FSM are dangerous. The reason is that the disjunction of the conditions of a set of dangeorus transitions has to be true.

Therefore, the executability of a set of dangerous transitions is defined as the disjunction of all conditions associated to each transition. For example if the set of transitions is $T = \{q?i, q!j\}$ (transitions are represented by their operations), the executability of T is defined as

$$\bigvee_{t \in T} cond(t) = cond(q?i) \vee cond(q!j) = \neg empty(q) \vee \neg full(q) \equiv 1.$$

The space of all combinations of transition is usually too large to be searched completely. Therefore, to infer an informative heuristic we group some FSMs to get a partition of the search space not too large, but informative enough to detect inter-correlations of the FSMs. The estimates h_6, h_7, and h_8 of h_3, h_4, and h_5 are respective refinements according to this observation.

4 Experiments

All experimental results were produced on a sun Workstation, UltraSPARC-II CPU with 248 MHz.

4.1 Dining Philosopher

In the dining philosopher problem the heuristic evaluation functions have an exponential impact on the search. Table 5 depicts the results with respect to the different heuristics.

We used a space limit of about 500 MByte and a time limit of about 10 minutes. The A* algorithm with h_0, h_3 and h_4 runs out of space first while with h_6, h_7 and h_8 it runs out of time first. To get an impression on the time and space complexity we exemplarily choose the 16-philosopher problem. A* with h_1 generated 1940056 nodes of which 769892 were distinct. The space consumption was about 500 MByte and the total search time about 2 minutes.

Obviously, only h_2 and h_5 scale linearly for exponential savings in the exploration such that even minimal witnesses for deadlocks in very high depth can be detected. In h_2 the optimal number of nodes are expanded while A* with h_5 expands about 1/4 nodes more than necessary. Solving the 200-philosopher with h_2/h_5 generates 20902/40604 distinct nodes. In both cases the allocated space is about 100 MByte with a total search time of about 15 seconds.

Table 6 shows the effect of increasing the influence of the heuristic estimator in the philosopher problem. Instead of using the priority $f = g + h$ in A*, the derivate WA* applies $f = g + \delta h$ with $\delta > 1$. We experimented with $\delta \in \{2, 3, 4, 5, 10\}$. Since the heuristic is not longer a lower bound, the solution length will not be optimal. However, the search space can be significantly reduced. For the dining philosopher protocol we loose at most 4 transitions in the solution length and for larger values of n we observe a similar scaling phenomena as above: For each value of δ the number of expanded nodes are only a constant number larger than the solution depth; 1067 for $\delta = 2$, 266 for $\delta = 3$, 249 for $\delta = 4$, 173 for $\delta = 5$, and 408 for $\delta = 10$.

In Table 7 we have depicted the effect of changing the algorithm in the philosopher problem (accoding to h_1 and $\delta = 1$). We used A* with a hashtable of 510509 entries (corresponding to about 500 MByte) and IDA* with transposition table of 100 thousand entries (about 100 MByte).

Further on, we applied the bit-state hashing variant of A* with horizon-list and a visited list of 510509 entries. IDA* with bit-state hashing was invoked with a bit-array of 10 million entries (corresponding to about 1.2 MByte).

Since we have 4 states in each FSM the state space of the dining philosopher is larger than 4^n. Note that even if assisting the original SPIN tool with the optimal solution length, using the supertrace algorithm and letting it run over our time limit, it cannot solve large instances ($n > 20$) to this problem. It

