First publ. in: Electronic Notes in Theoretical Computer Science 55 (2001), 3, pp. 343-356

Trail-Directed Model Checking

Stefan Edelkamp, Alberto Lluch-Lafuente and Stefan Leue

Institut für Informatik
Albert-Ludwigs-Universität
Georges-Köhler-Allee
D-79110 Freiburg, Germany
eMail: {edelkamp,lafuente,leue}@informatik.uni-freiburg.de
URL: www.informatik.uni-freiburg.de/~{edelkamp,lafuente,leue}

Abstract

HSF-SPIN is a Promela model checker based on heuristic search strategies. It utilizes heuristic estimates in order to direct the search for finding software bugs in concurrent systems. As a consequence, HSF-SPIN is able to find shorter trails than blind depth-first search.

This paper contributes an extension to the paradigm of *directed model checking* to shorten already established unacceptable long error trails. This approach has been implemented in HSF-SPIN. For selected benchmark and industrial communication protocols experimental evidence is given that *trail-directed model-checking* effectively shortcuts existing witness paths.

1 Introduction

The formal methods of model checking [4] have various applications in software verification[2]. Through the exploration of large state-spaces model checking produces either a formal proof for the desired property or a detailed description of an error trail. We concentrate on explicit state model checking and its application to the validation of communication protocols.

In the broad spectrum of techniques for tackling the huge state space that are generated in concurrent systems, heuristic search is one of the new promising approaches for failure detection. Early precursors execute explicit best-first exploration in protocol validation [18] and symbolic best-first search in the model checker Murϕ [22]. Symbolic guided search in CTL model checking is pursued in [3] and bypasses intense symbolic computations by so-called hints. Last but not least, the successful commercial UPPAAL verifier for real-time systems represented as timed automata has also been effectively enriched by directed search techniques [1].

Our own contributions to *directed model checking* integrate heuristic estimates and search algorithms to the μcke model checker [21], to a domain independent AI-planner [8], and to a Promela model checker [9,10]. The global state space is interpreted as an implicitly given graph spanned by a successor generator function, in which paths corresponding to error behaviors are searched. The length of the witness path is crucial to the designer/programmer to debug the erronous piece of software; shorter trails are easier to interpret in general.

In the model checker SPIN [13] safety properties are checked through a simple depth-first search of the system's state space, while liveness properties require a two-fold nested depth-first search. The error trail in the first case is a simple path from the start state to an error state, while in the second case we have a seeded cycle, that is a path composed by a prefix that leads to a seed state, followed by a cycle that is closed at this state.

Our experimental tool HSF-SPIN [1] provides AI heuristic search strategies like A*, IDA* and best-first for finding safety errors [10], and an improved version of nested depth-first search [9], based on exploting the never-claim representation of the required temporal property to simplify the checking process.

In this paper we concentrate on error trail improvement, an apparent need in practical software development. We expect that a possibly long witness for an error is already given. This trail might be found by simulation, test, random walk, or depth-first model checking. The witness is read as an additional input, reproduced in the model and then significantly improved by directed search.

HSF-SPIN tries to find errors faster than traditional tools by employing heuristic search strategies for non-exhaustive, guided state space exploration. While HSF-SPIN can be used for full verification through exhaustive state space search, this is not its primary objective and we note that other model checkers, like Spin or SMV, are likely to be more time and space efficient for this purpose.

The paper is structured as follows. First we give some background on the AI technique we use. In a next section we introduce the HSF-SPIN model checker and its usage in terms of its command line options. In the following sections we address the facets of trail-directed search, based on the hamming distance and the FSM distance. We distinguish between single-state trail directed search for safety errors and cycle-detection trail-directed search for liveness errors. Both approaches have been implemented in HSF-SPIN and in the experimental section we present first results. We close with some concluding remarks.

