Guided Invariant Model Checking Based on Abstraction and Symbolic Pattern Databases

Kairong Qian and Albert Nymeyer

School of Computer Science
The University of New South Wales
UNSW Sydney 2052 Australia
{kairongq, anymeyer}@cse.unsw.edu.au

Abstract Arguably the two most important techniques that are used in model checking to counter the combinatorial explosion in the number of states are abstraction and guidance. In this work we combine these techniques in a natural way by using (homomorphic) abstractions that reveal an error in the model to guide the model checker in searching for the error state in the original system. The mechanism used to achieve this is based on *pattern databases*, commonly used in artificial intelligence. A pattern database represents an abstraction and is used as a heuristic to guide the search. In essence, therefore, the same abstraction is used to reduce the size of the model and guide a search algorithm. We implement this approach in NuSMV and evaluate it using 2 well-known circuit benchmarks. The results show that this method can outperform the original model checker by several orders of magnitude, in both time and space.

1 Introduction

In the past decade, model checking has become the formal method of choice to verify reactive systems [1]. While it is an automated method, model checking inherently suffers from the well-known "state explosion" problem, where the complexity of the model is exponential in the number of components that comprise the system. Not surprisingly, most research in model checking is focused on ways to minimise the impact of state explosion. Symbolic model checking [2, 3,4] and on-the-fly, memory-efficient model checking [5,6] have had some quite dramatic successes in the field. Although these methods have been successfully used to verify some industrial strength systems, they are still unable to cope with large systems in general. In [7], Clarke et. al. have developed a method for using abstract interpretation of the system to reduce the complexity of (CTL) model checking. This is based on the observation that irrelevant information can be abstracted away from the concrete system to significantly reduce the size of the model, while preserving the properties to be verified. The drawback of this method is that it induces false negative results, namely, if this method reports a counterexample that shows the property in the abstract system is violated, then the counterexample may not be valid in the concrete system (called a *spurious*

K. Jensen and A. Podelski (Eds.): TACAS 2004, LNCS 2988, pp. 497–511, 2004.

counterexample). If a spurious counterexample is found, one has to refine the model and repeat the verification again. This is called the *abstraction-refinement* model checking paradigm [8,9].

More recently, inspired by heuristic search in artificial intelligence, many researchers have investigated the applicability of heuristic search techniques to model checking. This can be referred to as the *guided* model checking paradigm [10,11,12,13,14,15,16]. The effectiveness of this approach, however, depends strongly on the informedness of the search heuristics. Generally, only intuitive heuristics can be invented by humans. More informative heuristics can be very application-dependent and complex to derive.

In artificial intelligence, an application-independent heuristic, called *pattern databases*, has been successfully employed to solve hard search problems such as the N-puzzle and Rubik's cube [17,18]. To define a pattern database, we refer to a fixed part of a given state as a pattern. The set of all such patterns forms the domain of the database. Given any state, the pattern can be looked up in the pattern database to find the minimum path length from a state containing the pattern to a goal state. In essence, a pattern contains the subgoals that must be achieved to solve the problem [17].

For example, in a puzzle, we can consider a subset of tiles to be a pattern. A pattern database is then based on the retrograde analysis of the pattern starting at the goal state. Because we only consider the pattern during the course of pattern database construction, many original states may become indistinguishable from the pattern (i.e. if we mask out non-pattern information, two states result in the same configuration). Note that a pattern database is essentially an abstraction of a system.

In [19,20], Holte et. al. extend the notion of "pattern" to homomorphic abstraction in an attempt to automatically create application-independent heuristics. Generally, we can abstract a system using two methods: *homomorphic* abstraction and *embedded* abstraction. Homomorphic abstraction can be achieved by merging groups of states and all transitions within the group are therefore not observable in the abstract system, whereas embedded abstraction can be achieved by adding more transitions to the system. In the abstraction-refinement paradigm of model checking, to preserve the temporal properties of the system, one often uses data abstraction [7]. In essence, data abstraction is a homomorphic abstraction. We focus on safety properties in this work, all properties can be expressed using ACTL* of form $\mathbf{AG}\varphi$, where φ is a Boolean expression of atomic propositions.

Effective and intuitive heuristics are hard to find in guided model checking because of the high complexity. The lack of good heuristics hinders the ability of the guided search to reduce the size of the state space. As well, the refinement of an abstract system can be computationally expensive. In this work, we combine the abstraction and guided approach, using pattern databases that are derived from homomorphic abstractions of the system. These patterns guide the model checking algorithm towards a goal (error) state.

The rest of paper is structured as follows. We formally define homomorphic abstractions in Section 2 and symbolic pattern databases in Section 3. Our approach and experimental evaluation are described in Section 4 and 5 respectively. Finally we conclude the paper and propose some possible future research.

