Abstraction Databases

Stefan Edelkamp
Fachbereich Informatik
Universität Dortmund
stefan.edelkamp@cs.uni-dortmund.de

Alberto Lluch-Lafuente
Dipartimento di Informatica
Università di Pisa
lafuente@di.unipi.it

February 9, 2004

Abstract

Abstraction is one of the most important issues to cope with large and infinite state spaces in model checking and to reduce the verification efforts. The abstract system is often smaller than the original one and if the abstract system satisfies a correctness specification, so does the concrete one. However, abstractions may introduce a behavior violating the specification that is not present in the original system.

This paper bypasses this problem by proposing the combination of abstraction with heuristic search to improve error detection. The abstract system is explored in order to create a database that stores the exact distances from abstract states to the set of abstract error states. To check, whether or not the abstract behavior is present in the original system, efficient exploration algorithms exploit the database as a guidance.

1 Introduction

The ultimate goal of model checking [4] is to prove the correctness of a system model with respect to a given property. The correctness proof is often given through a complete exploration of the underlying state space. If the system does not satisfy the property, model checking algorithms return a witness of the error in form of a counterexample, which allows to locate and repair the faulty behavior. As a matter of fact, much of the success of model checking is due to its ability to find and report errors.

The main drawback of model checking is the *state explosion problem*. State spaces of software designs are large enough in practice to make an exhaustive exploration infeasible. Moreover, many systems have an infinite state space. For instance, a simple program with only one integer variable may easily span the entire integer range. In this case, abstraction schemes are one of the only chances to retain tractability.

It is not always easy to find abstractions that are correct with respect to the correctness specification and, at the same time, provide significant reductions. Exact approximations are abstractions that induce an abstract system, which is observationally equivalent to the original one; where the notion of observational equivalence depends, of course, on the logic used by specifications. They are difficult to be obtained and might result in weak reductions. Subsequently, research has mainly focused on over-approximations that induce *simulations* [18], since they are found easier and provide more drastic reductions. Roughly speaking, system $\hat{M}$ simulates a system M if every behavior of M is also present in $\hat{M}$. If the abstraction applied induces a simulation, then the abstract system simulates the original one. As a consequence, if it satisfies the specification so does the original one. The opposite direction, however, is not true. A bad behavior in the abstract system might not be present in the original one. In this case we call the bad behavior a *spurious error*. Frequently, the approximation is refined for a new abstraction that is consistent with the counterexample established, and the verification process starts again.

We introduce *abstraction databases* as a general paradigm to enhance the efficiency of existing model checkers. Our approach analyzes the abstract model in order to create a database which stores the shortest distance from a given abstract state to the set of abstract error states. This database is used as a heuristic during the exploration of the original system. As a result proving that an abstract error is present in the original system can be done more efficiently. The resulting heuristics have a property, namely *monotonicity*, which guarantees optimal counterexamples if A* [11] is used as traversal policy. The framework of model simulation will fit perfectly into the setting of estimate construction through databases.

Another view of the paper's contribution is that while abstractions turned out to be a universal tool to verify a temporal property, they were only of limited help for falsifying it. With abstraction databases we utilize abstractions for improved falsification.

Our approach has a further advantage. It fits to both explicit and symbolic model checking approaches. Abstraction databases are constructed by exploring the abstract state space. As a consequence, this exploration can be simulated in a form of symbolic reachability analysis. The inferred symbolic representation of the set of reachable states together with the according distances to the goal, then serves as a estimate that can be referenced either in explicit and symbolic heuristic search.

The paper is structured as follows. First we review *directed model checking* approaches and introduce *pattern databases* as applied in AI search. Then we turn to the concept of abstraction in model checking, including data and predicate abstraction. Afterwards we give a theoretical foundation on *abstraction databases* based on Kripke structures. This will exploit similarities between the concepts developed in AI and model checking. We use α-SPIN and an extension of the directed model checker HSF-SPIN to implement the approach. In the experiments we prove validity on the small set of examples that is provided together with the α-SPIN distribution. Finally, we draw conclusions, discuss related work, and indicate future research avenues.