[1] Available from `http://www.informatik.uni-freiburg.de/~lafuente/hsf-spin`

2 Heuristic Search

Depth-first search and breadth-first search are call *blind* search strategies, since they use no information of the concrete state space they explore. On the other hand, heuristic search algorithms take additional search information in form of an evaluation function into account. This function is used to rank the desirability of expanding a node u.

A* [11] uses an evaluation function $f(u)$ that is the sum of the generating path length $g(u)$ and the estimated cost of the cheapest path $h(u)$ to the goal. Hence $f(u)$ denotes the estimated cost of the cheapest solution through u. If $h(u)$ is a lower bound then A* is optimal, i.e. it finds solution paths of optimal length.

Table 1 depicts the implementation of A*, where $g(u)$ is the length of the traversed path to u and $h(u)$ is the estimate distance from u to a failure state.

A* (s)
  $Open \leftarrow \{(s, h(s))\}$; $Closed \leftarrow \{\}$
  while $(Open \neq \emptyset)$
    $u \leftarrow Deletemin(Open)$; $Insert(Closed,u)$
    if $(failure(u))$ **exit** *Goal Found*
    for all v **in** $\Gamma(u)$
      $f'(v) \leftarrow f(u) + 1 + h(v) - h(u)$
      if $(Search(Open, v))$
        if $(f'(v) < f(v))$
          $DecreaseKey(Open, (v, f'(v))$
      else if $(Search(Closed, v))$
        if $(f'(v) < f(v))$
          $Delete(Closed, v)$; $Insert(Open, (v, f'(v))$
      else $Insert(Open, (v, f'(v))$

Table 1
The A* Algorithm.

The algorithm divides the state space in three sets: the set *Open* of visited but not expanded states, the set *Closed* of visited and expanded states, and the set of not already visited states. Similar to Dijkstra's single source shortest path exploration [7], starting with the initial state, A* extracts states from the *Open* set, move them to the *Closed* set and insert their successors in the *Open* set until a goal state is found. In Table 1 the differences between Dijkstra's algorithm and A* are underlined. In each expansion step the state with best f value is selected to be expanded next. Nodes that have already been expanded might be encountered on a shorter path. Contrary to Dijkstra's algorithm, A* deals with the problem by re-inserting the corresponding nodes from the set of

already expanded nodes into the *Open* set. This scheme is called *re-opening*.

3 HSF-SPIN

HSF-SPIN merges the model checker Spin [2] and the heuristic search framework HSF [3]. It is basically an extension of HSF for searching state spaces generated by Promela models.

Like in Spin, two steps must be performed prior to the verification process. The first step generates the source code of the verifier for a given Promela specification. In the second step, the source code is compiled and linked for constructing the verifier. The verifier then checks the model. Among other parameters the user can specify the error type, the search algorithm, and the heuristic estimate as command line options. It is also possible to perform interactive simulations similar to Spin. When verification is done, statistic results are displayed and a solution trail in Spin's format is generated.

HSF-SPIN is based on Spin and its specification language Promela. However, HSF-SPIN is not 100% Promela compatible. Promela specifications with dynamic or non-deterministic process creation are not yet accepted in HSF-SPIN. HSF-SPIN can check all the properties that Spin can validate with the exception of non-progress cycles. HSF-SPIN supports sequential bit-state hashing, but not partial order reduction.

3.1 A First Example

The HSF-SPIN distribution includes a set of test models. For example, the file `deadlock.philosophers.prm` implements a Promela model of a deadlock solution to Dijkstra's dining philosophers problem. The executable `check` is a verifier of the model, similar to Spin's executable file `pan`. Deadlocks are checked by running the verifier with argument `-Ed` resulting in the following output.

```
HSF-SPIN 1.0
A Simple Promela Verifier based on Heuristic Search Strategies.
This tool is based on Spin 3.4.5  (by G.J. Holzmann) and
                    on HsfLight 2.0 (by S. Edelkamp)
Verifying models/deadlock.philosophers.prm...
Checking for deadlocks with Depth-First Search...
        invalid endstate (at depth 1362)
Printing Statistics...
        State-vector 120 bytes, depth reached 1362, errors: 1
             1341 states, stored
              431 states, matched
             1772 transitions (transitions performed)
```