2 Homomorphic Abstractions

In this work we are only interested in finite-state transition systems that can be formalised in the following way.

Definition 1 (Transition systems). *A finite state transition system is a 3-tuple* $M = (S, T, S_0)$*, where*

- S *is a finite set of states*
- $T \subseteq S \times S$ *is a transition relation*
- $S_0 \subseteq S$ *is a set of initial states*

The set of states of a transition system can be described by a non-empty set of state variables $X = (x_0, x_1, ..., x_n)$, where each variable x_i ranges over a finite domain D_i. A homomorphic abstraction of a transition system is denoted by a set of surjections $H = (h_1, h_2, ..., h_n)$, where each h_i maps a finite domain D_i to another finite domain $\hat{D}_i$ with $|\hat{D}_i| \leq |D_i|$. If we apply H to all states of a transition system, denoted by $H(S)$, we will generate an abstract version of the original, concrete system.

Definition 2 (Homomorphic abstraction). *Given a transition system* $M = (S, T, S_0)$ *and a set of surjective mapping functions* $H = (h_1, h_2, ..., h_n)$*, a homomorphic abstraction of* M *is also a transition system and denoted* $\hat{M} = (\hat{S}, \hat{T}, \hat{S}_0)$*, where*

- $\hat{S} = H(S)$ *is a set of states with* $|\hat{S}| \leq |S|$
- $\hat{T} \subseteq \hat{S} \times \hat{S}$ *is a transition relation, where* $(\hat{s_1}, \hat{s_2}) \in \hat{T}$ *iff* $\hat{s_1} = H(s_1) \wedge \hat{s_2} = H(s_2) \wedge \exists s_1 \exists s_2 (s_1, s_2) \in T$
- $\hat{S}_0 = \{\hat{s} | \hat{s} \in \hat{S} \wedge \hat{s} = H(s) \wedge s \in S_0\}$

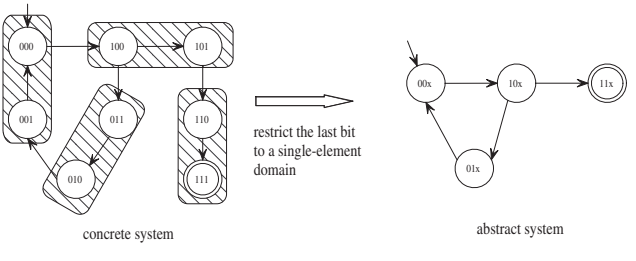

Fig. 1. A homomorphic abstraction of a transition system

Intuitively, a homomorphic abstraction is a kind of relaxation of the concrete transition system in the sense that we deliberately eliminate some information by merging groups of states. In [7], Clarke et. al. prove that a homomorphic abstraction preserves a class of temporal properties (ACTL*): in other words, the concrete and abstract system satisfy the same set of formulae. Note however that if the abstract system does not satisfy the property, we cannot conclude that the concrete system violates the property: the model needs to be further refined and model-checked again. Refinements however can be computationally expensive. Instead, we use the homomorphic abstraction to construct a pattern database to guide the search towards an error state in the concrete system.

3 Symbolic Pattern Databases

In the guided model checking paradigm, one must provide heuristics (or hints) to steer the model checking algorithm. In practice most heuristics are application dependent, so there is a strong need to discover a systematic way of relaxing the system. Pattern databases provides a systematic way of deriving heuristics. In essence, a pattern is a subgoal of the goal state. In this work we are interested in searching for a given goal state in a transition system rather than in verifying the system as a whole. We therefore need to add another element, a goal state G, to the 3-tuple M. Without loss of generality, we define G to be a single state [1]. If we have a transition system $M = (S, T, S_0, G)$, then a homomorphic abstraction will map the goal state to the subgoal state, i.e. $\hat{G} = H(G)$. By applying this mapping to the entire set of states of the system, we will have an abstract system $\hat{M} = (\hat{S}, \hat{T}, \hat{S_0}, \hat{G})$, where $\hat{T}$ and $\hat{S_0}$ are defined in the same manner as in Definition 2.

For example in Figure 1 we show a transition system and a homomorphic abstraction that restricts the right-most bit to a single-element domain. By applying this abstraction to every state of the system, we generate the abstract system shown in Figure 1 as well. A pattern database can be constructed based on this abstract system. An item in the pattern database is a 2-tuple $(\hat{s}, n)$, where $\hat{s} \in \hat{S}$ is an abstract state and n is the number of transitions required to reach $\hat{s}$ from the abstract goal. A simple example of a pattern database for the abstract system in Figure 1 is shown in Table 1.

