Error Detection with Directed Symbolic Model Checking

Frank Reffel[1] and Stefan Edelkamp[2]

[1] Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe,
reffel@ira.uka.de
[2] Institut für Informatik, Albert-Ludwigs-Universität Freiburg,
edelkamp@informatik.uni-freiburg.de

Abstract. In practice due to entailed memory limitations the most important problem in model checking is state space explosion. Therefore, to prove the correctness of a given design binary decision diagrams (*BDDs*) are widely used as a concise and symbolic state space representation. Nevertheless, *BDDs* are not able to avoid an exponential blow-up in general. If we restrict ourselves to find an error of a design which violates a safety property, in many cases a complete state space exploration is not necessary and the introduction of a heuristic to guide the search can help to keep both the explored part and the associated *BDD* representation smaller than with the classical approach.
In this paper we will show that this idea can be extended with an automatically generated heuristic and that it is applicable to a large class of designs. Since the proposed algorithm can be expressed in terms of *BDDs* it is even possible to use an existent model checker without any internal changes.

1 Introduction

To formulate the specification properties of a given design many different temporal logics are available, each of them with a different expressive power: (Fair-) CTL [6] is a branching time logic, LTL [18] is a linear time logic and CTL* [13] is a superset containing both of them. CTL* itself is a subset of the μ-calculus [17] which in addition allows to verify bisimulation and other more complex properties.

In practice, however, the characteristics people mainly try to verify are simple safety properties that are expressible in all of the logics mentioned above. They can be checked through a simple calculation of all reachable states. Unfortunately, this computation can become intractable for systems consisting of several asynchronously interacting modules.

Although *BDDs* [5] allow a succinct representation of a system they cannot always avoid an increase in *BDD*-sizes caused by the typical exponential blow-up of states. However, model checking is not only used to show the correctness of a complete system, but also as a very efficient method to find errors during the construction phase in order to avoid cost intensive correction phases later on.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 195–211, 1999.

In early design phases a system typically contains many errors such that nobody would expect a successful verification. We should try to detect these errors as soon as possible to avoid the calculation of the entire state space. Local model checking methods [14] attempt to exploit only a small part of the state space while global model checking techniques usually calculate all reachable states. Moreover, their fix-point calculation requires a backward traversal and a lot of work is spent in treating unreachable states. Hence, in order just to detect an error local model checking methods [24] can be more efficient. So a suitable application of model checking can replace parts of the classical debugging and testing work because it allows the detection of more errors in less time.

The method proposed in this paper focuses on safety properties. Starting with the set of initial states it performs a forward traversal of the system and exploits only that part of the set of reachable states that is most likely to lead to an error state. This is sufficient to construct a counter example of the violated property helping the designer to understand and fix the failure of the system. To guide the search a heuristic estimates the number of transition steps necessary to reach the error state. If the heuristic fulfills a certain property it guarantees the detection of a minimal counter example.

Our algorithm detects errors in systems unable to be verified by traditional symbolic model checking since the *BDDs* exhaust the available memory resources. Even if we assume pure forward traversal, after several iterations not containing an error state the large amount of states that has to be stored by an unguided search becomes too big; while heuristic search finds the error within an acceptable amount of time without suffering from memory problems.

Since all states have to be visited, our method fails to entirely validate a correct system, but this should be postponed until the end of the construction phase when most of the errors have been removed and the correctness of the system is more probable. The successful verification of large systems can be a very time consuming work which requires elaborated methods and a lot of experience. This results in a manually driven process with a lot of expertise demanding a specialist. We recommend a distinction of a verification to prove the correctness of a system and the use of a model checker as debugging tool, since the ultimate goal is to tediously prove the system only once and not after every detection and correction of an error.

The paper is structured as follows. In Section 2, we introduce some basics about *BDDs*. Section 3 addresses traditional symbolic model checking and Section 4 its proposed enhancement with a heuristic. The automatic inference of the heuristic is the topic of Section 5. Finally, Section 6 presents our results in verifying a buggy design of the tree-arbiter and the DME.

2 *BDD* Basics

Ordered binary decision diagrams (*OBDDs*) introduced by Bryant [5] are a graphical representation for boolean functions. A *BDD* $G(f, \pi)$ with respect to the function f and the variable ordering π is an acyclic graph with one source

and two sinks labelled with *true* and *false*. All other (internal) nodes are labelled with a boolean variable x_i of f and have two outgoing edges *left* and *right*. For all edges from an x_i labelled node to an x_j labelled node we have $\pi(i) < \pi(j)$, such that on every path in G the variables are tested in the same order and at most once. Reduced *BDDs* with respect to a fixed variable ordering are a canonical representation for boolean functions. A *BDD* is reduced if isomorphic sub-*BDDs* are merged and nodes whose outgoing edges lead to the same successor are omitted. Reduced *BDDs* are build directly, integrating the reduction rules into the construction algorithm. The variable ordering π can be choosen freely, but it has a great influence on the size of the *BDDs*, e.g. there are functions which have *BDDs* of linear size for a "good" and of exponential size for a "bad" ordering. The determination of an optimal ordering is an NP-hard problem but, for most applications, there exist several heuristics for non-optimal but "good" orderings [2]. Another method to improve the ordering is dynamic variable reordering [23] which is applied during the verification in case the *BDDs* become too large. In the following we will only speak of *BDDs*, however, we always mean reduced ordered *BDDs*.