2 Directed Model Checking for Improved Error Detection

Heuristic search [20] tackles state exploration problems by using an evaluation function, which represents the likelihood of a state to lead to a goal state. This estimator for the cost of the remaining solution path is called the *heuristic*.

One of main heuristic search representatives is A*, which explores states ordered with respect to the combined merit of generating path (the current optimal path to a state found by the algorithm so far) length and the estimate. If heuristic h is *admissible*, by means a lower bound on the optimal distance to the goal, then A* returns optimal solutions. If, in addition, h is *monotone*, that is for every two states u and v, where v is a successor of u, we have $h(v)-h(u)+1 \geq 0$, then the algorithm explores a number of states linear in the size of the state space. It has been shown that monotonicity implies admissibility [20]. The quality of the heuristics mainly influences the size of the state space that is explored by the algorithm until the goal is found: the better the estimate, the smaller the exploration effort.

Directed model checking incorporates heuristic search algorithms like A* to enhance the bug-finding capability of model checkers, by accelerating the search for errors and finding (near to) minimal counterexamples. In that manner we can mitigate the state explosion problem and the long counterexamples provided by some algorithms like depth-first search, which is often applied in explicit model checking.

One can distinguish about 4 main classes of evaluation functions based on the information they try to exploit. *Property specific* heuristics [9] analyze the error description as the negation of the correctness specification. In some cases the underlying methods are only applicable to special kinds of errors, e.g. on a deadlock with a heuristic that tries to block individual processes. In other cases the approaches are applicable to a wider range of errors with heuristics that analyze the error formula with respect to the current state, or estimates based on comparing state vector representations. We will see that projecting a state to its local processes, as done in the *FSM distance heuristic*, is in fact a special case of the abstraction database approach.

The second class has been denoted as being *structural* [10], in the sense that source code metrics govern the search. This class includes coverage metrics (such as *branch count*) as well as concurrency measures (such as *thread preference* and *thread interleaving*). Next there is the class of *user heuristics* that inherit guidance from the code designer in form of source annotations, yielding preference and pruning rules for the model checker. The last class are *planning heuristics* [8], where the problem of finding an error in the system is reduced to a planning task and solved using heuristic search action planners.

3 Pattern Databases in AI Search

Pattern database search [6] is an automatic technique for the improved design of admissible heuristics. A pattern database is a mere (hash) table, where indices are patterns and entries contain heuristic values. Patterns itself are simplified states according to a state relaxation function.

The pattern database technique was first applied to define effective heuristics for sliding-tile puzzles. For this case, any problem relaxation corresponds to removing a selected set of tiles from the board. The remaining set of tiles is referred to as the *pattern*. The *pattern database* stores all *pattern states* together with their shortest path distance on the simplified board to the pattern state for the goal. It is constructed in a BFS starting with the goal pattern and using inverse relaxed state transitions. For the search in the original puzzle, the pattern is selected in the active state with the stored distance value as estimator function.

Subsequently, pattern databases defined heuristics to find optimal solutions to random instances of the Rubik's Cube for the first time [15]. In this domain, some cube labels are removed from the overall cube. The *pattern* corresponds to the remaining set of colored faces. Three different relaxations were applied: one which leaves all the corner cubes colored, and two relaxations that selected 6 edge cubes each as the pattern.

As the shortest path distance in *pattern space* is smaller than or equal to the shortest path distance in the original one, we see that pattern databases deliver admissible heuristics; one for each pattern.