Table 1. An example of pattern database

abstract state ($\hat{s}$)	transitions from abstract goal state (n)
11	0
10	1
00	2
01	3

[1] Multiple goals can be handled quite easily by removing the detected goal state from the goal set and running the algorithm until the goal set is empty.

A pattern database is based on the breadth-first backward traversal of the abstract system $\hat{M}$. While traversing backward in the abstract system, we label each abstract state $\hat{s}$ with the number of transitions starting at state $\hat{G}$. Note that there can be more than one path from $\hat{G}$ to a state $\hat{s}$, but we keep the shortest one. (When the breadth-first traversal encounters a state that has been labelled before, it is simply ignored.) Since we are interested in finite systems, the breadth-first traversal will eventually expand all states backward-reachable from $\hat{G}$ in $\hat{M}$. This is called the fixed point. Once we reach this fixed point, we collect all states (with labels) and put them into a table similar to Table 1. This is the pattern database of the system M with respect to the homomorphic abstraction H. Note that many states will be labelled with the same number because they are the same distance from the abstract goal state.

In symbolic model checking, sets of states can be encoded as binary decision diagrams (BDDs) [2,21]. Since our approach is also based on symbolic model checking, we represent the entire pattern database using BDDs as well. Note that symbolic pattern databases have been used in planning problems and can represent very large pattern databases, and often uses relatively less memory than explicit representations [22]. This is mainly because of the compactness nature of BDDs [21]. We represent those states in explicit pattern databases that have the same label (the same number of transitions starting at $\hat{G}$) with a single BDD, denoted b_i, where i is the label of that set of states. Because the pattern database is derived using the backward breadth-first traversal in the abstract system, i should range over $\{0, 1, 2, \ldots, N\}$ where N is the maximum depth of the traversal. In the worst case, N would be $|\hat{S}| - 1$.

Definition 3 (Symbolic Pattern Database). *Given a transition system $M = (S, T, S_0, G)$ and a homomorphic abstraction H, the symbolic pattern database of M with respect to H is a set of BDDs $P = \{b_0, b_1, \ldots, b_N\}$, where b_0 is the BDD representing the abstract goals $\hat{G} = H(G)$, and b_i is the BDD representing all states at depth i in a breadth-first backward traversal in the abstract system starting at $\hat{G}$. The depth of P, denoted $|P|$, is defined to be the maximum depth N of the backward traversal, i.e. $|P| = N$.*

In Figure 2 we show the BDDs representing the symbolic pattern database of the example in Table 1. We use a Boolean vector $X = (x_0, x_1, x_2)$ to represent a state in the concrete system, so an abstract state can be represented by $\hat{X} = (x_0, x_1)$. The symbolic pattern database contains 4 BDDs representing the characteristic Boolean functions of the corresponding abstract states: $b_0 = x_0 \wedge x_1, b_1 = x_0 \wedge \overline{x_1}, b_2 = \overline{x_0} \wedge \overline{x_1}, b_3 = \overline{x_0} \wedge x_1$.

Intuitively, because the homomorphic abstraction clusters a set of states of the concrete system as a single state of the abstract system, the path in the concrete system can be short-circuited. In the extreme case, 2 states that are not reachable from each other in the concrete system can become reachable when mapping them to the abstract system. For a transition system $M = (S, T, S_0, G)$, we define the cost of any state s to be the minimum number of transitions using backward breadth-first traversal starting at G, denoted c_s. The following lemma

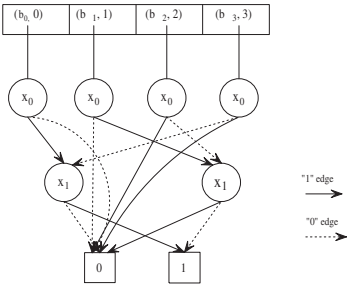

Fig. 2. An example of symbolic pattern database

shows that the cost of the state $\hat{s}$ in the abstract system is a lower bound of the cost of its corresponding concrete state s.

Lemma 1. *Given 2 transition systems* $M = (S, T, S_0, G)$ *and* $\hat{M} = (\hat{S}, \hat{T}, \hat{S_0}, \hat{G})$*, if* $\hat{M}$ *is a homomorphic abstraction of* M*, i.e.* $\hat{M} = H(M)$*, then for any state* $s \in S$ *and* $\hat{s} \in \hat{S}$*,* $c_s \geq c_{\hat{s}}$ *where* $\hat{s} = H(s)$*.*

Proof. Suppose $\pi = s_1, s_2, s_3, \ldots, s_n$ is an arbitrary path in M with $s_n = G$. If all states along the path are distinct, then $c_{s_1} = n$. If all the states $H(s_1), H(s_2), \ldots, H(s_n)$ are also distinct, then following the definition of a homomorphic abstraction, there exists a path $\hat{\pi} = H(s_1), H(s_2), \ldots, H(s_n)$. Thus, $c_{s_1} = c_{\hat{s_1}}$. If all states in $H(s_1), H(s_2), \ldots, H(s_n)$ are not distinct, say $H(s_i) = H(s_j), i < j$, then all states between them will be short-circuited in the abstract path, since a traversal from $H(s_{i-1})$ to $H(s_{j+1})$ only needs 2 transitions. Thus, $c_{s_1} > c_{\hat{s_1}}$.