In model checking *BDDs* help to overcome the memory limitations of explicit state representation methods [19]. They represent both the sets of states and the transition relation. Model checking temporal logic properties can be reduced to the calculation of fix-points. This calculation can be performed efficiently treating lots of states in each iteration step.

An important task is to determine the set of reachable states. Starting with the set of initial states the fix-point iteration corresponds to a breadth-first-search until no more new states are found. This is sufficient to check safety and simple reachability properties. To verify more complicated properties typically a backward state traversal is applied to calculate the necessary fix-points. As a drawback many unreachable states have to be represented because the reachability status of a given state is not known at the beginning of the verification.

3 Model Checking

First we expose the structure of the transition relation and examine the calculations that have to be performed to check safety properties with a classical symbolic model checker. Thereafter, we discuss alternative methods that try to overcome the weaknesses of the breadth-first-search approach.

3.1 Traditional Symbolic Model Checking

In order to apply a model checker we need a description of the system and the safety property to be verified. The μ-calculus is an example of a logic in which both descriptions can be expressed. The two predicates *Start* and *Goal* describe the set of initial states and the set of error states, respectively. In addition a predicate *Trans* is required that evaluates to *true* if and only if there is a

transistion between two successive states. For an interleaving model the predicate *Trans* is defined by the following equation:

$$Trans(State\ s, State\ t) = \bigvee_{i=1}^{n} Trans_i(s, t_i) \wedge CoStab_i(s, t).$$

Systems typically consist of several interacting modules. The interaction can be synchronous, asynchronous or interleaving. Here the verification of an interleaving model is described while for the verification of the other two models only small changes in the transition relation have to be made.

The predicate $CoStab_i$ describes that all modules except module i preserve their state and the predicate $Trans_i$ describes the transition relation of the single module i that might depend on the states of up to all other (n) modules but that only changes its own state s_i into t_i. The state of a single module consists of several (m) variables of type: bool, enumerated or integer (with limited values) or a combination of them. They are all translated into boolean variables such that for all modules we end up with an expression of the form:

$$Trans_i(State\ s, ModuleState\ t_i) = \bigwedge_{j=1}^{m} T_{i,j}(s, t_{i,j}),$$

where $t_{i,j}$ describes the state of variable j in module i. The transition $T_{i,j}$ of a single variable $s_{i,j}$ describes the possibility to change its value according to its input variables or to persist in its state. A backward traversal of the system then calculates the following fix-point:

$$\mu\ F_{Goal}(State\ s).Goal(s) \vee (\exists State\ succ.Trans(s, succ) \wedge F_{Goal}(succ)).$$

After determining the set of states satisfying this fix-point we check if it contains the initial state. As said above, the disadvantage of this approach is that many unreachable states have to be stored. The alternative is to start with the set of initial states and to make a forward traversal calculating the transitive closure *Reach* of the transition relation:

$$\mu\ Reach(State\ s).Start(s) \vee (\exists State\ prev.Trans(prev, s) \wedge Reach(prev)).$$

The efficiency can be improved: After each fix-point iteration we check if the set contains an error state in which case the verification can be aborted. Based on an interleaving (asynchronous) combination of modules, however, each order of transitions of the single modules has to be taken into account leading to state space explosion. Therefore, the sole calculation of reachable states might be impossible.

3.2 Other Approaches

An attempt to overcome the disadvantage of the unreachable states to be stored in a backward traversal is *local model checking* [7, 24]. It applies a depth-first

search allowing an intelligent choice of the next state to be expanded. It significantly reduces the verification time in case properties are checked that do not necessarily require the traversal of the whole state space. In general, local model checking uses an explicit state representation such that it cannot take profit from the elegant and space efficient representation based on *BDDs*.

An attempt to combine *Partial order reduction* with *BDDs* was made in [1]. Nevertheless, it was not yet as successful as global *BDD*-model checking.

Bounded model checking performs symbolic model checking without *BDDs* using *SAT* decision procedures [4]. The transition relation is unrolled k steps for a bounded k. The bound is increased until an error is found or the bound is large enough to guarantee the correctness of a successful verification. The major disadvantage of bounded model checking is the fact that it is difficult to guarantee a successful verification, since the necessary bound will be rather large and difficult to determine. Therefore, similar to our approach the most important profit of this method is a fast detection of errors.

Validation with guided search [25] is the only other approach known to the authors which tries to profit from a heuristic to improve model checking. The measure is the Hamming distance, i.e. the minimum number of necessary bit-flips to transfer a given bit-vector of a state to an erroneous one. For our purposes this heuristic is too weak. The lower bound presented in Section 5 has a larger range of values leading to a better selection of states to be expanded. Furthermore, the pure effect of the heuristic is not clearly evaluated. The authors compare the number of visited and explored states with a breadth-first-search, but unfortunately the *BDD*-sizes for the state representation, the key performance measure, are not mentioned. It does not become clear which parts of the verification are performed with *BDDs* and which parts are dealt otherwise. The proposed approach is combined with two other methods: *target enlargements* and *tracks*. The former corresponds to a certain kind of bidirectional search, which is not a heuristic but a search strategy close to perimeter search [10] and the latter seems to be highly manually driven and not suitable for automatisation, one of our principal aims. The combination of their methods to find an error leads to good results, but in our opinion, it seems that the heuristic only contributes a small part to this advancement.

