Abstraction-Guided Model Checking Using Symbolic IDA* and Heuristic Synthesis

Kairong Qian, Albert Nymeyer, and Steven Susanto

School of Computer Science & Engineering,
The University of New South Wales, Sydney Australia
{kairongq, anymeyer, ssus290}@cse.unsw.edu.au

Abstract. A heuristic-based symbolic model checking algorithm, BDD-IDA* that efficiently falsifies invariant properties of a system is presented. As in bounded model checking, the algorithm uses an iterative deepening search strategy. However, in our case, the search strategy is guided by a heuristic that is computed from an abstract model, which is derived from the concrete model by a synthesis technique. Synthesis involves eliminating so-called weak variables from the concrete specification, where the weak variables are identified by a data-dependency analysis. Unique to this work is the use of the depth-first IDA* search algorithm in a BDD setting, and the automatic synthesis of the heuristic. The performance of the approach on a large number of small examples is compared with standard BDD-based approaches. Experiments on a variety of real-world models from different domains are also conducted. The approach reveals a consistent improvement on all models, and in some cases a speed-up of 2 orders of magnitude is obtained.

Keywords: Formal verification, symbolic model checking, heuristic search, data abstractions, model approximations.

1 Introduction

Model checking [7] is often used in preference to theorem proving for the verification of properties in finite-state systems because of its high level of automation as well as its ability to produce counter-examples when a given property is found not to hold. The safety properties of a system can often be captured by one or more system invariants that characterise the set of states within which the system must reside. This process of checking invariants is also called a reachability analysis. The aim of a reachability analysis is to detect error states, where the paths leading to these states determine counter-examples to the invariant. Counter-examples provide valuable information to system developers about potential design errors in a system. In this work we are more focussed on falsification of invariants than verification.

This work is based on the symbolic model checking approach [15] in which symbolic data structures called binary decision diagrams (BDDs) [2] are used

F. Wang (Ed.): FORTE 2005, LNCS 3731, pp. 275–289, 2005.
© IFIP International Federation for Information Processing 2005

to represent a finite-state space. Invariant checking in symbolic model checking is usually done by either BDD-based or SAT-based algorithms. The BDD-based algorithm, BDD-BFS, conducts a breadth-first search on the system state space and records all reachable states. The goal for the search algorithm usually captures those states in which the invariants are violated. Being 'blind', BDD-BFS is an inefficient way to find error states as typically many regions of the state space will needlessly be searched. A BFS strategy is more suited to correct models and is wasteful of space as generally all reachable states need to be stored. As well, for large systems, the sizes of BDDs in BDD-BFS can grow exponentially, making state space exploration almost impossible in realistic cases.

An alternative symbolic model checking approach is called bounded model checking[1]. Bounded model checking translates the bounded semantics of the invariant into Boolean expressions and uses SAT procedures to determine their satisfiability. By incrementally increasing the bound, the algorithm iteratively deepens the state space exploration. If an error state is encountered at some level, the algorithm will terminate and report a counter-example. In general, SAT-based algorithms tend to detect counter-examples quicker than BDD-BFS due to the inherent DFS search strategy that SAT solvers use [5] if the counter-examples are short. SAT-based approaches, however, can be handicapped by a huge number of clauses that need to be input to the SAT solver. In our work, we combine aspects of both techniques by using BDDs to represent the state space and a heuristic DFS strategy to locate error states.

Because a BDD-based, depth-first search (BDD-DFS) strategy is not naturally layered like BFS, it requires a special mechanism to partition each BDD frontier during the search. The integration of BDD-DFS with heuristic search provides this mechanism: the heuristic values of states are not only used to estimate the distances to the goal, they are also used to partition the frontier into sub-frontiers. Each sub-frontier, represented by a single BDD, is treated by BDD-DFS as a single node in the search graph.

Our integration of heuristics and symbolic data is yet another development in the growing field of guided model checking [10,17,19,20] whose aim is to apply 'smart' technology to model verification. In previous work [17], we integrated the A* search algorithm into a symbolic model checker. In this paper, we instead use a 'more efficient' version of A* called IDA* (iterative deepening A*), designed to minimise memory usage. A* is in essence a mixture of BFS and DFS. If the heuristic is informative, the search is more like DFS, otherwise, with poor direction, the search works on a broad front. The tendency to mimic BFS when poorly informed means that A* can have exponential memory requirements. In 1985, Korf [13] devised IDA* that is basically DFS, but has some BFS characteristics. He found this algorithm was often better than A* in solving hard AI problems [14].

The new, integrated algorithm we develop is called BDD-IDA*. The advantages of using BDD-IDA* are:

1. The iterative and bounded DFS strategy in IDA* detects so-called shallow and corner bugs that are difficult to detect by unbounded DFS.

2. In BFS, the frontier is layered. In A*, the frontier is typically 'onion shaped' because of the action of the heuristic (biasing the search towards a particular path that leads to a goal state). Pruning of the search space in fact occurs before the bound is reached, so the frontier is more pointed to the goal for each iteration.
3. IDA* has the same linear (in the search depth) space complexity as DFS. (But note, as we use BDDs to represent sets of states, the actual space requirement can be exponential in the number of states.)