4 The Guided Invariant Model Checking Algorithm

4.1 Standard Invariant Model Checking Algorithm

In computational tree logic (CTL), an invariant of a transition system can be expressed as $AG\varphi$. In symbolic model checking, we can use two methods, namely pre-image and image computation, to check the correctness of this class of properties. Given a set of states F and transition relation T, pre-image and image are computed as follows: $\mathit{pre\text{-}image}(T, F) = \{s | (r \in F) \wedge ((s, r) \in T)\}$ and $\mathit{image}(T, F) = \{s | (r \in F) \wedge ((r, s) \in T)\}$. The pre-image invariant checking is based on a greatest fixed point calculation algorithm [2,1] as characterised by $\mathbf{AG}\varphi = \nu Z.\varphi \wedge \mathbf{AX}Z$, where νZ is the greatest fixed point operator. In practice, this algorithm corresponds to a backward breadth-first search and may be inefficient because some states generated by the backward search may not be reachable from the initial state of the system (and hence need not have been computed).

Another method, image computation, is based on forward breadth-first search. This algorithm requires two inputs: the transition system (S, T, S_0) and the error state $\overline{\varphi}$, where φ is the invariant that holds in any state along all paths. For convenience, we use the transition system with a goal element $G = \overline{\varphi}$ as an input, where $G \notin S_0$. The algorithm is shown in Figure 3. In each iteration, a set of new reachable states R_{new} and the difference between R_{old} and R_{new}, F (frontier), are computed. To check whether the invariant has been violated, F is intersected with the error state in each iteration as well. If the intersection is

```
Procedure InvarCheck (S, T, S_0, G)
1    R_old ← False
2    R_new ← S_0
3    while(R_old ≠ R_new)
4        F ← R_new ∧ ¬R_old
5        R_old ← R_new
6        F ← Image(T, F)
7        if (F ∧ G ≠ False)
8            return ErrorFound
9        R_new ← R_old ∨ F
10   return NoErrorFound
```

Fig. 3. Symbolic invariant checking algorithm

not empty, we terminate the algorithm and report the error. If the intersection remains empty and the set of reachable states does not change (a fixed point), then we can claim that invariant φ is never violated in this model. Note that we test the existence of an error state in each iteration (on-the-fly), whereas some model checkers compute the entire set of reachable states and then intersect it with error states. For large transition systems, computing all reachable states may not be feasible as the BDD representing the reachable states is too large. In this work, we use an on-the-fly technique to test for error states.

4.2 An Example

Before formally introducing our guided model checking algorithm, we illustrate the technique on the transition system that we saw in Figure 1. We use a Boolean vector $X = (x_0, x_1, x_2)$ to represent a state, and the invariant we are interested in is that all three Boolean variables cannot be true simultaneously in any state. This property can be expressed in CTL as $\mathbf{AG}(\overline{\alpha})$, where $\alpha = \overline{x_0} \vee \overline{x_1} \vee \overline{x_2}$. Hence, our search goal (error state) is the complement of the property α. To construct the pattern database, we define a homomorphic abstraction H that abstract the third Boolean variable, x_2, to a single-element domain. So the abstract system can be constructed as shown in Figure 1. We then apply standard model checking (the **InvarCheck** algorithm shown in Figure 3) to the abstract system. In this example, **InvarCheck** will report an error because there is a path leading from $\hat{S}_0 = (0, 0)$ to $\hat{G} = (1, 1)$ in the abstract system. At this point, instead of

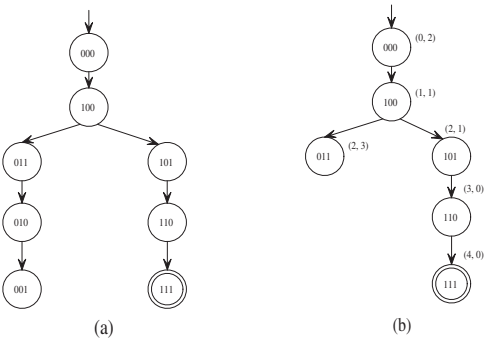

Fig. 4. The search trees generated by the (a) **InvarCheck**, and (b) **GuidedInvarCheck** algorithms for the example in Section 4.2

refining the abstract system and checking it again, we use the abstract system to construct a symbolic pattern database as shown in Table 1 and Figure 2.