In contrast our algorithm entirely utilizes the *BDD* data structure such that the only interesting point are the sizes of the *BDDs* and not the number of states represented by it. In the examples of Section 6 the overall time and memory efficiency of our approach is shown to outperform traditional *BDD* breadth-first-search.

4 Directed Model Checking

In *BDD* based breadth-first-search all states on the search horizon are expanded in one iteration step. In contrast our approach is directed by a heuristic that determines a subset of the states on the horizon to be expanded which most promisingly leads to an error state. Non-symbolic heuristic search strategies are

well studied. A* [15] is an advancement of Dijkstra's algorithm [8] for determining the shortest paths between two designated states within a graph.

The additional heuristic search information helps to avoid a blind breadth-first-search traversal but still suffers from the problem that a huge amount of states has to be stored. In this section an algorithm similar to A* is proposed to improve symbolic model checking.

4.1 *BDDA**

Edelkamp and Reffel have shown how *BDDs* help to solve heuristic single-agent search problems intractable for explicit state enumeration based methods [11]. The proposed algorithm *BDDA** was evaluated in the Fifteen-Puzzle and within Sokoban.

The approach exhibits a new trade-off between time and space requirements and tackles the most important problem in heuristic search, the overcoming of space limitations while avoiding a strong penalty in time. The experimental data suggests that *BDDA** challenges both breadth-first-search using *BDDs* and traditional *A**. Sokoban is intractable to be solved with explicit state enumeration techniques (unless very elaborated heuristics, problem graph compressions and pruning strategies are applied) and the Fifteen-Puzzle cannot be solved with traditional symbolic search methods. It is worthwhile to note that especially in the Sokoban domain only very little problem specific knowledge has been incorporated to regain tractability.

The approach was successfully applied in AI-planning [12]. The authors propose a planner that uses *BDDs* to compactly encode sets of propositionally represented states. Using this representation, accurate reachability analysis and backward chaining are apparently be carried out without necessarily encountering exponential representation explosion. The main objectives are the interest in optimal solutions, the generality and the conciseness of the approach. The algorithm is tested against a benchmark of planning problems and lead to substantial improvements to existing solutions. The most difficult problems in the benchmark set were only solvable when additional heuristic information in form of a (fairly easy) lower bound was given.

4.2 Heuristics and *A**

Let $h^*(s)$ be the length of the shortest path from s to a goal state and $h(s)$ its estimate. A heuristic is called *optimistic* if it is always a lower bound for the shortest path, i.e., for all states s we have $h(s) \leq h^*(s)$. It is called *consistent* if we have $h(u) \leq h(v) + 1$, with v being the successor of u on any solution path. Consistent heuristics are optimistic by definition and optimistic heuristics are also called *lower bounds*.

Heuristics correspond to a reweighting of the underlying problem graph. In the uniformly weighted graph we assign the following assignment to the edges $w(u, v) = 1 - h(u) + h(v)$. Fortunately, up to an additional offset the shortest paths values remain the same and no negative weighted loops are introduced.

Consistent heuristics correspond to a positively weighted graph, while optimistic heuristics may lead to negative weighted edges.

In A^* there are three sets. The set *Visited* of states already expanded, the set *Open* containing the states next to be expanded and the states which have not yet been encountered. During the calculation every state always belongs to exactly one of these sets. When a state is expanded it is moved from *Open* to *Visited* and all its successors are moved to *Open* unless they do not already belong to *Visited*. In this case they are inserted back to *Open* only if the current path is shorter than the one found before. This is done until the goal state is encountered or the set *Open* is empty. In the later case there exists no path between an initial state and a goal state. The correctness result of A^* states that given an optimistic estimate the algorithm terminates with the optimal solution length.

4.3 Tailoring *BDDA** for Model Checking

In the *BDD* version of A^* the set *Visited* is omitted. To preserve correctness the successors of the expanded states are always inserted into *Open*. This relates to the expansion of the entire search tree corresponding to the reweighted graph. The closely related explicit state enumeration technique is iterative deepening A^*, *IDA** for short [16]. With an increasing bound on the solution length the search tree is traversed in depth-first manner. Note, *IDA** was the first algorithm that solved the Fifteen Puzzle. The admissibility of BDDA* is inherited by the fact that Korf has shown that given an optimistic heuristic IDA* finds an optimal solution.

For model checking omitting the set *Visited* turns out not to be a good choice in general such that the option to update the set of visited states in each iteration has been reincarnated. In difference to A^*, however, the length of the minimal path to each state is not stored. The closest corresponding single-state space algorithm is *IDA** with transposition tables [22]. Transposition tables store already encountered states to determine that a given state has already been visited. This pruning strategy avoids so-called *duplicates* in the search. However it is necessary to memorize the corresponding path length to guarantee admissibility for optimistic heuristics. Fortunately one can omit this additional information when only consistent heuristics are considered. In this case the resulting cost-function obtained by the sum of path length g and heuristic value h is monotone.