A second important feature of our work is that we have extended the idea presented in earlier work [17] of using an abstract version of the concrete model as a heuristic (the so-called pattern database). In that work we did not address the problem of how to obtain the abstract model. In this work, we address this issue and present an automatic method to generate abstract models that is based on a data-dependency analysis of the concrete model specification. In this analysis, we determine the *strength* of each variable. This information is used by the heuristic generator to eliminate those variables that are considered less relevant (or weak), and thereby reduce the size of the model. We refer to this technique as *heuristic synthesis*. Being able to automatically determine a heuristic frees the system designer/verifier from this task, and makes the guided model checker fully automatic.

In summary, a number of model-checking and artificial intelligence approaches have been combined to produce an integrated, fully automatic framework that allows more efficient property falsification than alternative approaches. The rest of paper is organised as follows. Section 2 reviews the guided model checking framework we use and presents the BDD-based IDA* algorithm. In Section 3 the three-phase heuristic synthesis procedure based on abstractions is illustrated. We describe the tool that we have developed and the experiments in Section 4. Section 5 discusses related work from the literature and Section 6 concludes this work.

2 Guided Model Checking and Symbolic IDA*

The general framework for our abstraction-guided approach is based on work presented in [17]. We depict our approach in Figure 1. The process starts with the concrete model. In the first step we generate, automatically, a data abstraction of the concrete model using a data dependency analysis. The abstract model is taken as input by a standard model checker that uses a BFS search algorithm. If the model checker verifies the abstract model successfully, we terminate. If the abstract model fails the verification, we construct a heuristic using the abstract model. The guided model checking algorithm is then invoked to check the concrete system using the heuristic as guide. The outcome of the guided model checker is either that the concrete model is verified, or that a counter-example (*CX* in the figure) is produced. The approach in this paper differs from [17] in two aspects. 1) We use an automatic heuristic synthesis procedure, and 2) as our primary focus is on falsification and not verification, we use a DFS-based heuristic search algorithm, IDA*, and modify it to use BDDs.

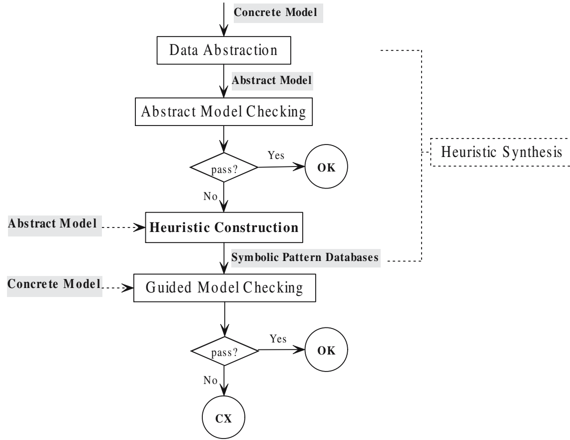

Fig. 1. The abstraction-guided model checking framework

The algorithm, called BDD-IDA*, is based on the explicit IDA* algorithm. The algorithm takes four inputs. The inputs $\mathcal{S}_0$ and $\mathcal{G}$ of the algorithm are BDDs representing the initial set of states and the goal, i.e. set of "bad" states. The input BDD $\mathcal{R}$ is the transition relation. Note that we denote the calculation of the image of a given set of states $\mathcal{S}$ by $\mathcal{R}(\mathcal{S})$. The input σ is a heuristic that will be illustrated in next section. Finally, the input *Bound* determines the search depth. We use an explicit stack where each element in the stack is of the form $(g, h, \mathcal{S})$. The integer g indicates the actual number of transitions from $\mathcal{S}_0$ to $\mathcal{S}$ and h the heuristic estimation of number of transitions from $\mathcal{S}$ to $\mathcal{G}$. In line 10, the algorithm calls the procedure **SplitAndPush**. This procedure uses the heuristic σ to partition a set of states $\mathcal{S}$ (that constitute the frontier of the IDA* search) into subsets, and together with their associated costs g and h, pushes each subset onto the stack. We show this procedure in the next session.

Procedure BDD-IDA* $(\mathcal{S}_0, \mathcal{R}, \mathcal{G}, \sigma, Bound)$

1 $\quad stack.push() \leftarrow (0, 0, \mathcal{S}_0)$
2 $\quad counter \leftarrow 1$
3 $\quad$ **while** $(counter \leq Bound)$ **do**
4 $\qquad$ **while** $(stack \neq \phi)$ **do**
5 $\qquad\quad (g, h, \mathcal{S}) \leftarrow stack.pop()$
6 $\qquad\quad$ **if** $(\mathcal{S} \wedge \mathcal{G} \neq \phi)$
7 $\qquad\qquad$ **return** *Bound*
8 $\qquad\quad$ **if** $(h + g < counter)$
9 $\qquad\qquad \mathcal{S} \leftarrow \mathcal{R}(\mathcal{S})$
10 $\qquad\qquad$ **SplitAndPush**$(g, \mathcal{S}, \sigma)$
11 $\qquad counter \leftarrow counter + 1$
12 $\qquad stack.push() \leftarrow (0, 0, \mathcal{S}_0)$
13 $\quad$ **return** *NoErrorInBound*

Note that we do not memorise the set of reachable states in the algorithm as we are only interested in the falsification of invariant properties.