n	l	h_0, h_3, h_4	h_1	h_2	h_5	h_6	h_7, h_8	l_{spin}	t_{spin}
2	10	13	11	10	10	11	10	11	14
3	14	23	16	14	16	15	14	18	22
4	18	54	23	18	21	21	21	54	75
5	22	124	34	22	26	38	27	66	80
6	26	356	55	26	31	54	32	362	402
7	30	946	100	30	36	78	37	330	422
8	34	2899	203	34	41	102	42	1362	1797
9	38	8018	446	38	46	428	47	1466	1915
10	42	24590	1027	42	51	962	143	94222	12407
11	46	69429	2424	46	56	2200	216	9382	12314
12	50	212033	5791	50	61	-	-	46822	69383
13	54	-	13914	54	66	-	-	52744	69386
14	58	-	33519	58	71	-	-	162032	213253
15	62	-	80844	62	76	-	-	248930	327555
16	66	-	195091	66	81	-	-	914398	1.203×10^6
17	70	-	-	70	86	-	-	1.574×10^6	2.071×10^6
18	74	-	-	⋮	⋮	-	-	8.506×10^6	1.180×10^7
19	78	-	-	⋮	⋮	-	-	-	-
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
50	202	-	-	202	251	-	-	-	-
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
100	402	-	-	402	501	-	-	-	-
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
200	802	-	-	802	1001	-	-	-	-
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 5: Counterexample path length l and number of expanded nodes to compute the minimal witnesses for the deadlock the scalable dining n-philosophers problem. The results are compared with the solution depth l_{spin} and the number of transitions t_{spin} when using SPIN.

n	l	$g+h_1$	$g+2h_1$	$g+3h_1$	$g+4h_1$	$g+5h_1$	$g+10h_1$
2	6*10	11	11	11	11	11	11
3	6*14	16	16	16	16	16	16
4	6*18	23	23	27	27	31	39
5	6*22	34	42	61	69	104	141
6	2*26,4*30	55	93	199	189	158	230
7	2*30,4*34	100	267	277	260	196	324
8	1*34,5*38	203	866	302	285	210	392
9	1*38,5*42	446	1086	308	291	215	437
10	1*42,5*46	1027	1113	312	295	219	453
11	1*46,5*50	2424	1117	316	299	223	458
12	1*50,5*54	5791	1121	320	303	227	462
13	1*54,5*58	13914	1125	324	307	231	466
14	1*58,5*62	33519	1129	328	311	235	470
15	1*62,5*66	80844	1133	332	315	239	474
16	1*66,5*70	195091	1137	336	319	243	478
17	-,5*74	-	1141	340	323	247	482
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
50	-,5*206	-	1273	472	455	379	614
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
100	-,5*406	-	1473	672	655	579	814
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
200	-,5*806	-	1873	1072	1055	979	1214
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 6: Number of expanded nodes and solution lengths in the dining philosopher problem for WA* according to different weightings.

n	l	A*	IDA*	A* bit-state	IDA* bit-state
2	10	11	47	11	47
3	14	17	82	17	82
4	18	26	128	26	128
5	22	42	187	42	187
6	26	75	265	75	256
7	30	149	376	149	376
8	34	322	554	322	554
9	38	734	881	734	881
10	42	1723	1555	1719	1554
11	46	4105	3054	4086	3050
12	50	9850	6532	9751	6523
13	54	23714	14775	23182	14717
14	58	57179	34509	533697	34110
15	62	137965	-	109675	79622
16	66	-	-	-	182683
17	70	-	-	-	398440
18	74	-	-	-	781490
19	78	-	-	-	1226536
20	82	-	-	-	1484648

Table 7: Number of expanded nodes and solution lengths in the dining philosopher problem for different heuristic search algorithms.

simply finds no solution. Moreover, as Table 7 indicates, SPIN finds solutions at untractable depths.

4.2 Optical Telegraph

The optical telegraph protocol ([25]) is a further scalable benchmark problem that contains a deadlock. Each of the n station has two control units with access to channels *up* and *down* for writing and receiving message from other stations that are arranged in a ring. Only one unit in each station can operate at a time. The Promela specification of the protocol is depicted in Table 8. It has been rewritten in a pure message-passing form.

Tables 9 and 10 show the solution length and the number of expanded nodes according to different heuristics as in the previous example. Searching for optimal or near solutions with breadth-first search and the other implemented heuristic estimates results in thrashing computational resources.