The set *Open* is a priority queue sorted according to the costs of the states. The costs of a state s is the sum of the heuristic and the number of steps necessary to reach s. The priority queue *Open* can be symbolically represented as a *BDD* *Open*(costs, state) in which the variables for the binary representation of the costs have smaller indices than those for the representation of states. In Figure 1 the algorithm is represented in pseudo code. The *BDD Open* corresponds to a partitioning of the states according to their costs.

Due to the variable ordering a new *BDD* operation (not included in standard *BDD* libraries) might efficiently combine three steps in the algorithm: the determination of the set of states with minimal costs contained in the queue,

its costs f_{min}, and the new queue without these states. The function follows the path from the root node by always choosing the left successor – provided it does not directly lead to *false* – until the first state variable is encountered. This node is the root of *Min*, the persecuted path corresponds to the minimal costs f_{min}. The set *Open* excluding the just expanded states is obtained when *Min* is replaced by *false* probably followed by some necessary applications of the *BDD*-reduction rules.

Note, that the range of the costs has to be chosen adequately to avoid an overflow. To determine the set *Succ* the costs of the new states have to be calculated. As in *Open* only the costs of a state are stored and not the path-length the new costs are the result of the formula $f' - h' + 1 + h$ with f' and h' being the costs and the heuristic value of the predecessor. The value 1 is added for the effected transition and h is the estimate for the new state. Afterwards it remains to update the set *Visited* which is merged with *Min*. Furthermore the new states *Succ* are added to *Open* which should contain no states comprised in *Visited*.

Input *BDD Start* of the initial, *BDD Goal* of the erroneous states, *BDD Trans* representing the transition relation, and *BDD Heuristic* for the estimate of the entire search space.
Output "Error state found!" if the algorithm succeeds in finding the erroneous state, "Complete Exploration!", otherwise.

Visited(State s) := *false*
Open(Costs f, State s) := *Start*(s) ∧ *Heuristic*(f,s)
while (∃ s_1,f_1. *Open*(f_1,s_1))
  if (∃ s′,f′. *Open*(f′,s′) ∧ *Goal*(s′)) **return** "Error state found!"
  Min(f,s) := *Open*(f,s) ∧ f=f_{min}
  Succ(f,s) := ∃ f′,s′,h,h′. *Heuristic*(h,s) ∧ *Heuristic*(h′,s′) ∧
    Min(f′,s′) ∧ *Trans*(s′,s) ∧ f=f′−h′ + 1+h
  Visited(s) := *Visited*(s) ∨ ∃ f. Min(f,s)
  Open(f,s) := (*Open*(f,s) ∨ *Succ*(f,s)) ∧ ¬*Visited*(s)
return "Complete Exploration!"

Fig. 1. Heuristic based algorithm in Model Checking.

5 Inferring the Heuristic

The heuristic estimates the distance (measured in the number of transition steps) from a state to an error state. According to the type of the system such a step can have different meanings. For a synchronous system one step corresponds to one step in each module. In an asynchronous system a subset of all modules can perform a step and finally for an interleaving model exactly one module executes a transition. The challenging question is how to find a lower bound estimate (optimistic heuristic) for typical systems.

First of all, $h \equiv 0$ would be a valid choice, but in this case A^* exactly corresponds to breadth-first-search. Therefore, the values of the heuristic have to be positive to serve as an effective guidance in the search: The more diverse the heuristic values the better the classification of states. In this case we select most promising states for failure detection and distinguish them from the rest. As an effect in each iteration only a few states have to be expanded.

The next intuitive heuristic is the Hamming distance mentioned above. The measure is optimistic if in one transition only one x_i can change. Unfortunately, this is not true in general. The main drawback of this heuristic, however, is that in general the number of variables necessary to define an error state are few in comparison to the number of state variables. Hence, the Hamming distance typically has a small range of values and the number of different partitions of states are too less to significantly reduce the number of states to be expanded.

In the sequel we propose an automatic construction of a heuristic only based on the safety property and the structure of the transition function. We assume that the formula f describing the error states is a boolean formula using $\wedge$ and $\vee$ while negation is only applied directly to variables. In CTL the safety property with respect to the property f is denoted by $\mathrm{AG}(\neg f)$.

5.1 Definition

Table 1 describes the transformation of the formula f into a heuristic Heu_f. In the first two cases the sub-formulas f_k must not contain another $\vee$-operator (respectively $\wedge$) at the top level.

$$Heu_f(s) = \begin{cases} \min_{k=1,\ldots,n} Heu_{f_k}(s), & \text{if } f = f_1 \vee \cdots \vee f_n \\ \max_{k=1,\ldots,n} Heu_{f_k}(s), & \text{if } f = f_1 \wedge \cdots \wedge f_n \\ Heu_{s_{i,j}}(s), & \text{if } f = s_{i,j} \\ Heu_{\overline{s_{i,j}}}(s), & \text{if } f = \overline{s_{i,j}} \end{cases}$$

Table 1. Property-dependent determination of heuristic values.