The essence of the guided algorithm is that each state (set of states) is associated with an estimated distance to the goal as well as the actual distance from the initial state. We use the symbolic pattern database constructed from the homomorphic abstraction to assign to each state an estimated value (Figure 1 and Table 1). We map a state in the concrete system to an abstract state and look up its estimated value in the database. For example, for the state $(0, 1, 1)$, the corresponding abstract state is $(0, 1, x)$ and the estimated value is 3 (see Table 1). In a symbolic pattern database, each item is a BDD representing a set of abstract states, so the database look-up can be accomplished by calculating the conjunction of two BDDs. For example, to look for the estimated distance for state $(1, 0, 0)$, we iteratively compute $(x_0 \wedge \overline{x_1} \wedge \overline{x_2}) \wedge b_i$ for i from N to 0. If the resulting BDD is not constant $False$, we assign i to $(1, 0, 0)$ as the estimated distance. In this particular case, $(x_0 \wedge \overline{x_1} \wedge \overline{x_2}) \wedge b_1$ is not constant $False$, so we assign the estimated value 1 to that state. Thus, the symbolic pattern database will partition a set of states according to their estimated value. Our invariant model checking algorithm is therefore guided by the symbolic pattern database in its search for an error in the concrete system. In Figure 4, we show the difference in the search tree with and without heuristic guiding. In this figure we label the states in the guided algorithm by a pair consisting of the number of actual and estimated transitions (resp.).

4.3 Guided Invariant Model Checking Algorithm

The guided algorithm is shown in Figure 5. In contrast to the standard algorithm **InvarCheck**, the guided algorithm takes a homomorphic abstraction function H as input in addition to the concrete transition system.

```
Procedure GuidedInvarCheck ((S, T, S_0, G), H)
1    if (InvarCheck((H(S), H(T), H(S_0), H(G)) = NoErrorFound)
2        return NoErrorFound
3    P ← construct(H(S), H(T), H(S_0), H(G))
4    SearchQueue ← (0, 0, S_0)
5    Closed ← False
6    while (SearchQueue ≠ φ)
7        (g, h, F) ← SearchQueue.pop()
8        if (F ∧ G ≠ False)
9            return ErrorFound
10       Closed ← Closed ∨ F
11       F ← Image(T, F) ∧ ¬Closed
12       QueueImage(P, F, g + 1)
13   return NoErrorFound

Procedure QueueImage (P, Img, Cost)
1    n ← |P|
2    while (n > 0)
3        I ← b_n ∧ Img
4        if (I ≠ φ)
5            SearchQueue ← (Cost, n, I)
6            Img ← Img ∧ ¬I
7        if (Img = φ) return
8        n ← n − 1
9    SearchQueue ← (Cost, ∞, I)
```

Fig. 5. The guided invariant checking algorithm that uses a pattern database.

In line 1, an abstract model is constructed using the abstraction function H and standard **InvarCheck** is called to prove the invariant. If this succeeds, then the invariant is true in the concrete system (as discussed in Section 2). If this fails, the algorithm then constructs a symbolic pattern database (line 3) according to the abstract function H (as discussed in Section 3). *SearchQueue* in line 4 is a priority queue used to determine which state(s) should be explored first. The element of the queue is a 3-tuple, (g, h, S) where g is the actual number of transitions to S from the initial state, h is estimated number of transitions to a goal (error) state and S is a BDD representing a set of states. When determining which element should be dequeued for further exploration, *SearchQueue* considers $f = g + h$ as the priority key and pops the element with minimum f. In lines 5-13, the heuristic search algorithm A* [23,24] is adapted to symbolically explore the state space in the concrete model.

The difference between the guided algorithm and **InvarCheck** is that whenever the image of the frontier, F, is computed, we employ the symbolic pattern database to partition this image and assign each sub-image with a heuristic evaluation before we push it back to the search queue. This is shown in procedure

QueueImage in Figure 5. Given a set of states, Img, this procedure iterates through every item b_i in P, and checks whether there exists a subset I of Img, such that $H(G)$ can be reached from $H(I)$ in the abstract system (line 3 of procedure **QueueImage**). Note that if the Img cannot be partitioned by P, we simply push it back to the search queue with heuristic evaluation ∞ (line 9).

We prove that the guided algorithm **GuidedInvarCheck** is both correct and optimal.