In contrast to the dining philosopher example SPIN scales very good in this problem. We assume that the traversal ordering in SPIN is close to optimal. However, the achieved solution lengths are not optimal. Applying WA* in h_2 with different δ values as depicted in Table 10 also allows to find the deadlock even for a large number of stations. We observe that h_2 with $\delta = 2$ scales very good and yield optimal solutions. Larger values of δ leads to solutions that are not optimal. Surprisingly, all δ-values larger than 3 result in the same number of expansion and solution length.

Table 11 shows the effect of different heurisic search algorithms in this domain (with h_2 and $\delta = 1$). We used the same memory sizes as in the dining philosophers example. As expected A* runs out of space, while IDA* runs out of time. In the final iterations of IDA* with bit-state hashing the number of expansions are comparably small: 14, 21, 42, 49, 622, 629, and 19511.

5 Current and Future Work

Partial-order reduction as implemented in SPIN [22] is probably the most important option to reduce search space to representives of equivalence classes of permuted independent statements. We currently re-implement partial-order as given in SPIN to significantly speed up the directed search methods. Furthermore, we will investigate, how nested-depth first search can be improved with a directed cycle-detection search. It has already been shown that nested-depth first search and partial-order methods can cooperate [23].

For the long term we will develop an integrated protocol definition and validation system based on Promela specifications and integrate VIP, HSF-SPIN and VEGA; briefly reviewed in the following.

```
#define NSTATIONS 4
#define true 1
#define false 0
chan station_control[NSTATIONS] = [1] of {byte};
chan up[NSTATIONS] = [1] of { byte };
chan down[NSTATIONS] = [1] of { byte };
mtype = { start, attention, data, stop, control }
proctype station(byte id; chan in, out)
{
        do
        :: in?start ->
                station_control[id]?control;
                out!attention;
                do
                ::in?data -> out!data
                ::in?stop -> break
                od;
                out!stop;
                station_control[id]!control
        :: station_control[id]?control;
                out!start;
                in?attention;
                do
                ::out!data -> in?data
                ::out!stop -> break
                od;
                in?stop;
                station_control[id]!control
        od
}
init
{
        byte nstations=NSTATIONS;
        atomic{
                do
                ::nstations>0 ->
                        nstations--;
                        station_control[nstations]!control;
                        run station(nstations, down[nstations], up[nstations]);
                        run station(nstations,
                                    up[(nstations+1)%NSTATIONS],
                                    down[(nstations+1)%NSTATIONS])
                ::nstations==0 -> break
                od
        }
}
```

Table 8: The optical telegraph problem implemented in Promela.

n	l	h_0, h_3, h_4	h_1	h_2	h_5	h_6	h_7, h_8	l_{spin}	t_{spin}
2	14	42	46+2	15	45+2	33	14	16	17
4	26	1772	3063+4	168	2238+4	753	67	30	31
6	38	110746	278787+6	5198	142329+6	-	2414	44	45
8	50	-	-	-	-	-	-	58	59

Table 9: Counterexample path length l and number of expanded to compute the minimal witnesses for the deadlock the scalable optical telegraph problem with n units. The results are compared with the solution depth l_{spin} and the number of transitions t_{spin} when using SPIN.

n	l	$g + h_2$	$g + 2h_2$	$g + 3h_2$	$g + 4h_2$	$g + 5h_2$	$g + 10h_1$
2	6*14	15	15	15	15	15	15
4	2*26,4*36	168	43	34	34	34	34
6	2*38,4*42	5198	74	57	57	57	57
8	1*50,4*56	-	111	84	84	84	84
10	1*62,4*70	-	154	115	115	115	115
12	1*74,4*84	-	203	150	150	150	150
14	1*86,4*98	-	258	189	189	189	189
16	1*98,4*112	-	319	232	232	232	132
18	1*110,4*126	-	386	279	279	279	179
20	1*122,4*140	-	459	330	330	330	330
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
30	1*182,4*210	-	914	645	645	645	645
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
40	1*242,4*280	-	1519	1060	1060	1060	1060
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 10: Number of expanded nodes and solution lengths in the optical telegraph problem for WA* according to different weightings.

n	l	A*	IDA*	A* bit-state	IDA* bit-state
2	14	15	22	15	93
3	21	94	189	94	298
4	26	164	518	164	480
5	33	2451	6253	2448	4171
6	38	5151	18113	5145	8377
7	45	77724	-	76246	121875
8	50	174996	-	172314	264137

Table 11: Number of expanded nodes and solution lengths in the optical telegraph problem for different heuristic search algorithms.