[2] http://netlib.bell-labs.com/netlib/spin/whatispin.html
[3] http://www.informatik.uni-freiburg.de/~edelkamp/Hsf

```
              25 atomic steps
            1341 states, expanded
        Range of heuristic was: [0..0]
Writing Trail
Wrote models/deadlock.philosophers.prm.trail
        Length of trail is 1362
```

The verifier runs depth-first search, since it is the default search algorithm. It finds a deadlock at depth 1,362. Following such a long trail is tedious. The A* algorithm (option `-AA`) and a simple heuristic estimate for deadlock detection (option `-Ha`) finds a deadlock at optimal depth 34, expanding and storing less states (17 and 67, respectively), and performing less transitions (73).

3.2 Compile and Run-Time Options

The HSF-SPIN verifier accepts only a reduced subset of Spin's compile-time options, for example `-DVECTORSZ` and `-DGCC`. The only specific compile-time option is `-DDEBUG`, to report debug information when running. Each command line argument of HSF-SPIN has the form $-Xx$, where X is the option to be set and x is the value for the option. For example, argument `-Ad` sets the option *search algorithm* to the value *depth-first search*. By giving an option no value, the list of available values for that option is printed. For example, executing `check -A` prints all available search algorithms.

Executing the HSF-SPIN verifier without arguments outputs all available run-time options, e.g. `-Ax`, where x is the search algorithm (A*, IDA*, DFS, NDFS, etc.); `-Ex`, where x is the error to be checked (Deadlock, Assertion, LTL, etc.); and `-Hx`, where x is the heuristic function (Formula-based, Hamming distance, FSM distance, etc.).

4 Improvement of Trails

Since various explicit on-the-fly model checkers like Spin search the superimposed global state space in depth-first manner, they report the first error that has been encountered even if it appears at a high search depth. One natural option to improve the trail is to impose a shallower depth on the depth-first search engine. However, there are two severe drawbacks to this approach.

The first one is that bounds might increase the search efforts by magnitudes, since a fixed traversal ordering in bounded depth-first exploration in large search depths might miss the lasting error states for a fairly long time. Therefore, even if the first error is found fast, improvements are possibly difficult to obtain. Moreover, to find shorter trails by manual adjusting bounds is time consuming, e.g., trying to improve an optimal witness will fail and result in a full state exploration.

The second drawback, which we call *Anomaly in Depth-Bounded Search* (cf. Figure 1) is even more crucial to this approach. It can be observed when

experimenting with explicit state model checkers that allow the search depth to be limited to a maximum, such as it can be done in SPIN, and in which visited states are kept in a hash-table to avoid an exponential increase in the number of expanded nodes due to the tree expansion of the underlying graph. This implicit pruning result in the fact, that duplicate errors in smaller depths will not necessarily be detected anymore, since they might be blocked by nodes that are already stored. This anomaly emerges frequently in practice when atomic transitions are used, which correspond to potentially long non-branching paths in the search tree. In other words, depth-bounded search with node caching is not complete for error detection in shallower depths than the given bound [4] .

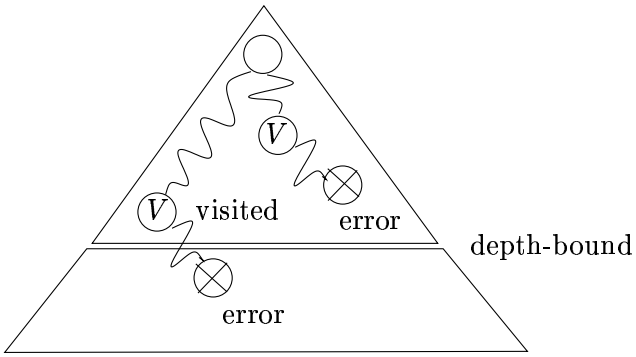

Fig. 1. Anomaly in Depth-Bounded Search.