3 Heuristic Synthesis

In this section we will outline how the heuristic σ is synthesised. It is a three-phase process. (1) Abstraction: a data dependency analysis is used to automatically define an abstraction function for the concrete model. (2) Approximation: an approximation of the abstract model is then computed in order to avoid the computational penalty for exact abstraction. (3) Heuristic Construction: a standard BDD-BFS algorithm is used to compute all reachable frontiers in the approximate model. The result of this synthesis is a set of BDDs $\sigma = \{B_1, B_2, \ldots, B_n\}$ where each B_i represents a set of states in the abstract model with the same heuristic value.

Phase 1: Abstraction

Let $M = (S, R, S_0)$ denote a concrete model where S is a set of states, $S_0 \subseteq S$ is a set of initial states and R is the transition relation. Let $H : S \to \hat{S}$ be a surjection that maps the concrete state space onto an abstract space $\hat{S}$ with $|\hat{S}| \leq |S|$. H therefore induces an abstract model that is defined as follows.

Definition 1 (Abstraction). *The abstraction of M w.r.t. H is denoted by $\hat{M} = (\hat{S}, \hat{R}, \hat{S_0})$, where*

- $\hat{S} = \{\hat{s} | s \in S \wedge \hat{s} = H(s)\}$
- $\hat{S_0} = \{\hat{s} | \hat{s} \in \hat{S} \wedge \hat{s} = H(s) \wedge s \in S_0\}$
- $\hat{R} \subseteq \hat{S} \times \hat{S}$ *is a transition relation, where* $(\hat{s_1}, \hat{s_2}) \in \hat{R}$ *iff* $\hat{s_1} = H(s_1) \wedge \hat{s_2} = H(s_2) \wedge \exists s_1 \exists s_2 (s_1, s_2) \in R$

In symbolic model checking, S_0 and R of M are usually represented by two first-order formulas, $\mathcal{F}_0(x_1, x_2, ..., x_n)$ and $\mathcal{F}_R(x_1, x_2, ..., x_n, x'_1, x'_2, ..., x'_n)$, where $\{x_1, x_2, ..., x_n\}$ and $\{x'_1, x'_2, ..., x'_n\}$ are variables that represent the current state and next state of the model. Without loss of generality we assume all variables range over same domain D, hence the state set of M is $S = D \times D \times ... \times D$. Let $\{\hat{x}_1, \hat{x}_2, ..., \hat{x}_n\}$ and $\{\hat{x}'_1, \hat{x}'_2, ..., \hat{x}'_n\}$ be variables that represent the current state and next state of $\hat{M}$. We denote $H(x_i) = \hat{x}_i$ iff H maps every value of x_i to an abstract value of $\hat{x}_i$. Let $\hat{\mathcal{F}_0}$ and $\hat{\mathcal{F}_R}$ denote the formulas that represent $\hat{S}_0$ and $\hat{R}$ respectively. Using the quantification on $\mathcal{F}_0$ and $\mathcal{F}_R$, we construct $\hat{S}_0$ and $\hat{R}$ by evaluating the following formulas.

1. $\hat{\mathcal{F}_0} = \exists x_1 ... \exists x_n (H(x_1) = \hat{x}_1 \wedge ... \wedge H(x_n) = \hat{x}_n \wedge \mathcal{S}_0(x_1, ..., x_n))$
2. $\hat{\mathcal{F}_R} = \exists x_1 ... \exists x_n \exists x'_1 ... \exists x'_n (H(x_1) = \hat{x}_1 \wedge ... \wedge H(x_n) = \hat{x}_n \wedge H(x'_1) = \hat{x'}_1 \wedge ... \wedge H(x'_n) = \hat{x'}_n \wedge \mathcal{R}(x_1, ..., x_n, x'_1, ..., x'_n))$

To build the abstraction from a concrete model, we first need to define H. In BDD-based symbolic model checking, every variable of the concrete model is encoded using a set of Boolean variables. Let V be a set of Boolean variables

that encode all the variables in M. Following [5,17], we define H by restricting a subset $V_{inv} \subset V$ to a single-element domain, i.e. $H(\mathcal{F}) = \exists v_0 ... \exists v_m \mathcal{F}$ for all $v_i \in V_{inv}$, where $\mathcal{F}$ is formula representing a set of concrete states. This abstraction essentially makes the variables in V_{inv} invisible. Note that in our earlier work [17], the user had to provide V_{inv} to build the abstract model; possibly a difficult task. In this paper we describe an automated method to generate V_{inv} that is based on a data dependency analysis.