Theorem 1 (Correctness). *Given a transition system (S, T, S_0, G) and a homomorphic abstraction H, and let R_g and R_i be the indication returned from* **GuidedInvarCheck**$((S, T, S_0, G), H)$ *and* **InvarCheck**(S, T, S_0, G) *respectively. Then $R_g \Leftrightarrow R_i$.*

Proof. As **InvarCheck** and **GuidedInvarCheck** both use state space exploration, if **GuidedInvarCheck** detects an error, so will **InvarCheck**, and vice versa. If there is no error in the system, **InvarCheck** will explore all reachable states (the fixed point) and report $NoErrorFound$. In this case, we have to prove **GuidedInvarCheck** also explores all reachable states. This is detected by the *Closed* set that stores all states explored by **GuidedInvarCheck**. When all reachable states have been explored, $Image(T, F) \wedge \overline{Closed}$ (line 11) will be an empty set, so nothing is pushed into the search queue. Hence, the search queue will eventually become empty and $NoErrorFound$ will be returned. Thus, in all cases the two algorithms will return the same result, i.e. $R_g \Leftrightarrow R_i$.

An important outcome of the model checking technique is that it can provide the counterexamples (or witnesses in the case of existential properties), showing why the property is violated. A counterexample, or error trace, is a path starting at the initial state of the concrete system and leading to an error state. Given a transition system $M = (S, T, S_0, G)$, we define an error trace as follows.

Definition 4. *An error trace is a finite path, $\pi = s_0, s_1, \ldots, s_n$, in M, where $s_0 \in S_0$, $s_n = G$ and $(s_i, s_{i+1}) \in T$ $(i \in [0, n-1])$. The length of this path, denoted $L(\pi)$, is the number of states along the path, i.e. $L(\pi) = n + 1$.*

Generally, the shorter the length of the error trace, the easier it is for human beings to interpret it. Because **InvarCheck** corresponds to a breadth-first forward search, it will determine the minimum-length error trace. The following theorem ensures that **GuidedInvarCheck** detects the minimum error trace as well.

Theorem 2 (Optimality). *Let $M = (S, T, S_0, G)$ be a transition system and H be a homomorphic abstraction. Let π_i and π_g be the error traces detected by* **InvarCheck** *and* **GuidedInvarCheck** *respectively. Then $L(\pi_g) = L(\pi_i)$.*

Proof. The proof of the theorem can be established by proving that **GuidedInvarCheck** detects the shortest path from the initial state s_0 to a goal state $s_g = G$. Note that the state space exploration algorithm in **GuidedInvarCheck** is adapted from the heuristic search algorithm A*. If the lower bound heuristic is used, the algorithm guarantees the path is shortest (minimum cost) [23,24].

So we need to prove the symbolic pattern database heuristic is a lower bound. According to lemma 1, for any path $\pi = s_0, s_1, \ldots, s_n$ in the concrete system and its corresponding path $\hat{\pi} = \hat{s_0}, \hat{s_1}, \ldots, \hat{s_n}$ in the abstract system, $c_{s_0} \geq c_{\hat{s_0}}$. Thus the symbolic pattern database heuristic is a lower bound and $L(\pi_g) = L(\pi_i)$.

We could of course use more than one symbolic pattern database to guide the algorithm. Let $M = (S, T, S_0, G)$ be a transition system and $H_0, H_1, \ldots, H_{k-1}$ be homomorphic abstractions. We hence can construct k symbolic pattern databases. When partitioning the image in the procedure **QueueImage**, we search all symbolic pattern databases to find the largest heuristic estimated value for the sub-image. This will make the heuristic estimation more accurate and guide the search more efficiently, but it also increases the computation complexity. Because all H_i are homomorphic abstractions, every symbolic pattern database is still a lower bound heuristic. Using multiple symbolic pattern database will therefore preserve the optimality of the algorithm. Note that if multiple pattern databases are used, the error trace will have the same length as the trace detected by using a single pattern database. Using multiple pattern databases instead of a single pattern database would involve a straightforward extension to the **GuidedInvarCheck** algorithm.

5 Experimental Evaluation

To determine the effectiveness of guiding, we have implemented our algorithm in the model checker, NuSMV [4]. For the purpose of comparison, we modify the so-called ad hoc algorithm in NuSMV [2] to test for the existence of an error, as shown in Figure 3. Experiments are carried out in a machine running Linux with an Intel 933Hz CPU and 512MB RAM.

In this work, we did not use any input variable ordering. The abstraction method we use is to make some Boolean variables invisible. Note that because our approach does not involve any refinements, we require that the abstraction not to be too coarse. In general, the criteria to select an abstraction granularity is that it should be feasible to construct the symbolic pattern database in a small amount of time with maximum depth. The relation between the granularity ("abstractness") of the abstraction and the accuracy of the resulting heuristic has been studied by Prieditis and Davis in 1995 [25]. In this work, we set the threshold for the construction to be 60 seconds. If the construction cannot finish within 60 seconds, we have to abandon the abstraction and choose another more abstract system.