In addition, pattern databases for different patterns can be combined in order to get a more accurate heuristic. One option to combine more than one pattern database heuristic is to take the maximum of the heuristic values for that state provided by the different databases. The result is still an admissible heuristic. In solving random instances to Rubik's Cube this was the option of choice. For the sliding-tile puzzles, however, where only one tile can move at a time, *disjoint pattern databases* [16] have been devised by counting moves of the pattern tiles only. The problem is that different pattern databases may count operators twice, since an operator can have a non-trivial image under more than one relaxation. If we assert a count of 1 to each operator only for one relaxation, then the sum of the retrieved pattern database values taken as a combined estimate preserves admissibility, while being more accurate.

4 From Patterns to Abstractions

The idea of a pattern can be generalized as follows. If a state s is represented as a state vector $(s^1, \ldots, s^k)$ with variables in some finite domains, then patterns are established by a projection, reducing the domains of some s^i, $i \in \{1, \ldots, k\}$. In the original interpretation vector positions were replaced by *don't cares*.

In model checking this corresponds to a form of *data abstraction*, which exploits the fact that specifications for software models usually consider fairly simple relationships among the data values in the system. In such cases, one can map the domain of the actual data values into a smaller domain of abstract data values. Such mapping induces a mapping of the states of the system, which in turn induces an abstract system. If variables have domain D and all abstract variables have domain $A \subseteq D$, then an abstraction function ϕ maps values from D into A.

In many cases the abstract system simulates the original one [4]. Data abstraction is not the only option we have. With *predicate abstraction* the concrete states of a system are mapped to abstract states according to their evaluation under a finite set of predicates [21]. Automatic predicate abstraction approaches have been designed and implemented for finite and infinite state systems. The predicates $p_1, \ldots, p_n$ define an abstraction function $\phi : S \rightarrow \{0,1\}^n$ with $\phi(s) = \hat{s}$ if for all i we have $\hat{s}_i = p_i(s)$, that is abstract states refer to truth vectors of atomic propositions. An abstract state is reachable if it is an abstraction of a concrete state that is reachable. Invariants to be checked are supplied as additional predicates. As in the above case, predicate abstractions turn often out to yield simulations.

The common use of abstraction in practice consists of a simple life cycle. Once an abstraction simulation has been defined, the model checker is used to verify the correctness property on the abstract system. If the result is positive, then we know, that the original system satisfies the property. A negative answer, however, does not ensure that the original system violates the property too. There are various ways to proceed.

Usually, a negative answer is accompanied with a counterexample, which is just a finite path that violates the property. An abstract path might not have a corresponding path in the concrete system. In this case, the path is called *spurious*. Hence, an abstract counterexample has to be analyzed in order to check whether it is spurious or not. One possible approach is to use the symbolic algorithm described in [3], which basically consists of a forward search. Initially, it starts with the set of states given by $\phi^{-1}(\hat{s}_0)$. Then it computes the set of successors of such states and performs the intersection with the set of states given by $\phi^{-1}(\hat{s}_1)$. The algorithm goes on until it establishes an empty set or $\phi^{-1}(\hat{s}_n)$.

The procedure is simpler if there is a clear correspondence between the abstract and the concrete system, where each code line in the abstract system corresponds to a line in the concrete system. As a consequence, each abstract transition has a corresponding concrete transition and one can just try to simulate the abstract counterexample in the original system [19]. If this is not possible, then the counterexample is spurious. Otherwise, the abstract counterexample is said to be feasible and one has to check whether it really violates the property.

Another approach is to analyze the *chose-free* state space of a system [19]. The idea is to bound non-deterministic branches of the state space, such that counterexamples are always *deterministic paths*. This approach uses a result, which states that every abstract non-deterministic path has a corresponding concrete path [22]. If one of these methods certificates the existence of a spurious counterexample, the abstraction must be refined accordingly, and the previously described *abstract-check-refinement* cycle starts again.