Data Dependency Analysis. The aim of this analysis is to estimate the strength of each variable v_i in V, and remove weak variables to form an abstract model. The analysis is based on the cone of influence (COI) abstraction techniques [7]. Let $V_p \subseteq V$ be a set of variables that appears in the specification φ. The COI of V_p, denoted by C, is the minimal set of variables such that:

- $V_p \subseteq C$
- if for some $v_i \in C$ its $next(v_i)$ depends on v_j, then $v_j \in C$

If $|C| < |V|$, then we construct a reduced model M' that only contains variables in C. It is proved in [7] that the reduced and original models form a bi-simulation relation w.r.t. all CTL specifications that only have variables from V_p, i.e. $M' \models \varphi \leftrightarrow M \models \varphi$. As a result, model checking can be performed on the reduced model. Of course, every variable in C must be included, otherwise the reduced model is not bi-similar.

We use abstraction only to synthesise the heuristics that guide the model checker of the concrete model, so we do not need to restrict ourself to removing only the variables outside of C (unlike [7]). Although all variables in C can influence the variables in φ, the degree of influence will not normally be the same. Some variables in V_p are more strongly influenced by variables in $C - V_p$ than others. To determine the subset of variables of V on which the truth of φ is heavily reliant, we build a dependency tree. Let $D(v)$ be the positive integer denoting the distance from v to the root of the dependency tree. The smaller $D(v)$ is, the stronger the influence of v on φ. The following algorithm computes $D(v)$ for all $v \in C$.

initialise $i := 0$, $C := V_p$ and $V_t := V_p$;
while C changes do
  $i := i + 1$;
  for each $v_i \in V_t$, compute all its dependable variables;
    put those who are not in C into V_{tt};
    assign $D(v) := i$ for all $v \in V_{tt}$;
    assign $C := C \cup V_{tt}$ and $V_t := V_{tt}$;

To determine V_{inv} we need to set the threshold d for $D(v)$, and compute $V_{inv} := (V - C) \bigcup \{v | v \in C \wedge D(v) \geq d\}$, where $(V - C)$ are all variables that are outside of the COI. (In our tool the value of d is a run-time option.)

Phase 2: Approximation

Having defined H, we need to evaluate $\hat{\mathcal{F}}_0$ and $\hat{\mathcal{F}}_R$ in order to compute $\hat{S}_0$ and $\hat{R}$ for $\hat{M}$. We could evaluate them directly, i.e. quantifier elimination. For asyn-

chronous systems, $\hat{\mathcal{F}}_R$ is usually made up of a disjunction of transition blocks and existential quantification can distribute over them. Synchronous models however consist of conjunctions of small transition blocks and existential quantifiers do not distribute over them. This means that we need to build a monolithic BDD for the formulae $\mathcal{F}_0$ and $\mathcal{F}_R$ and then perform quantifier elimination on them. This is computationally very expensive, especially in the case of $\hat{\mathcal{F}}_R$.

This problem can be avoided if we allow the existential quantifiers to distribute over the conjunctions. In other words, we want to push the quantification to the small transition blocks. This computation can be relatively easy because the blocks are often small. But of course the resulting model is not $\hat{M}$ anymore, but an *approximation* of it, which we denote $\hat{M}_{app}$. It is proved in [6] that this approximation does not cause the loss of any initial states and transitions, i.e. $\hat{M}_{app}$ simulates $\hat{M}$ (see [6] or [16] for the definition of a simulation relation for Kripke structures). By transitivity of the simulation relation, $\hat{M}_{app}$ simulates M because $\hat{M}_{app}$ simulates $\hat{M}$ and $\hat{M}$ simulates M [16]. In order to show the correctness of the mechanism, we need to show that $\hat{M}_{app}$ contains the information that we can use to estimate the length of counter-examples of M. This we do in the following lemma.

Lemma 1. *If a state $s \in S$ is reachable in M from any state in S_0, then its abstract counterpart $\hat{s}$ is also reachable in $\hat{M}_{app}$ from any state in $\hat{S}_0$.*

The proof of this lemma is omitted. In essence, this lemma implies that if a counter-example is present in the original model, it must be manifest in the abstract model $\hat{M}$ as well as in the approximation $\hat{M}_{app}$. Note that both the abstraction and approximation do not 'lose' any transitions of the original system, although internal transitions with one abstract state are not visible in $\hat{M}$. Thus, the admissibility of the approach will therefore not be affected (see [17]). This guarantees the resulting counter-example will be the shortest.

Phase 3: Constructing a Heuristic

The purpose of a heuristic is to estimate the number of transitions from each concrete state to a goal state (or error state). The heuristic value for each state $s \in S$ in M is simply the number of transitions from $\hat{s} = H(s)$ to the abstract goal state in $\hat{M}_{app}$. This type of heuristic is usually referred to as a *pattern database* [8,11,17], where *pattern* is another term for abstraction. The term *database* means the heuristic is a memory-based heuristic that can be handled by a hash table, where the indices represent abstract states and the entries are heuristic values. As in our earlier work [17], we use a set of BDDs to store the heuristic, called symbolic pattern databases (SPDB) and denoted by $\sigma = \{B_1, B_2, ..., B_n\}$. Note that the set of states represented by these BDDs are disjoint, i.e. $B_1 \wedge B_2 \wedge ... \wedge B_n = \phi$. Each B_i represents a set of abstract states that have the same heuristic value, and hence have the same number of transitions to the abstract goal state. A SPDB can be constructed using both backward and forward blind BFS search in $\hat{M}_{app}$:

Backward Construction. Use a BDD-based BFS strategy to explore $\hat{M}_{app}$, and start at the abstract goal. Put each frontier-BDD into the heuristic hash table with the number of iterations as the entry in SPDB. Terminate if an abstract initial state is encountered.