With this construction the heuristic value depends only on $Heu_{s_{i,j}}(s)$ and $Heu_{\overline{s_{i,j}}}(s)$ which rely on the structure of the transition relation. As explained in Section 3.1 the transition of variable j in module i is described by $T_{i,j}(s, t_{i,j})$. The devices $T_{i,j}$ are typically some standard electronic elements such as the logical operators *or*, *and*, *xor*, etc. In a general setting, however, they can be arbitrary formulas.

Table 2 exemplarily depicts the values for the function $Heu_{s_{i,j}}$ for every binary boolean formula. For a general boolean function $s_{i,j} = g : B^n \mapsto B$ with n arguments the sub-function $Heu_{s_{i,j}}(s)$ has the following value:

$$\min_{x \in B^n | g(x)=1} \{ \max_{i \in \{1..n\}} \{\text{number of transitions necessary until } s_i = x_i\}\}$$

$g(x_1, x_2)$	$Heu_{s_{i,j}}(s) =$ if $(s_{i,j})$ then 0 else:
0	∞
$x_1 \wedge x_2$	$1 + \max\{Heu_{x_1}(s), Heu_{x_2}(s)\}$
$\neg(x_1 \rightarrow x_2)$	$1 + \max\{Heu_{x1}(s), Heu_{\overline{x_2}}(s)\}$
x_1	$1 + Heu_{x_1}(s)$
$\neg(x_2 \rightarrow x_1)$	$1 + \max\{Heu_{\overline{x_1}}(s), Heu_{x_2}(s)\}$
x_2	$1 + Heu_{x_2}(s)$
$x_1 \not\leftrightarrow x_2$	$1 +$ (if (x_1) $\min\{Heu_{\overline{x_1}}(s), Heu_{\overline{x_2}}(s)\}$ else $\min\{Heu_{x_1}(s), Heu_{x_2}(s)\}$)
$x_1 \vee x_2$	$1 + \min\{Heu_{x_1}(s), Heu_{x_2}(s)\}$

Table 2. Transition-dependent determination of heuristic values for $t_{i,j} = g(s_{i_1,j_1}, s_{i_2,j_2})$. The remaining 8 binary functions are obtained by duality.

Note, that Table 2 is valid for both an asynchronous and a synchronous model. In the case of an interleaving model the heuristic can be improved: if Heu_{x_1} and Heu_{x_2} appear only once in the whole formula and x_1 and x_2 belong to different modules then *max* can be replaced by *plus*.

5.2 Refinement-Depth

Actually, the rules can be applied to Heu_f infinitely often such that we have to limit the number of its applications and to define a base case.

Definition 1. *In a first step all possible rules of Table 1 are applied to* Heu_f. *The* refinement-depth *of the heuristic formula is the number of times all possible replacements given in Table 2 are applied.*

The rules are applied appropriately to reach the desired depth. Afterwards each remaining $Heu_{s_{i,j}}(s)$ is replaced by

$$\text{if } (s_{i,j}) \text{ then 0 else 1}$$

A higher refinement-depth corresponds to an improvement of the estimate, but on the other hand the *BDD* representing the heuristic becomes bigger and from a certain depth on the benefit from further refinements becomes very small or even disappears. So the aim is to find a trade-off between the *BDD*-size and the refinement-depth. In many cases already simple heuristics lead to a noticeable effect. Therefore, a feasible strategy can be to start with a simple heuristic and to refine it more and more until an error state is found.

5.3 Example

A part of an electronic circuit is given in Figure 2. Let $f = s_{1,0} \wedge s_{2,0}$ be the description of the error state. Table 3 demonstrates the construction of the heuristic for a refinement depth of 1 and 2.

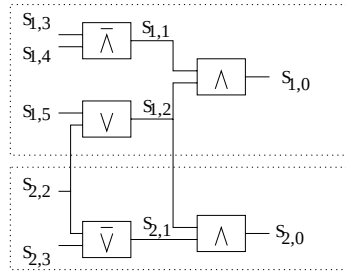

Fig. 2. Part of an electronic circuit

To show that our heuristic yields to a better partitioning of the state space by a wider range of heuristic values look at the following state $s = \{s_1, s_2\}$ with $s_1 = (1, 1, 1, 1, 1, 0)$ and $s_2 = (0, 0, 1, 0)$

As $s_{1,0}$ is *true* and $s_{2,0}$ is *false* the Hamming distance is 1. Our construction of the heuristic takes into account that it is not possible to reach a state where $s_{2,0} = true$ with one transition. Using a refinement depth of 2 leads to $\mathrm{H}_{s_{1,0} \wedge s_{2,0}}(s) = 3$. The variable $s_{2,2}$ has to become *false* before the *nor*-element can change the value of $s_{2,1}$ and in the third transition $s_{2,0}$ can switch to *true*.