5.1 VIP

The Visual Interface to Promela (VIP) supports visual editing and maintenance of v-Promela models, which in turn is a visual, object-oriented extension to Promela [33]. VIP provides Promela code generation. The objective of this research is to reconcile Promela with state-of-the-art visual modeling techniques for real-time systems, in particular UML-RT, and to provide suitable tool support. VIP allows for the visual editing and maintenance of v-Promela models and provides a Promela code compiler. Later implementations will include result interpretation, support for dynamic capsule structures and object-oriented concepts at behavior and structure level.

5.2 VEGA

The project VEGA [4] offers a distributed, convenient, and flexible environment for visualization. The base is a powerful client/server architecture where the algorithms themselves may run on a UNIX server. VEGA offers distributed visualization and algorithms may be examined without need for software installation. A small bandwidth communication interface guarantees good performance even in slow networks. The algorithm source code is safe, because it is executed on the server. With the VEGA C++ class library adapting existing C/C++ algorithms to VEGA is made easy. The VEGA client interactively draws, loads, saves, and modifies scenes of geometric objects. It can be used to visualize algorithms on-line or to show saved runs off-line. The algorithm visualization may be adjusted to user preferences with view attributes. VEGA offers xfig and GIF export in addition to its own very simple file format.

VEGA is thought of a Web-Interface to support finite state machine and message sequence chart visualization as in given in XSPIN and to merge its facets with the VIP tool.

6 Related Work

Heuristic search was applied to the area of protocol verification by different researchers. On the symbolic validation side there are two approaches known to the authors.

Validation with Guided Search of the State Space [47] uses BDD-based symbolic search of the Murϕ validation tool. The best first search procedure incorporates symbolic information based on the Hamming 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 μcke model checker with a symbolic variant of the A* algorithm [42] 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 single-agent search there is one piece of work integrating A* like exploration for protocol verification entitled the PROVAT strategy [34]. The system assumes that the two only available protocol operations are *send* and *receive*. The authors describe two different kinds of heuristics: global heuristic 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 of the horizon nodes 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 from 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 to the shortest queue.

The best result on the X.21 protocol was found on a channel size of 4. PROVAT expands 3.4 % of the nodes of exhaustive search to locate *all* deadlock states. Unfortunately, due to the development of partial and symbolic search methods this research branch was not pursued for a long time.

Omitting the predecessor information in the A* algorithm for the bit-state version seems possible. Bidirectional and frontier divide-and-conquer search methods [29, 31] bypass memory requirements for the visited list and search with the horizon list exclusively. In order to retrieve the solution path in the exploration, the search process is invoked from both the init and the goal state until an intermediate state is found, on which the problem is divided.

Very recently symbolic model checking techniques are encountering AI *planning*, as the different aspects *traditional planning* [15], *non-deterministic planning* [7], *universal planning* [9], *conformant planning* [8], and *directed planning* [14] indicate. Since planning is in fact model checking [18], protocol verification share similar properties. In fact communication protocols specify non-deterministic planning problems with resources - a current hype research

topic in AI. 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 [44]. The approach to parse LTL specification in the problem description language seems unnatural, since the opposite is by far more apparent.

Therefore, our approach provides another bridge for the gap between AI-planning and verification. Since Promela specifications have the expressiveness to deal with variables and suited to concurrent processes, Promela can serve as a input language for non-deterministic planning problem with resources. The goal can be encoded to violate a certain assertion such that directed search method can be applied.