Forward Construction. Instead, start at an abstract initial state, and store each frontier temporarily. If the search detects the abstract goal, then extract the path backward from the goal to the initial state. The set of BDDs that comprises the path is a forward SPDB. This process is the same as the counter-example extraction in standard invariant checking. The paths generated here are a subset of the paths extracted by backward SPDB.

It is of course possible that the heuristic synthesis procedure cannot find a trace in the approximation $\hat{M}_{app}$. In that case the original model M does not have a counter-example (by Lemma 1).

Splitting the BDD. Let $\sigma = \{B_1, B_2, \ldots, B_n\}$ be the heuristic (SPDB) that is synthesised by the the 3-phase process described above. The following algorithm splits a BDD into several BDDs that are the disjoint subsets of the original set of states. In order to contain the BDD size after splitting, we use the *restrict* operator on BDDs, denoted by $\downarrow$. Note the subscript i of each B_i indicates the number of transitions B_i to the error state in the abstract model. The heuristic of a concrete state is simply the value of i of its corresponding abstract state and is used by BDD-IDA* to prioritise the state space search and hence for efficient error detection.

Procedure SplitAndPush $(Cost, \mathcal{S}, \sigma)$
  for each $B_i \in \sigma$ **do**
    $I \leftarrow \mathcal{S} \downarrow B_i$
    if $(I \neq \phi)$
      $stack.push() \leftarrow (Cost + 1, i, I)$
      $\mathcal{S} \leftarrow \mathcal{S} \wedge \overline{I}$
  if $(\mathcal{S} \neq \phi)$
    $stack.push() \leftarrow (Cost + 1, \infty, \mathcal{S})$

4 The GOLFER Tool and Experiments

The algorithms described above have been implemented in an model checker called GOLFER. The tool GOLFER incorporates the heuristic search algorithms A*, IDA* and weighted A*, and will construct a SPDB as part of heuristic synthesis using the abstraction/approximation techniques described above.

GOLFER is built on top of the open-source model checker NuSMV [4], and offers almost all the BDD-based verification facilities included in the system. It allows, for example, the user to choose an input ordering and transition-partition heuristics. Additionally, we have implemented a frontier-partition heuristic that is important in the guided model checking algorithm. As well as the automatic abstraction construction that uses the data dependency analysis, GOLFER allows the user to input the abstraction H as a file of variable strength values.

The work is of course on-going and the current version of GOLFER consists of about 3200 lines of C code (not including the NuSMV code).

To evaluate the ideas presented in this paper we have experimentally compared the performance of BDD-IDA* in GOLFER and the standard algorithms in NuSMV (namely BDD-BFS and a SAT-based bounded model checking algorithm). Note that in our experiment both BDD-IDA* and BDD-BFS use the same transition partition method of NuSMV for each model. We compare the run-time and memory usage for these methods. These approaches all operate in a fully automated manner, without any user interference except that we need to set up a bound for our algorithm and the SAT-based algorithm. In the experiment we set the bound to 50 and use the zChaff solver for the SAT algorithm. We first compare the performance of GOLFER and NuSMV on a simple game. We then follow with more realistic models. In the experiment we used known good BDD variable orderings for both BDD-IDA* and BDD-BFS when they were available. We also tried random orderings and found both algorithm has similar sensitivity to the same ordering. All experiments were carried out using shell scripts. The timeout operation (set to 2 hours) was implemented by a perl script. All experiments were conducted in a shared Intel X86 machine (CPU P4 933MHz) running Linux with 6G RAM.

The Sliding-Tile Puzzle. consists of a board of $n \times n$ squares occupied by $n^2 - 1$ tiles. Each tile exactly fits on one square and is labelled by a number ranging from 1 to $n^2 - 1$. Starting in some given initial configuration of the tiles on the board, the aim of the game is to move the tiles one at a time by utilising the empty square until some given goal configuration is reached. Each state of the puzzle has between 2 and 4 successors, hence the branching factor for the search graph is small. In the experiment, we use a 3×3 board and 8 tiles. We encode the puzzle in the SMV input language and randomly generate 500 solvable initial configurations. We show the results for our algorithm BDD-IDA* and the standard BDD-BFS approach in NuSMV in Figure 2. (The SAT-based approach is not included at this stage as it is not competitive on small models.)