We have observed this behavior in some of our models. For example, in a model of a telephony system after establishing a witness of length 756, the search with a new bound 755 fails to find one of the remaining error states. For the same Promela model, error detection alternates with different search depths bound: up to bound 67 no error is found, from bound 68 to 139 an error is found, from bound 140 to 154 no error is found, from 155 onwards an error is again found, and so on.

A simple method to correct this anomaly is to enforce *revisiting* of some states. More precisely, a state is revisited (reexplored) when it is reached on a shorter path. Therefore, each state is stored in the hash-table together with its smallest depth value. In fact, this observation was already made for the Spin model checker, in which the anomaly is fixed with the `-DREACH` directive. However, since entire subtree structures for revisited states are re-explored, this method causes a possibly exponential increase in time complexity.

Therefore, we aim at a different aspect of trail improvement; namely heuristic search. The idea is to take the failure state or some of its defining features to set up a heuristic estimate that guides the search process into the direction of that particular state. In contrast to heuristic search strategies described in

[4] Note that to the contrary, the iterative-deepening variant of A* (IDA*) is complete, since it invokes the depth-first search process starting with the smallest available bound and increasing this bound the smallest possible amount.

previous work [9,10], we exhibit refined information. The main argument is that it is easier to find a specific error situation instead of finding any member according to a general error description. We distinguish two heuristics and two search algorithms. The first heuristic is designed to focus exactly the state that was found in the guidance trail, while the second heuristic relaxes this requirement to important aspects for the given failure type. The two algorithms divide in trail-directed search for safety property violation and trail-directed search for liveness property violation.

4.1 Hamming Distance Heuristic

Let S be a state of the search space given in a suitable binary encoding, i.e. as a bit vector $S = (s_1, \ldots, s_k)$. Further on, let S' be the error state we are searching for. One coarse estimate for the number of transitions necessary to get from S to S' is the number of bit-flips necessary to transform S into S'. The estimate is called the *Hamming distance* $H_{HD}(S, S')$, determined by

$$H_{HD}(S, S') = \sum_{i=1}^{k} |s_i - s'_i|$$

Obviously, $|s_i - s'_i| \in \{0, 1\}$ for all $i \in \{1, \ldots, k\}$. Note that the estimate $H_{HD}(S, S')$ is not a lower bound, since one transition might change more than one bit in the state description at a time. Moreover, the Hamming distance can be refined by taking the binary encoded values of the state variables and their modifiers into account. Nevertheless, the Hamming distance reveals a valuable ordering of the states according to their goal distances.

4.2 FSM Distance Heuristic

Another distance metric centers around the local states of the finite state machines, which together with the communication queues and variables generate the system's global state space.

Let $(pc_1, \ldots, pc_l)$ be the vector of all FSM locations in a state S, i.e. pc_i, $i \in \{1, \ldots, l\}$, denotes the corresponding program counter. The FSM distance metric $H_{FSM}(S, S')$ according to the goal state S' with FSM state vector $(pc'_1, \ldots, pc'_l)$ is calculated in each FSM separately. When assuming independence of the execution in each finite state machine we can approximate

$$H_{FSM}(S, S') = \sum_{i=1}^{k} D_i(pc_i, pc'_i)$$

The distances $D_i(pc_i, pc'_i)$ are calculated as the minimal graph theoretical distance from pc_i to pc'_i, $i \in \{1, \ldots, l\}$. These values are computed beforehand for each pair of local states with the all-pairs shortest-path algorithm of Floyd-Warshall, so that the retrieval of each value $D_i(pc_i, pc'_i)$ is a constant time

operation. In contrast to the Hamming distance, the FSM distance abstracts from the current queue load and from values of the local and global variables. We expect that the search will be then directed into equivalent error states that could potentially be located at smaller search tree depths (see Figure 3).