Note that heuristic search is currently the most effective approach in planning. Four of five honored planning systems in the general planning track of the AIPS-2000 competition at least partially incorporate heuristic search. In traversing huge state spaces of all combinations of grounded predicates they all rely on inadmissible estimates. Fortunately, the competition problem instances yield good solutions even when invoking non-optimal planning algorithms.

7 Conclusion

Protocol verification and directed search can cooperate. We have presented a vivid research branch to effectively search for errors in protocol specifications with current heuristic search techniques that incorporate information to reduce the search space by magnitudes, although the information encoded in the heuristics is relativly weak. For example the values in the state distance matrices do not adequately respect the number of receive and send operations according to the number of elements in the queue. Establishing good estimates for the actual number of transpositions without necessarily encounter a combinatorial explosion is the most challenging research topic in the near future.

The problem of determining whether an arbitrary message passing system contains deadlocks is a PSPACE complete problem [24]. However, focusing a specific protocol with heuristic estimates at hand directed search detects errors even in large depths. Our implementation provides an efficient interface for any single state space exploration algorithm. Validating liveness properties corresponds to detect cycles in protocol execution and to apply nested search procedures [23]. The directed approach can contribute to this approach, searching for the state we started from.

Acknowledgment S. Edelkamp and A. Lluch Lafuente are working in a DFG project entitled Heuristic Search and its Application in Protocol Validation.

References

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

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, 1999.

[3] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, et al. VIS: A system for verification and synthesis. In *CAV-93*, pages 428–432, 1993.

[4] C. A. Bröcker. Vega—a user-centered approach to the distributed visualization of geometric algorithms. Technical Report 117, Institut für Informatik, 1998.

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

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

[7] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for AR. In *ECP*, 1997.

[8] A. Cimatti and M. Roveri. Conformant planning via model checking. In *ECP*, pages 21–33, 1999.

[9] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *AAAI*, pages 875–881, 1998.

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

[11] R. Dial. Shortest path forest with topological ordering. *Communications of the ACM*, pages 632–633, 1969.

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

[13] S. Edelkamp. *Data Structures and Learning Algorithms in State Space Search*. PhD thesis, University of Freiburg, 1999. DISKI, Infix, Band 201.

[14] S. Edelkamp. Heuristic search planning with bdds. In *PUK*, pages 16–25, 2000.

[15] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state en coding length. In *ECP-99*, pages 135–147, 1999.

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

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

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

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

[20] 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.

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

[22] G. Holzmann. An improved protocol reachability analysis technique. *Software, Practice & Experience*, 18(2):137–161, 1988.

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

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

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

[26] A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search*. PhD thesis, University of Alberta, 1999.

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

[28] R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI-97*, pages 700–705, 1997.

[29] R. E. Korf. Divide-and-conquer bidirectional search: First results. In *IJCAI*, pages 1184–1189, 1999.

[30] R. E. Korf and L. A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *AAAI*, pages 1202–1207, 1996.

[31] R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. In *AAAI*, pages 910–916, 2000.

[32] T. Kropf. *Introduction to Formal Hardware Verification*. Springer, 1999.

[33] S. Leue and G. Holzmann. v-promela: A visual, object-oriented language for spin. In *ISORC*, 1999.

[34] 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.

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

[36] K. L. McMillan. Overview of verification. In M. K. Inan and R. P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, pages 3–13. Springer, 1998.

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

[38] K. L. McMillan. Temporal logic and model checking. In M. K. Inan and R. P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, pages 36–54. Springer, 1998.

[39] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking and model checking. In *LNCS 1102*, pages 411–414, 1996.

[40] D. Peled. Model checking using automata theory. In M. K. Inan and R. P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, pages 36–54. Springer, 1998.

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

[42] F. Reffel and S. Edelkamp. Error detection with directed symbolic model checking. In *FM-99*, pages 195–211. Springer, 1999.

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

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

[45] A. S. Tanenbaum. *Computer Networks (sec. ed.)*. Prentice Hall, 1989.

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

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

[48] J. Zucker. *The Propositional μ-Calculus and Its Use in Model Checking*, pages 117–128. Springer, 1993.