In Figure 2 we group, average and order the data for the runs that result in the same solution depth. Generally, but not always, the shorter the solution

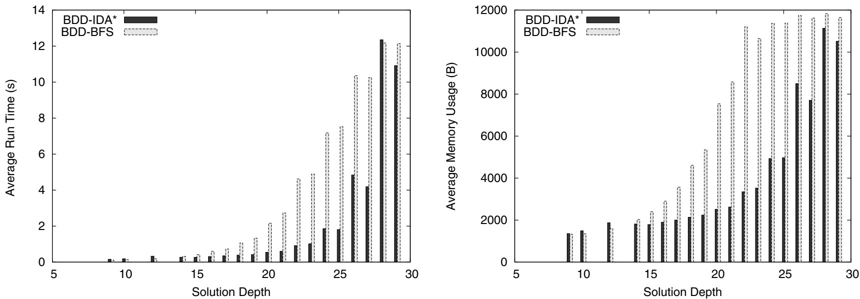

Fig. 2. Run-time and memory usage for IDA* and BFS for the sliding-tile puzzle

depth, the faster the model checker finds the solution. Most solutions for the 500 starting configurations were in the range 17 to 27. Within this solution range BDD-IDA* outperforms BDD-BFS in both time and memory. For configurations with shorter solutions, BDD-BFS is generally faster than BDD-IDA* because of the overhead of the abstraction process in BDD-IDA*, which dominates when the goal configuration is just a few transitions away. For configurations with the longest solutions, BDD-IDA* and BDD-BFS perform similarly. It is not clear why this is the case. It is true that there are few long solutions, so the sparseness of data may be contributing to this behaviour. However, we have observed the heuristic is quite poor for these configurations. If we manually generate a better heuristic for these configurations, we found that BDD-IDA* performed much better than BDD-BFS. We therefore feel that the data dependency analysis may be responsible, and conjecture that abstracting the system by eliminating supposedly weak variables loses validity in the longest runs. This may be an artifact of the particular data-dependency analysis that we have used.

While this data provides an interesting comparison between the 2 methods, one needs to remember that these methods are certainly not the best way to solve this kind of puzzle. An explicit-state model checker for example could be made to solve these puzzles faster than any of the above methods.

Real-World Examples. We applied the BDD-IDA* and BDD-BFS to the 8 models listed in Table 1. In this table, we show the type of model and the size of the SMV specification in each case. Some of the models can be found from

Table 1. Model used in the experimentation

Name	Description	Type	Lines SMV code
dme	distributed mutual exclusive ring	circuit	102
leader	concurrent leader election	protocol	129
mutex	mutual exclusion	protocol	116
ns	Needham-Schroeder public key protocol	protocol	319
peter	Peterson's mutual exclusion algorithm	protocol	126
sr	sender receiver protocol	protocol	106
tarb	tree arbiter	circuit	142
tcas	traffic collision avoiding system	controller	3269

Bwolen Yang's collection of SMV models. If a model is parameterised, the value of the parameter is indicated by a numerical suffix in the name of the model. We sometimes also used the same model with different invariants. These models contain a parenthesised 'p' suffix in the model name.

The experimental results for comparing BDD-IDA*and BDD-BFS are shown in Table 2. For each model, we show the number of Boolean variables (#Vars) that are used to encode the model, the length of the counter-examples (CX), and the run-time and the number of BDD nodes for each of the methods (when possible). The table shows that BDD-IDA* consistently outperforms BDD-BFS in all but one case, *peter-3*. Note that the run-times for BDD-IDA* include the time for heuristic synthesis. We believe that the poor performance in the case

Table 2. Experimental results for BDD-IDA*, BDD-BFS

			Run-time		# BDD Nodes	
Model	# Vars	CX	BDD-IDA*	BDD-BFS	BDD-IDA*	BDD-BFS
dme6	240	28	78.46	–	655163	–
dme8	320	30	7033.99	–	4499580	–
leader-3	85	16	1.01	1.65	305231	535585
leader-4	128	18	11.07	40.33	228193	223136
leader-5	168	20	111.62	502.18	981095	2335851
leader-6	200	22	1123.79	5486.61	3945215	4988551
mutex-16	141	10	0.66	11.13	208967	439208
mutex-20	175	10	1.01	36.13	320392	598775
mutex-24	207	10	1.56	87.59	461404	2486530
mutex-28	239	10	2.30	208.93	158186	4898940
ns (p1)	87	14	0.90	13.14	183538	117207
ns (p2)	133	14	7.43	341.80	103938	1267943
peter-3	72	26	0.59	0.42	151819	144783
peter-4	103	42	10.06	54.87	278411	542995
sr-11	273	14	0.44	67.14	177872	237768
sr-12	297	14	0.70	95.83	211911	262272
tarb-15	244	24	15.29	431.64	258489	6743882
tarb-17	276	24	32.11	712.31	464059	12649365
tarb-19	308	24	28.46	561.77	626627	11079508
tcas (p1)	292	12	4.44	25.07	190876	617102
tcas (p2)	292	16	3.43	92.11	536990	2918189
tcas (p3)	292	24	9.45	2364.88	328944	17165753
tcas (p4)	292	18	116.37	250.39	1745925	10673258
tcast (p1)	292	12	5.34	27.94	230336	623381
tcast (p2)	292	18	38.05	275.91	623307	6224724
tcast (p3)	292	16	4.37	107.33	562949	1625174
tcast (p4)	292	16	4.97	93.85	552857	1620110

Table 3. Experimental results for BDD-IDA*, BDD-BFS and SAT

	Run-time		
Model	BDD-IDA*	BDD-BFS	SAT
dme6	78.46	–	176.37
dme8	7033.99	–	521.77
ns (p2)	7.43	341.80	192.80
tarb-15	15.29	431.64	121.61
tarb-17	32.11	712.31	156.45
tarb-19	28.46	561.77	195.11

of *peter-3* is an artifact of its smallness: the run-time is short and the automatic heuristic synthesis is an overhead that BDD-BFS does not have. In the cases *mutex* and *tcas*, BDD-IDA* can be up to 2 orders of magnitude faster. In most cases, less BDD nodes are used, sometimes an order of magnitude less. In the few cases where more BDD nodes were used, it was the same order of magnitude.