x	$Heu_x(s)$ for $x =$*true* in s	$Heu_x(s)$ for $x =$*false* in s
For refinement depth 1:		
$s_{1,0}$	0	$1 + \max\{\mathrm{H}_{s_{1,1}}, \mathrm{H}_{s_{1,2}}\}$
$s_{2,0}$	0	$1 + \max\{\mathrm{H}_{s_{1,2}}, \mathrm{H}_{s_{2,1}}\}$
For refinement depth 2:		
$s_{1,1}$	0	$1 + \min\{\mathrm{H}_{\overline{s_{1,3}}}, \mathrm{H}_{\overline{s_{1,4}}}\}$
$s_{1,2}$	0	$1 + \min\{\mathrm{H}_{s_{1,5}}, \mathrm{H}_{s_{2,2}}\}$
$s_{2,1}$	0	$1 + \max\{\mathrm{H}_{\overline{s_{2,2}}}, \mathrm{H}_{\overline{s_{2,3}}}\}$
Base cases:		
$s_{1,5}, \overline{s_{1,3}}, \overline{s_{1,4}}$	0	1
$s_{2,2}, \overline{s_{2,2}}, \overline{s_{2,3}}$	0	1

Table 3. Example of the heuristic estimate.

5.4 Properties

As indicated the heuristic can be improved to allow a better partitioning of the set of states to be expanded. A non-optimistic heuristic can lead to a faster detection of an erroneous state but on the other hand it can increase the length of the counter example. The construction of the heuristic however always leads to an optimistic heuristic. To prove this we will use the following lemma:

Lemma 1. $\mathrm{Heu}_{s_{i,j}}(s)$ *is a lower bound for the number of transitions which are necessary to reach a state from* s *where* $s_{i,j} = true$.

Proof. The property will be shown by induction on the refinement depth:

Refinement depth 0: In this case we have $Heu_{s_{i,j}} = \text{if } (s_{i,j}) \text{ then } 0 \text{ else } 1$. (The argumentation for negated variables $\overline{s_{i,j}}$ is similar.) If $s_{i,j}$ is *true* no transition step is necessary. Hence, $Heu_{s_{i,j}}(s) = 0$. In case $s_{i,j}$ is *false* at least one transition step has to be made to change its value. Therefore, $Heu_{s_{i,j}}(s) = 1$ is a lower bound.

Refinement depth k: We will suppose that for a refinement depth less than k the functions $Heu_{s_{i,j}}$ fulfill the desired property. After the first application of a rule from Table 2 for the introduced formulae Heu_{x_i} we have a refinement depth of $k-1$. This implies that it takes at least Heu_{x_i} steps to change the values of the involved variables. It depends on the formula $g(x_1, x_2)$ which changes the value of $s_{i,j}$ that it is necessary for both variables x_1 and x_2 to have a certain value, or that it is sufficient that one of them has a certain value. This determines if the minimum or the maximum of the involved functions Heu_{x_i} is used. In both cases after the variables x_1 and x_2 have been assigned to a value, which allows g to change the value of $s_{i,j}$ from *false* to *true*, at least one additional step is necessary. Therefore, 1 can be added and $Heu_{s_{i,j}}$ still remains a lower bound. ∎

Using Lemma 1 it it quite easy to prove that Heu_f according to the construction introduced above is a lower bound estimate.

Theorem 1. *The function* Heu_f *is optimistic.*

Proof. It remains to show that the rules of Table 1 lead to an optimistic heuristic since the sub-functions $Heu_{s_{i,j}}$ underestimate the number of transitions necessary to achieve the desired value for $s_{i,j}$. This can be shown easily by induction on the number of applications of the rules of Table 1 so we will only explain the main idea for the proof.

It is based on the fact that for an $\vee$-formula it is sufficient that one of the f_i becomes *true*, so the minimum of the Heu_{f_i} is chosen and for an $\wedge$-formula all f_i have to be fulfilled so the maximum of the Heu_{f_i} can be chosen for the heuristic value. ∎

Note, that for an asynchronous or a synchronous system in one transition step the values of various variables can change, therefore it is not possible to summarize over the Heu_{f_i} for example in case of an $\wedge$-formula. In contrast, for an interleaving model the sum could be used if the f_i depend on variables in different modules because only one module can change its state in a single transition.

As already indicated to guarantee the computation of the minimal counterexample in the proposed extension to *BDDA** it is not sufficient to use an optimistic heuristic. Fortunately, it is possible to show the consistency of our automatically constructed heuristic:

Theorem 2. *The function* Heu_f *is consistent.*

Proof. We have to show that

$$\forall State\ s,t.Trans(s,t) \Rightarrow Heu_f(s) \leq 1 + Heu_f(t).$$

This property follows directly from the fact that for all variables x we have

$$\forall State\ s,t.Trans(s,t) \Rightarrow Heu_x(s) \leq 1 + Heu_x(t).$$

We will prove this by induction on the refinement depth similar to Lemma 1. For a refinement depth of 0 there is nothing to prove because $Heu_x(s) \leq 1$. For refinement depth k we will show the property for the operator $\wedge$ (cf. Table 2). In this case $Heu_x(s)$ is defined as $1 + \max\{Heu_{x_1}(s), Heu_{x_2}(s)\}$. For Heu_{x_1} and Heu_{x_2} we have a refinement depth of $k-1$ so the property holds for these formulas:

$$\begin{aligned} Heu_x(s) &\leq 1 + \max\{1 + Heu_{x_1}(t), 1 + Heu_{x_2}(t)\} \\ &\leq 1 + (1 + \max\{Heu_{x_1}(t), Heu_{x_2}(t)\}) \\ &\leq 1 + Heu_x(t) \end{aligned}$$