The two benchmark circuits we use in this paper were published in David L. Dill's thesis [26]. Since we focus on error detection, we use two "buggy" designs for our evaluation. The first circuit family is called a tree arbiter circuit, which is used to enforce mutual exclusion among users accessing shared resources.

[2] In NuSMV, safety properties can be checked by using either standard fixed point evaluation algorithm, or an "ad hoc" algorithm which computes the set of reachable states and intersects it with the set of error states.

The basic element of this circuit family is the arbiter cell, which implements the mutual exclusion between two users. For more than two users, cells can be connected to form a tree-like structure. The bottom level cells are connected to users and the top cell to the shared resource. The request signal from a user propagates upwards from the bottom level of the tree, and if the shared resource is granted to the user, the acknowledgement propagates downwards only to the user who requested it. The second circuit family, distributed mutual exclusion ring (DME), is used to implement mutual exclusion as well. Instead of forming a tree-like structures, DME cells are connect as a ring. Mutual exclusion is implemented by passing a token in the cell ring. In this work, we construct interleaving finite models for these circuits and check the invariant that no two users receive an acknowledgement simultaneously. The results of our experiment is shown in Table 2.

Table 2. Experimental Results for the tree arbiter and DME

			InvarCheck		**GuidedInvarCheck**	
Circuits	Depth	BDD vars	Total nodes	CPU time (s)	Total nodes	CPU time (s)
tree-arb 7	20	50	322,452	1.340	47,778	2.357
tree-arb 9	20	64	483,286	2.300	634,783	3.120
tree-arb 11	21	78	1,249,438	6.360	560,439	2.440
tree-arb 13	21	92	5,593,886	13.590	450,156	2.520
tree-arb 15	24	106	161,297,839	4759.000	4,262,998	19.290
tree-arb 17	24	120	—	$>$ 6 hours	7,323,386	35.260
tree-arb 19	24	134	—	$>$ 6 hours	7,922,396	34.930
dme 06	26	114	—	$>$ 6 hours	3,316,858	18.240
dme 08	30	152	—	$>$ 6 hours	93,794,232	1137.000

Note that all error traces detected by our method have exactly the same length as detected by standard **InvarCheck** in NuSMV. For each circuit, we experimented with a few invariants and only report those errors with depth more than 20, because short error traces can be easily detected by both algorithms regardless of model size. We also report the number of BDD variables used to encode each model to reflect the size of the system. The memory use of the two algorithms is reflected in the total nodes allocated by the BDD engine that is contained in NuSMV. The hyphens in the table are for those experiments that did not terminate within (a randomly chosen) 6 hours.

For the tree arbiter circuits, **GuidedInvarCheck** can easily handle up to 19 cells, whereas **InvarCheck** cannot handle more than 15 cells in less than 6 hours. Note that there is a time-line cross-over in the table between 9 and 11 cells. For smaller circuits, **InvarCheck** can detect errors faster than **GuidedInvarCheck**. For larger circuits, the performance of **InvarCheck** deteriorates rapidly, while the time taken by **GuidedInvarCheck** remains quite constant. This occurs because in systems with a low level of concurrency, BDD-based

breadth-first search is more efficient than guided search. As described in Section 4.3, **GuidedInvarCheck** also needs to partition the image (BDD slicing) and this introduces some overhead as well. For systems with a high level of concurrency, BDD-based breadth-first search is dominated by huge BDD computations, whereas guided search partitions large BDDs into smaller ones, and only explores promising states, thereby avoiding the exponential growth in the size of the BDDs.

We only experimented with 2 of the circuits in the DME family as **InvarCheck** could not cope with circuits that had more than 6 cells in the available time. **GuidedInvarCheck** could however handle up to 8 cells with an error depth of 30. For larger circuits, we need to resort to manipulating the variable ordering to improve the performance.

The experimental results indicate that the guided approach can outperform standard model checking by several orders of magnitude, in both time and required memory. As expected, **GuidedInvarCheck** not only detects the errors much quicker than **InvarCheck**, but also found the shortest error traces.

6 Conclusions and Future Work

In this paper, we have presented a symbolic model checking algorithm that combines homomorphic abstraction and guided search techniques. We introduce a mechanism called symbolic pattern databases to provide a heuristic to guide the model checker. The pattern databases represent the relaxed system and associate each state of the system with a heuristic value (i.e., the estimated number of transitions to an error state). This is required by the underlying heuristic search algorithm to partition the states and guide the search.

The guided search is of course only used when an error is detected in the abstract system. In essence we double-dip on the abstraction: the abstraction reduces the size of the state space directly, but also indirectly as a result of the guided search. There is no need for further abstraction refinements in this work, although embedding guided search in an iterative abstraction-refinement approach would make interesting further work.