In some cases, however, the strategy of validating abstract counterexamples is not suitable. We claim that an abstract counterexample might be the symptom of an error, even if it has no corresponding concrete counterexample. Consider the following trivial example for abstraction of a simple program:

```
x:=0; while x<n do x++; end do
```

As an abstraction on variable x we take $\phi(0) = \texttt{ZERO}$, $\phi(n) = N$, and $\phi(i) = \texttt{MIDDLE}$ for $i \in \{1, \ldots, n-1\}$, thus inducing the following abstract program:

```
x:=ZERO
while x<N do
 if(x=0) then x=MIDDLE
 else x=nondeterministically MIDDLE or N
end do
```

The original system has a finite path $(s_0, \ldots s_n)$, where s_i represents the state in which $x = i$. We use an abstraction ϕ, such that $\phi(s_i) = \phi(s_{i+1})$ for $1 \leq i \leq (n-2)$. It can be shown that ϕ is a simulation that induces an abstract system with infinitely many paths $(\phi(s_0), \phi(s_1)^j, \phi(s_n))$, where $j \geq 1$.

Clearly, only the path given by $j = n - 2$ has a corresponding concrete path. However, a model checker looking for an abstract state in which $x = n$, would find path $(\phi(s_0), \phi(s_1), \phi(s_n))$, which is a spurious counterexample. Classical approaches would try to refine the abstraction, while we propose to consider the path as a symptom for the existence of a similar path in the original system.

5 Abstraction Databases in Theory

For a formal treatment of abstraction databases we assume that the model is given in form of a Kripke structure. A *Kripke structure* $M = (S, L, AP, \rightarrow)$ consists of a set of states S, a set of atomic propositions AP, a labelling function $L : S \rightarrow 2^{AP}$ and a transition relation $\rightarrow \subseteq S \times S$. A set of initial states is often associated together with M. In the following we abbreviate $(s, t) \in \rightarrow$ with $s \rightarrow t$.

Given two Kripke structures $M = (S, L, AP, \rightarrow)$ and $\hat{M} = (\hat{S}, \hat{L}, \hat{AP}, \hat{\rightarrow})$ with $\hat{AP} \subseteq AP$, relation $\sim \subseteq S \times \hat{S}$ is a *simulation relation* between M and $\hat{M}$ whenever for all $s \sim \hat{s}$ we have $L(s) \cap \hat{AP} = \hat{L}(\hat{s})$ and for every state s_1 with $s \rightarrow s_1$ there is a state $\hat{s_1}$ and $\hat{s} \hat{\rightarrow} \hat{s}_1$ and $s_1 \sim \hat{s}_1$. We also say that $\hat{M}$ simulates M denoted by $M \preceq \hat{M}$.

It has been show that if $M \preceq \hat{M}$, then for every universal path quantified CTL* formula f with atomic propositions that are contained in $\hat{AP}$, $\hat{M} \models f$ implies $M \models f$ [4]. In addition, for every path $(s_0, s_1, \ldots)$ there is a corresponding path $(\hat{s}_0, \hat{s_1} \ldots)$ with $s_i \sim \hat{s_1}$, $i \geq 0$. So every behavior of M is also a behavior of $\hat{M}$.

An *abstraction* ϕ of a Kripke structure $M = (S, L, AP, \rightarrow)$ is a mapping from states to abstract states that induces an abstract Kripke structure $\hat{M} = (\hat{S}, \hat{L}, \hat{AP}, \hat{\rightarrow})$ as follows: $\hat{S} = \phi(S) = \{\phi(s) \mid s \in S\}$, and if $s \rightarrow s_1$ then $\phi(s) \hat{\rightarrow} \phi(s_1)$, i.e. $\hat{\rightarrow} = \{(\phi(s), \phi(s_1)) \mid s \rightarrow s_1\}$. Generally speaking, we apply a surjection of the state space graph $(S, \rightarrow)$ into $(\hat{S}, \hat{\rightarrow})$. Abstraction ϕ is a homomorphism, since for all transitions $s \rightarrow s_1$ we have $\phi(s) \hat{\rightarrow} \phi(s_1)$. This implies that if state t is reachable from state s then $\phi(t)$ is reachable from $\phi(s)$.