The proof for the other operators of Table 2 is analogue expect for the operator $\not\leftrightarrow$. The interesting case is a transition where in state s the variables x_1 and x_2 are assigned to *true* and in state t both variables are assigned to *false*. In this case the structure of the formula changes: For state s we have

$$Heu_x(s) = 1 + \min\{Heu_{\overline{x_1}}(s), Heu_{\overline{x_2}}(s)\}$$

and in state t we establish

$$Heu_x(t) = 1 + \min\{Heu_{x_1}(t), Heu_{x_2}(t)\}.$$

The circumstance that there is a transition from s to t which changes the values of x_1 and x_2 from *true* to *false* implies that $Heu_{\overline{x_1}}(s) = Heu_{\overline{x_2}}(s) = 1$ and $Heu_x(s) = 2$ while for state t we have $Heu_{x_1}(t),\ Heu_{x_2}(t) \geq 1$. Therefore, the following equation completes the proof:

$$Heu_x(t) = 1 + \min\{Heu_{x_1}(t), Heu_{x_2}(t)\} \geq 1 + \min\{1,1\} = 2 = Heu(s)$$

■

Note, that breadth-first-search finds the error state in the minimal number of iterations. In contrast in the heuristic search approach several states remain unexpanded in each iteration such that the number of necessary iteration steps increases. In the worst case we have a quadratic growth in the number of iterations [11]. On the other hand, especially for large systems, a transition step expanding only a small subset of the states is much faster than a transition based on all states. Therefore, this apparent disadvantage even turns out to be very time-efficient surplus as the examples in the next section will show.

6 Experimental Results

In our experiments we used the μ-calculus model checker μcke [3] which accepts the full μ-calculus for its input language [20]. The *while*-loop has to be converted into a least fix-point. As it is not possible to change two sets (*Open, Visited*) in the body of one fix-point the *Visited* set is simulated by one slot in the *BDD* for *Open*. The next problem is that the function for *Open* is not monotone because states are deleted from it after they have been expanded. Monotony is a sufficient criterion to guarantee the existence of fix-points. The function for *Open* is not a syntactic correct μ-calculus formula but as the termination of the algorithm is guaranteed by the monotony of the set *Visited* we can apply the standard algorithm for the calculation of μ-calculus fix-points.

Unfortunately, we cannot take advantage of the special *BDD* operation determining the set of states with minimal costs in this case. These calculations have to be simulated by standard operations leading to some unnecessary overhead that in the visible future has to be avoided in a customized implementation.

For the evaluation of our approach we use the example of the tree-arbiter [9] a mechanism for distributed mutual exclusion: $2n$ user want to use a resource which is available only once and the tree-arbiter manages the requests and acknowledges avoiding a simultaneous access of two different users. The tree-arbiter consists of $2n-1$ modules of the same structure such that it is very easy to scale the example. Since we focus on error detection we experiment with an earlier incorrect version – also published in [9] – using an interleaving model.

The heuristic was devised according to the description in Section 5 with a refinement-depth of 6. We also experimented with larger depths which implied a reduction neither in time nor in size. Since the algorithm for the automatic construction of the heuristic has not yet been implemented and since the number of different errors increases very fast with the size of the tree-arbiter we searched for the detection of a special error case. Table 4 shows the results in comparison with a classical forward breadth-first-search. To guarantee the fairness of the comparison we terminated the search at the time the error state has first been encountered.

For the tree-arbiter with 15 modules or less the traditional approach is faster and less memory consuming, but for larger systems its time and memory efficiency decreases very fast. On the other hand, the heuristic approach found the error even in large systems, since its memory and time requirements increases slowly. For the tree-arbiter with 23 modules the error could not be found with breadth-first-search and already for the version with 21 modules 9 garbage collections were necessary not to exceed the memory limitations, whereas the first garbage collection with the heuristic method was invoked at a system of 27 modules. For the tree-arbiter with 27 modules we also experimented with the heuristic. When we double its values the heuristic fails to be optimistic, but the error detection becomes available without any garbage collection. Moreover, although more than three times more iterations were necessary only about 8% more time was consumed. This illustrates that there is much room for further research in refinements of the heuristic.

	BFS			Heuristic			
# Mod	#it	max nodes	time	depth	#it	max nodes	time
15	30	991374	46s	4	104	10472785	483s
				6	127	5715484	288s
17	42	18937458	3912s	6	157	7954251	476s
19	44	22461024	6047s	6	157	8789341	540s
21	44	26843514	24626s(9)	6	157	9097823	530s
23	>40	-	>17000s	6	157	9548269	516s
25	-	-	-	6	169	21561058	1370s
27	-	-	-	6	169	25165795	1818s(1)
				6(x2)	593	23798202	1970s

Table 4. Results for the tree-arbiter. In parenthesis the number of garbage collections is given.