We cannot see from this data how much of the improved run-times comes from the 'falsification superiority' of DFS over BFS (note the very different BDD-partitioning schemes used in both strategies clouds this issue somewhat as well), and how much is a result of the guided search. We have used the same run-time options in all cases. In a few cases, we did notice that by changing certain run-time options such as the threshold of partition size or partition heuristics, we could improve the performance for BDD-BFS. However, we could never make it perform better than BDD-IDA*. We have not tried to fine-tune the partitioning scheme used in BDD-IDA*. Placing this work in context we should note that all the models contain at most a few invariant properties, and we know these properties are false. The experimental context is hence somewhat artificial and

BDD-IDA*may not produce such performance improvements when used to verify models containing many properties. Furthermore, BDD-IDA* detects counter-examples. If the algorithm does not return before the timeout then we cannot say whether a counter-example exists or not.

We also compared the run-time of BDD-IDA*, BDD-BFS and SAT methods and the results are shown in Table 3. Note that we only do so if the SAT has better performance than BDD-BFS. Of 6 models we have experimented, BDD-IDA*detects error faster than SAT in 5 models. For model "dme8", our BDD-IDA* runs much slower than SAT for some unknown reason.

Optimality. The performance of BDD-IDA* is dependent on the quality of the heuristic. Suppose the heuristic cost for a BDD is h^* and the exact cost h, then the quality is determined by the smallness of $\mid h^* - h \mid$. We in fact don't care whether h^* over- or under-estimates the real cost, but if it does over-estimate the cost, then we cannot guarantee that the algorithm will find the shortest counter-example. In model checking this is not normally an issue, but in general, and in particular in artificial intelligence, it can be a very serious issue. In fact, the heuristic synthesis procedure used in this work always results in a heuristic that under-estimates the cost because it is based on homomorphic abstractions [17]. To improve the effectiveness of the heuristic, we could instead use the heuristic cost $a \times h^*$ instead of h^*, where $a > 1$ is a constant factor that can be tuned for specific applications. We could go a step further and use the total cost formula $f^* = b \times g + a \times h^*$ where a, b are constant values and g is the exact cost from the initial BDD (or state) to the current BDD (state). This may speed up the search dramatically, but optimality can no longer be guaranteed.

5 Related Work

The main work in BDD-based guided model checking uses prioritised traversal techniques. In [3] Cabodi et. al. proposed a mixed forward-backward prioritised traversal algorithm that checks invariant properties. This work is closely related to our approach as both share the idea of using prioritised traversal as well as abstractions and approximations. However, Cabodi et. al. differ in the way they combine these aspects:

– They construct an approximation of the concrete model and use it to approximate the forward reachable state set, which is then used to prioritise the backward traversal of the state space. They use a best-first search algorithm. We use approximation for the computationally-intensive abstraction process. The heuristic synthesis in our approach is goal-oriented and BDD-IDA* also takes the real cost into account. As well, the state space search of our approach is iterative deepening which characterises the SAT-based search strategy.
– They study only forward-backward traversal orders, which are more suited to circuits than communication protocols (for example) because traversals in both directions may not be possible even in the abstract model due to high branching factors. In our work we do not restrict ourselves in this way.

During the approximation phase, we do not have to traverse from the target to the initial states. Because our approximation is target-oriented, partial approximation traversal also serves as a heuristic for estimating the length of partial counter-examples.

In [18,12] prioritised traversal algorithms are proposed based on BDD partitioning (or subsetting). A so-called "high-density" reachability analysis is used as a BDD optimisation technique to traverse the state space of the system, and the density of a BDD is defined to be the ratio of states the BDD can represent over its size. The BDD with higher density will gain higher priority in the state space traversal. This technique only aims at optimising the size of BDDs and offers no guidance to the model checking algorithm in the search for error states. In our work, heuristics are synthesised from the abstract model, and provide direct information about potential error states in the model. The above approaches use the VIS language and we use SMV. It would be possible combine our work with the above approaches, but there would be difficulties.

Most work in guided model checking is based on the explicit-state representation [9,20,21]. The first work on guided model checking in [22] applies prioritised state space exploration to model checking and proposes practical heuristics to guide the search. Heuristic search algorithms such as A* and IDA* have also been used in explicit-state model checking in HFS-SPIN [9], recently Hopper (implemented on top of Murφ) [20] and FLAVERS [21]. All this work shows that the heuristic search algorithm can enhance the model checker's ability to detect counter-examples. The role of BDDs, particularly in combination with IDA*, is an important aspect of our work of course, as is the use of heuristic synthesis, which none of these authors above have addressed.