It is important to note that guided model checking algorithms like ours are designed for debugging, and not verification. For systems that have no errors, the guided approach does not have any conceptual advantage over conventional model checking algorithms. However, the guided method does slice large BDDs, which reduces the sizes of the BDDs substantially. Although pattern databases are memory-based heuristics, the use of BDDs to represent them also helps to counter any potential size problem, and make a seamless integration with symbolic model checking possible. In this work, we only use BDDs to store symbolic pattern database (the abstracted state space). An interesting next step would be to determine whether other alternative data structures, such as algebraic decision diagrams (ADDs) and multi terminal BDDs (MTBDDs), have better performances than BDDs.

While the guided algorithm is fully automated, we still require human interaction to determine the level of abstraction. In this work, we only implement one method of constructing an abstraction. The improvement that we have found using guided model checking is of course expected: what sets this work apart is the way we have derived the heuristic. Using our approach, the quality of the heuristic will be dependent on the quality of the user-constructed abstraction rather than the application domain. Our next step is to investigate how the way that the abstraction is constructed affect efficiency of the guidance algorithm in practice.

References

1. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
2. McMillan, K.: Symbolic model checking. Kluwer Academic Publishers, Boston, MA (1993)
3. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. In: Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, Washington, D.C., IEEE Computer Society Press (1990) 1–33
4. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: 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 Lecture Notes in Computer Science., Springer (1999) 495–499
5. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods in System Design **1** (1992) 275–288
6. Holzmann, G.J.: The model checker SPIN. Software Engineering **23** (1997) 279–295
7. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems **16** (1994) 1512–1542
8. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings the 12th International Conference on Computer Aided Verification, Chicago, IL, July 15-19. Volume 1855 of Lecture Notes in Computer Science., Springer (2000) 154–169
9. Clarke, E., Gupta, A., Kukula, J., Strichman, O.: 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 Lecture Notes in Computer Science., Springer (2002) 265–279
10. Yang, C.H., Dill, D.L.: 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, ACM Press (1998) 599–604
11. Alur, R., Wang, B.Y.: "Next" heuristic for on-the-fly model checking. In: Proceedings of the 10th International Conference on Concurrency Theory,, Eindhoven, The Netherlands, August 24-27. Volume 1664 of Lecture Notes in Computer Science., Springer (1999) 98–113
12. Bloem, R., Ravi, K., Somenzi, F.: Symbolic guided search for CTL model checking. In: Proceedings of the 37th Conference on Design Automation, Los Angeles, CA, June 5-9, ACM (2000) 29–34

13. Edelkamp, S., Lafuente, A.L., Leue, S.: Directed explicit model checking with HSF–SPIN. In: Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, Proceedings. Volume 2057 of Lecture Notes in Computer Science., Springer (2001) 57–79
14. Reffel, F., Edelkamp, S.: Error detection with directed symbolic model checking. In: FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, Proceedings, Volume I. Volume 1708 of Lecture Notes in Computer Science., Springer (1999) 195–211
15. Cabodi, G., Nocco, S., Quer, S.: 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 Lecture Notes in Computer Science., Springer (2002) 471–484
16. Santone, A.: Heuristic search + local model checking in selective mu-calculus. IEEE Transactions on Software Engineering **29** (2003) 510–523
17. Culberson, J.C., Schaeffer, J.: 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 Lecture Notes in Computer Science., Springer (1996) 402–416
18. Korf, R.: Finding optimal solutions to to Rubik's cube using pattern databases. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, July 27-31, Providence, Rhode Island, AAAI Press / The MIT Press (1997) 700–705
19. Holte, R.C., Mkadmi, T., Zimmer, R.M., MacDonald, A.J.: Speeding up problem solving by abstraction: A graph oriented approach. Artificial Intelligence **85** (1996) 321–361
20. Holte, R., Hernadvolgyi, I.: Experiments with automatically created memory-based heuristics. In: Proceedings of the 4th International Symposium on Abstraction, Reformulation, and Approximation, Horseshoe Bay, Texas, USA, July 26-29. Volume 1864 of Lecture Notes in Computer Science., Springer (2000) 281–290
21. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **C-35** (1986) 677–691
22. Edelkamp, S.: Symbolic pattern databases in heuristic search planning. In: Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, April 23-27, Toulouse, France, AAAI (2002) 274–283
23. Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, USA (1984)
24. Nymeyer, A., Qian, K.: Heuristic search algorithm based on symbolic data structures. In: Proceedings of the 16th Australian Joint Conference in Artificial Intelligence, Perth, Australia, 3-5 December. Volume 2903 of Lecture Notes in Artificial Intelligence., Springer (2003) 966–979
25. Prieditis, A., Davis, R.: Quantitatively relating abstractness to the accuracy of admissible heuristics. Artificial Intelligence **74** (1995) 165–175
26. Dill, D.L.: Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. The MIT Press, MA (1988)