Moreover, abstraction ϕ can be extended to modify the label set, that is $\hat{L} = \phi(L)$ but we impose that $\hat{AP} \subseteq AP$, and if $L(s) \in AP$ then $\hat{L}(\phi(s)) = L(s) \cap \hat{AP}$.

We see that the definition of an abstraction is close to the definition of a simulation relation between M and $\hat{M}$ with $\hat{s} = \phi(s)$ and $\hat{s_1} = \phi(s_1)$, with the difference that in case of a simulation only the existence of a successor state $\hat{s_1}$ of s with $\hat{s}_1 \sim s_1$ is required, where here we fix $\hat{s_1}$ to be $\phi(s_1)$. We will see that for exploration we will have to relax the requirement on s_1.

So far we have that an abstraction ϕ as defined above yields a simulation relation between M and $\hat{M}$. Now we are not only interested in the existence of paths but also in shortest paths between states in the original and in abstract space.

Theorem 1 *Let M be a Kripke structure, ϕ be an abstraction, and $\hat{M}$ be the abstract structure under ϕ. For each two concrete states s and t we have that the shortest path from the abstract state $\phi(s)$ to $\phi(t)$ is not longer than the shortest path from s to t.*

Proof: Let $p = (s = v_0, \ldots v_n = t)$ be the shortest path form s to t in M. By homomorphism $\phi(p) = (\phi(v_0), \ldots \phi(v_n))$ connects $\phi(s)$ to $\phi(t)$. The optimal path from $\phi(s)$ to $\phi(t)$ can only be shorter or equal.

There are two options to define the *abstract error*. Suppose that we have a set of error states T violating invariant the temporal f. Safety model checking can be reduced to invariant checking. For this case $\neg f$ is true in t if and only if t is an error. The abstract error state set is $\phi(T) = \{\phi(t) \mid t \in T\}$. In case of an extended abstraction we have $L(\phi(t)) = L(t) \cap \hat{AP}$. So if $L(t)$ satisfies f, $L(\phi(t))$ satisfies the restriction of f to $\hat{AP}$, designating the abstract error.

Theorem 1 suggests to use the shortest path in the abstract state space to the abstract error as an estimator in M. To avoid involved computations in the search phase, a pre-computed look-up table is created. In other words, an *abstraction database* according to an abstraction ϕ is a table with shortest path distances from each abstract state $\hat{s}$ to the abstract error set. While Theorem 1 proves the admissibility of the heuristic estimate we can also prove monotonicity.

Theorem 2 *Let M be a Kripke structure, ϕ be an abstraction, and $\hat{M}$ be the abstract structure under ϕ. The shortest path distance to the abstract error is monotone.*

Proof: For each transition $s \rightarrow s_1$, homomorphism yields $\phi(s) \hat{\rightarrow} \phi(s_1)$. By the triangular inequality of shortest paths we have that 1 plus the shortest path from $\phi(s_1)$ to the abstract error is larger than or equal the shortest path from $\phi(s)$ to the abstract error.

Theorem 2 is important, since in this case the combined merit in A* is monotone, so that A* variants can be implemented without any reopening strategy, which is more difficult to code and can lead to exponential running time with respect to the model size. The reasoning is as follows. Let g be the depth of node s with successor s_1 in the search tree and h be its estimate. Then, by monotonicity we have $g(s_1) + h(s_1) = g(s) + 1 + h(s_1) \geq g(s) + h(s)$.

Until now, we have assumed that the state space graph for M is given explicitly. In practice, we have to explore it *on-the-fly* by successive enumeration. Therefore, defining $\hat{\rightarrow}$ by abstract state pairs is not sufficient, so that we also need to define abstract transitions to span abstract space. This brings us back to the simulation relation. If we define a mapping ϕ on the successor generation function, abbreviated as $\phi(\rightarrow)$, we require that each successor in abstract space has at least one successor in original space as a preimage. For Kripke structures $M = (S, L, AP, \rightarrow)$ this means that if $\phi(s)\ \phi(\rightarrow)\ \hat{s}_1$, then there exists s_1 with $s \rightarrow s_1$ and $\phi(s_1) = \hat{s}_1$.