6 Conclusion and Future Work

In this work we have presented a fully automatic, symbolic, abstraction-guided model checker that builds its own abstract model, and uses this model as heuristic to guide the model checker. The main contribution of this work is its integration of:

- a data dependency analysis that is used to build an abstract model
- IDA* with heuristic synthesised from the abstract model, and
- a BDD partitioning algorithm in BDD-IDA* based on the heuristic.

The heuristic, which plays a vital role in guided search, is 'double-dipped' in this research: it not only guides the IDA* search strategy, it also provides a mechanism to partition the search space (i.e. BDDs). While it is true that the internal BDD operations are more complex than standard BDD-BFS, this is hidden from the user.

The 'bug-hunting' ability of DFS has long been recognised, as is attested by the huge popularity of SPIN. But SPIN is not symbolic of course, and is not guided, and being conventional DFS, is not always able to find the nearest bugs, which BFS does so well. The GOLFER tool adds a functionality to NuSMV that is all of the above, without the expense of BFS.

There is of course much work to be done. Foremost will be gaining a better understanding of the best type of data analysis to use to build the abstract model. It is not clear whether other notions of weak and strong may be more appropriate in determining whether a variable is important or not in guiding the search algorithm. For example, the current notion of eliminating weak variables does not work when all variables are equally related to the property variables. One direction for future research in this area is machine learning. Extending the framework to search for counter-examples for liveness properties is also an useful step to take.

Finally, in the abstraction-refinement framework Clarke et al. [5] uses abstraction in a very different way to us (we use it only to compute a heuristic). Nevertheless, the efficient way we detect counter-examples, which play an important role in refinement, could be usefully employed in that framework.

Acknowledgement. We would like to thank all anonymous referees for their corrections and suggestions.

References

1. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207. Springer-Verlag, 1999.
2. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction*, C-35(8):677–691, Aug 1986.
3. G. Cabodi, S. Nocco, and S. Quer. Mixing forward and backward traversals in guided-prioritized bdd-based verification. In *Proceedings of the 14th International Conference Computer Aided Verification, Copenhagen, Denmark, July 27-31*, volume 2404 of *LNCS*, pages 471–484. Springer, 2002.
4. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification, Trento, Italy, July 6-10*, volume 1633 of *LNCS*, pages 495–499. Springer, 1999.
5. E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proceedings of the 14th International Conference on Computer Aided Verification, Copenhagen, Denmark, July 27-31*, volume 2404 of *LNCS*, pages 265–279. Springer, 2002.
6. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
7. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
8. J. C. Culberson and J. Schaeffer. Searching with pattern databases. In *Proceedings of the 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, Toronto, Ontario, Canada, May 21-24*, volume 1081 of *LNCS*, pages 402–416. Springer, 1996.
9. S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF–SPIN. In *Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, Proceedings*, volume 2057 of *LNCS*, pages 57–79. Springer, 2001.

10. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. Technical report, Albert-Ludwigs-Universitt Freiburg, 2001.
11. S. Edelkamp and A. Lluch-Lafuente. Abstraction databases in theory and model checking practice. In *Proceedings of Workshop on Connecting Planning Theory with Practice, International Conference on Automated Planning and Scheduling (ICAPS), Whistler, Canada*, 2004.
12. R. Fraer, G. Kamhi, B. Ziv, M. Y. Vardi, and L. Fix. Prioritized traversal: Efficient reachability analysis for verification and falsification. In *Proceedings of 12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 389–402. Springer, 2000.
13. R. Korf. Iterative-deepening A*: An optimal admissible tree search. In *Ninth International Joint Conference on Artificial Intelligence(IJCAI-85)*, pages 1034–1036, LA,California,USA, 1985. Morgan Kaufmann.
14. R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, July 1993.
15. K. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Boston, MA, 1993.
16. K. Qian and A. Nymeyer. Abstraction-based model checking using heuristical refinement. In *Proceedings of the 2nd Internation Symposium on Automated Technology for Verification and Analysis (ATVA'04), to appear*, LNCS. Springer, 2004.
17. K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Proceedings of the 10th Internation Conference on Tools and Algorithms for the Construction and Analysis of Systems, Barcelona, Spain,*, volume 2988 of *LNCS*, pages 497–511. Springer, 2004.
18. K. Ravi and F. Somenzi. High-density reachability analysis. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, pages 154–158. IEEE Computer Society, 1995.
19. A. Santone. Heuristic search + local model checking in selective mu-calculus. *IEEE Transactions on Software Engineering*, 29(6):510–523, 2003.
20. K. Seppi, M. Jones, and P. Lamborn. Guided model checking with a bayesian meta-heuristic. In *Proceedings of the Fourth International Conference on Application of Concurrency to System Design (ACSD'04)*, pages 217–226. IEEE, 2004.
21. J. Tan, G. S. Avrunin, L. A. Clarke, S. Zilberstein, and S. Leue. Heuristic-guided counterexample search in flavers. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pages 201–210. ACM Press, 2004.
22. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Proceedings of the 35th Conference on Design Automation, Moscone Center, San Francico, California, USA, June 15-19*, pages 599–604. ACM Press, 1998.