4.3 Safety Errors

Trail-directed search for safety errors, as visualized in Figure 2, takes a trail as an additional input for the model-checker and searches for improvements of its length, especially for a concise and transparent bug-finding process.

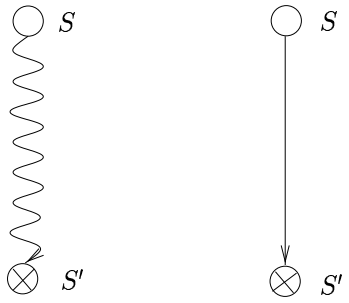

Fig. 2. Safety Error Trail is Shortened by Trail-Directed Search.

In our case we extract the error state S' to focus the search by the above heuristics $H_{HD}(S, S')$ and $H_{FSM}(S, S')$. These estimates are integrated in the heuristic search algorithm A*. Recall that is complete, and that, if the estimate is a lower bound, the path is optimal.

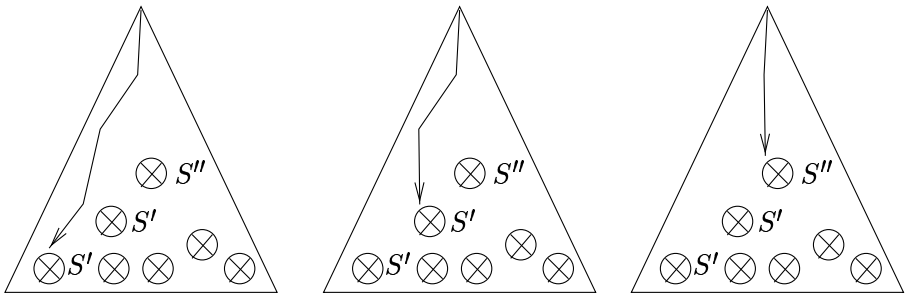

Fig. 3. Search Trees of Ordinary Search, Full-State Trail-Directed Search, and Partial-State Trail-Directed Search.

Figure 3 depicts the search tree inclusive the established trail according to ordinary search, A* with the Hamming distance heuristic H_{HD}, and A* with the FSM distance heuristic H_{FSM}. Since H_{HD} uses the entire error state description, we call this search *full-state trail-directed search*, while in case of H_{FSM} only a part of the error state description is used, such that this approach is referred to as *partial-state trail-directed search*.

Remember that a trail to a violated liveness property consists of a path with an initial prefix to a seed state and a cycle starting from that state. Therefore, we can improve the witness trail by trail-directed A*-like search in both parts (cf. Figure 4).

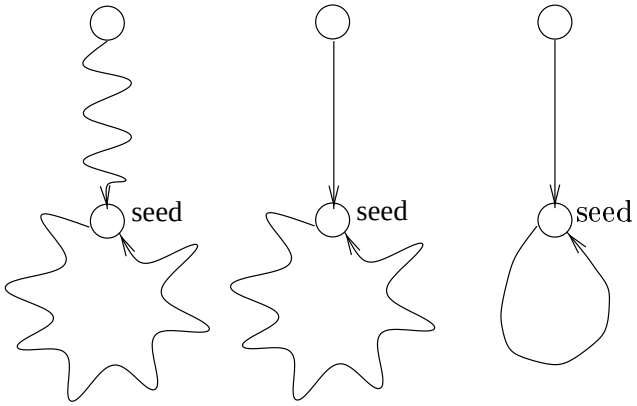

Fig. 4. Liveness Error Trail Shortened in Two Phases.

In a first improvement phase we search for shortcuts of the path to the seed state. In an independent second phase we perform a cycle-detection search, i.e. a search guided by the seed state from which it has started. In both cases the proposed estimate that we propose is the Hamming distance heuristic H_{HD}, since we are searching for the exact seed state, and not for an equivalent one.