The second example we used for the evaluation of our approach is the asynchronous DME [9]. Like the tree-arbiter it consists of n identical modules and it is also a mechanism for distributed mutual exclusion. The modules are arranged in a ring structure whereas the modules of the tree-arbiter form a pyramid. In this case we also experimented with the set *Visited* and it turns out that it was more efficient to omit it like proposed in [11]. For this variation only a small change in the calculation of *Open* is necessary. Like in the previous example the results in Table 5 show that the heuristic approach is more memory efficient and less time-consuming. The first experiment in the Table uses the set *Visited* that was omitted in the other experiments. This led to a greater iteration depth because several states are visited more than once. Nevertheless this turned out to be more time and memory efficient. The increase of the refinement-depth to 7 allows to reduce the verification time and no garbage collection remains necessary.

	BFS			Heuristic			
# Mod	#it	max nodes	time	depth	#it	max nodes	time
6	23	26843514	5864s(5)	6v	35	29036025	2207s(4)
				6	53	25165795	1009s(1)
				7	53	25159862	813s(0)

Table 5. Results for the asynchronous DME. In parenthesis the number of garbage collections is given.

7 Conclusion and Discussion

We presented how a heuristic can successfully be integrated into symbolic model checking. It is recommended to distinguish between the use of a model checker in order to prove a property and the use as a debugging tool. For debugging exhaustive search of the reachable state space can be avoided and the heuristic can decrease both the number of expanded states and the *BDD*-sizes which allows the treatment of bigger systems. It was shown how a heuristic can be automatically designed for a large class of systems allowing the application of this method also for non-experts.

The experiments demonstrated the effectiveness of the approach and we plan to test the algorithm with more example data and to evaluate further refinements of the heuristic and its construction.

There are lots of choices for an experienced user to modify and improve the estimate or even to use non-optimistic heuristics allowing a better partitioning of the state space. This can be more important than the determination of the minimal counter-example. Pearl [21] discusses limits and possibilities of overestimations in corresponding explicit search algorithms. One proposed search scheme, called *WIDA**, considers costs of the form $f = \alpha g + (1-\alpha)h$. If $\alpha \in [.5, 1]$ the algorithm is admissible. In case $\alpha \in [0, .5)$ the algorithm searches according to overestimations of the heuristic value compared to the path length g. The literature clearly lacks theoretical and practical results for symbolic searches according to non-optimistic heuristics.

Acknowledgments

F. Reffel is supported by DFG within the graduate program on *Controllability of Complex Systems*. S. Edelkamp is associated member in a DFG project entitled *Heuristic Search and Its Application in Protocol Verification*.

We thank the anonymous referees for the interesting comments and suggestions.

References

[1] R. Alur, R. Brayton, T. Henzinger, S. Qaderer, and S. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, volume 1254 of *LNCS*, pages 340–351, 1997.

[2] A. Biere. *Effiziente μ-Kalkül Modellprüfung mit Binären Entscheidungsdiagrammen*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 1997.

[3] A. Biere. mucke - efficient μ-calculus model checking. In *Computer Aided Verification*, volume 1254 of *LNCS*, pages 468–471, 1997.

[4] 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*, 1999. to appear.

[5] R. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[6] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency*, volume 803 of *LNCS*, pages 124–175. REX School Symposium, Springer, 1993.

[7] R. Cleaveland. Tableau-based model checking in the propositional μ-calculus. *Acta Inf.*, 27:725–747, 1990.

[8] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1(5):269–271, 1959.

[9] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* An ACM Distinguished Dissertation. The MIT Press, 1988.

[10] J. F. Dillenburg and P. C. Nelson. Perimeter search (research note). *Artificial Intelligence*, 65(1):165–178, Jan. 1994.

[11] S. Edelkamp and F. Reffel. OBDDs in heuristic search. In O. Herzog and A. Günter, editors, *Proceedings of the 22nd Annual German Conference on Advances in Artificial Intelligence (KI-98)*, volume 1504 of *LNAI*, pages 81–92. Springer, 1998.

[12] S. Edelkamp and F. Reffel. Deterministic state space planning with BDDs. Technical Report 120, Institut für Informatik, Universität Freiburg, 1999.

[13] E. A. Emerson and J. Srinivasan. Branching time temporal logic. In *REX workshop*, volume 354 of *LNCS*, pages 123–172. Springer-Verlag, 1989.

[14] J. C. Fernandez, L. Mounier, C. Jard, and T. Jéron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1:251–273, 1992.

[15] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, SCC-4(2):100–107, 1968.

[16] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. reprinted in Chapter 6 of *Expert Systems, A Software Methodology for Modern Applications*, P.G. Raeth (Ed.), IEEE Computer Society Press, Washington D.C., 1990, pp. 380-389.

[17] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[18] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New York, 1985. ACM.

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

[20] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science, 5th GI Conference*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.

[21] J. Pearl. *Heuristics : Intelligent search strategies for computer problem solving.* Addison-Wesley series in Artificial Intelligence. Addison-Wesley, 1984.

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

[23] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 139–144. IEEE, 1993.

[24] C. Stirling and D. Walker. Local model checking in the modal μ-claculus. *Theoretical Computer Science*, 89:161–177, 1991.

[25] C. H. Yang and D. Dill. Validation with guided search of the state space. In *35nd Design Automation Conference*, 1998.