5 Experiments

In our experimental study selected examples for trail improvements are used. We apply the above algorithms to trails obtained by our depth-first search algorithm, producing the same or very similar results to SPIN's depth-first search traversal.

First we consider deadlock detection. As an example we choose the industrial GIOP protocol [15] with a seeded bug and a model of a concurrent program that solves the stable marriage problem [19] [5]. Table 2 shows that the witness trail is improved to about a half of its original length. The values 67 and 65 in the GIOP model are close to the optimal trail length of 58. In the second case the solution path obtained when using the FSM-based heuristic is near to the optimum of 62 and notably better of the length provided by the algorithm with the Hamming distance heuristic. However, in both examples the search efforts are significantly higher in the case of the FSM-based heuristic than in the case of the Hamming distance heuristic.

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

		DFS	TDA*,H_{HD}	TDA*,H_{FSM}
GIOP	Stored States	326	988	30,629
(N=2,M=1)	Transitions	364	1,535	98,884
	Expanded States	326	432	24,485
	Witness Trail	134	67	65
Marriers	Stored States	407,009	26,545	225,404
(N=4)	Transitions	1,513,651	56,977	467,704
	Expanded States	407,009	16,639	192,902
	Witness Trail	121	99	66

Table 2
Improving Trails of Deadlocks with Trail-Directed Search in the GIOP and Marriers models.

In the second set of examples we examine another safety property class, namely state invariants. The two protocols we consider are a Promela model of an Elevator system [6] and the POTS telephony protocol model [16]. Table 3 shows that the witness trail is shortened by trail-directed search from 510 to 203 and from 756 to 67, respectively. In this case there is no significant difference between the two heuristic estimates.

		DFS	TDA*,H_{HD}	TDA*,H_{FSM}
Elevator	Stored States	292	38,363	38,538
($N = 3$)	Transitions	348	146,827	147,277
	Expanded States	292	38,423	38,259
	Witness Trail	510	203	203
POTS	Stored States	506,751	2,668	2,019
	Transitions	1,468 10^6	6,519	4,889
	Expanded States	506,751	2,326	997
	Witness Trail	756	67	67

Table 3
Improving Trails of Invariants Violation with Trail-Directed Search in the Elevator and POTS models.

A *bad sequence* corresponds to a violation of a liveness property. How-

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

ever, it does not reflect a cyclic witness but a simple path. The results in Table 4 shows the impact of trail improvement in this scenario for a model of a Fundamental-Mode Circuit (FMC [20]).

		DFS	TDA*,H_{HD}	TDA*,H_{FSM}
FMC	Stored States	270	438	419
($N = 3$)	Transitions	364	664	624
	Expanded States	279	437	412
	Witness Trail	259	73	73

Table 4
Improving the Trail of a Bad Sequence in the FMC model.

The last example is trail improvement for liveness properties that include cycles at seed states in their witness paths. Once more we use the Elevator protocol as a representative example.

		NDFS	TDA*,H_{HD}	INDFS	TDA*,H_{HD}
Elevator	Stored States	171	11,205	166	10,930
($N = 2$)	Transitions	259	38,307	194	37,656
	Expanded States	208	10,901	166	10,764
	Seed at Depth	187	173	177	163
	Cycle Length	90	90	90	90
	Total Length	277	263	267	253

Table 5
Improving the Trail of Liveness Property Violation in the Elevator Protocol.

Table 5 depicts the results of trail-directed search applied to trails obtained by nested depth-first search (NDFS) and the improved version of this algorithm (INDFS). It is shown that cycle seeds are found at smaller depths for the error trails of both algorithms, while the cycle length has not been improved. On the other hand considerable work is necessary to improve the length of the trail. Since this is only a single data point more protocols with liveness properties are required for a better judgment.

6 Conclusions

While previous work on *directed model checking* concentrates on detecting unknown error states, the paradigm of *trail-directed model checking* contemplates the improvement of trails result from error detections, simulations, etc.

On the other hand, although paths to errors could be improved with *directed model checking*, the new paradigm proposes richer heuristics based on the information of a singleton given error states. Moreover *directed model checking* is restricted to safety properties, while *trail-directed model checking* is able to improve error trails corresponding to such type of properties.

Trail improvement in our directed model checking tool HSF-SPIN turns out to be an effective aid in software design of concurrent systems. With an acceptable overhead already existing paths are reduced by heuristic search for the established error. The first results are promising and put forth the idea of *trail-directed model checking*, that might include more information than the mere description of the error state.

One early approach for focusing trail information is *diagnostic model checking for real-time systems* [17]. It also shifts attention to highlight failure detection, but does not clarify why the established traces are improved compared to ordinary failure trails. Another line of research aims not only to report what went wrong, but explain why it went wrong. However, most approaches in this class such as assumption truth-maintenance systems implemented in the General Diagnostic Engine (GDE [6]) turn out to scale badly.

At the moment we concentrate on SPIN's Promela specification language, but in future we are interested in verifying real software in Java and C. The Bandera tool [5] developed at Kansas University allows slicing of distributed Java-Programs with an export to either SPIN or SMV. The same research line is pursued by the Automated Software Engineering group at NASA Ames Research Center that apply a Java byte code verifier, called Java Path Finder [12]. On the other side, Holzmann [14] has pushed the envelope for actual C-Code verification with the SPIN validator.

References

[1] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Petterson, and J. Romijn. Guiding and cost-optimality in UPPAAL. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 66–74, 2001.

[2] B. Bérard, A. F. M. Bidoit, F. Laroussine, A. Petit, L. Petrucci, P. Schoenebelen, and P. McKenzie. *Systems and Software Verification.* Springer, 2001.

[3] R. Bloem, K.Ravi, and F.Somenzi. Symbolic guided search for ctl model checking. In *Conference on Design Automation (DAC)*, pages 29–34, 2000.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *International Converence on Software Engineering (ICSE)*, 2000.

[6] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, pages 1340–1330, 1987.

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

[8] S. Edelkamp. Directed symbolic exploration and its application to AI-planning. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 84–92, 2001. Precursor *S. Edelkamp, Directed Symbolic Exploration in Planning* published in European Conference on Artificial Intelligence (ECAI), Workshop on New Results in Planning, Scheduling and Design (PUK-2000).

[9] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *8th International SPIN Workshop on Model Checking Software*, Lecture Notes in Computer Science 2057, pages 57–79. Springer, 2001.

[10] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.

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

[12] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000.

[13] G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[14] G. J. Holzmann and M. H. Smith. Software model checking: Extracting verification models from source code. In *Formal Description Techniques for Distributed Systems and Communication Protocols, Protocol Specification, Testing and Verification (FORTE/PSTV)*, pages 481–497. Kluwer, 1999.

[15] M. Kamel and S. Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *International Journal on Software Tools for Technology Transfer*, 2(4):394–409, 2000.

[16] M. Kamel and S. Leue. Vip: A visual editor and compiler for v-promela. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, pages 471–486. Springer, 2000.

[17] K. G. Larsen, P. Pettersson, and W. Yi. Diagnostic model-checking for real-time systems. In *Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 575–586. Springer, 1995.

[18] 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.

[19] D. McVitie and L. Wilson. The stable marriage problem. *Communications of the ACM*, 14(7):486–492, 1971.

[20] B. Rahardjo. Spin as a hardware design tool. In *First SPIN Workshop*, 1995.

[21] F. Reffel and S. Edelkamp. Error detection with directed symbolic model checking. In *World Congress on Formal Methods (FM)*, Lecture Notes in Computer Science, pages 195–211. Springer, 1999.

[22] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Conference on Design Automation (DAC)*, pages 599–604, 1998.