Now the definition chosen for an abstraction matches the one given for a simulation relation. Therefore, the two aspects coincide. For abstractions that are simulations, Theorems 1 and 2 remain valid, so that the shortest path distance heuristic for abstract space is a monotone estimator function.

Consequently, we can use any abstraction that is a simulation relation to build an *abstraction database*. In database construction, the main limitation is the number of (abstract) states that can be hold in memory. Since databases are pre-computed, it is important to approximate an upper bound on the storage requirements each abstraction imposes. While there are approximations for the size of the abstract state spaces in the AI search domains, for general model checking we have not yet sufficient answers to establish a tight bound.

Disjoint abstraction databases are derived from disjoint pattern databases as follows. Two abstract databases according to abstraction functions ϕ and ψ are *disjoint*, if for all transitions t' in the abstract model for ϕ and transitions t'' in the abstract model for ψ we have $\phi^{-1}(t') \cap \psi^{-1}(t'') = \emptyset$.

In other words, each operator introduces cost 1 in at most one abstraction, and cost 0 in all others. It $\phi^{-1}(t') \cap \psi^{-1}(t'') \neq \emptyset$ then the sum of the cost of t' and t'' in the respective abstract models is less than or equal to 1. Since in this case each operator is counted at most once, for each state we have that the sum of the shortest path distances from $\phi(s)$ and $\psi(s)$ to their respective abstract error state set is still a lower bound to the shortest path from s to the error in the original space.

As an example, take the FSM distance estimate that applies to asynchronous systems. It is defined as the sum of the shortest path distances in the local state transition graph for each process. We see that state transitions are partitioned into disjoints set for each process component and only one component performs a state transition at a time.

What is the running time for database construction? Backward breadth-first search starting from the set of abstract error states suffices to determine the state-to-goal distances in abstract space. This corresponds to linear time and space with respect to the size of the abstract model.

As one implication, the running time for the construction of the FSM distance shrinks from $O(|P|^3)$ according to the all-pair shortest path algorithm of Floyd-Warshall to $O(|P| + | \rightarrow_P |)$ for BFS in the inverse graph, where $|P|$ is the state space size of local process P and $| \rightarrow_P |$ is the number of transitions that apply in P.

6 Implementing Abstraction Databases

The empirical work will take the SPIN model checker as a practical case study. Since we are interested in directed search with abstraction databases we extended our experimental tool *HSF-SPIN* [9], where the tool α-SPIN [17] serves as a welcome extension to SPIN [14] for defining abstractions and enumerating the abstract model.

SPIN produces source files that encode the state description and state transition function in native C code. These are linked together with the validator to allow exploration of the model. The graphical user interface XSPIN allows to code the model, to run the validator with different parameters, to show the internal automata representation, and to simulate traces with message sequence charts.

α-SPIN extends the SPIN validator by allowing the user to abstract the Promela model with suitable abstraction functions. The functions can be archived in a library for later reuse. The tool especially features *data abstraction* to the finite domain variables in the systems. The information flow is as follows. The model is scanned, processed with an XML parser, and simplified by the user selecting appropriate variables and abstraction functions, also coming in XML format. The result is another XML representation of the abstract model which, in turn, is translated back into Promela. Consequently, the approach produces no source conflict with the SPIN model checker itself. This allows us to select any model checker with same input specification. α-SPIN has been integrated in XSPIN, so that variables of the abstraction and according can be selected with the mouse. Besides other features, α-SPIN allows the integration of LTL property abstractions, over- and under-approximations, and the maintenance of different libraries of abstraction functions. To enhance readability of the abstract Promela model that is generated, much of the abstraction mapping is provided in form of pre-compiler directives that are an essential part of the Promela specification.

The experimental HSF-SPIN model checker has been designed to allow different search algorithms to apply by providing a general state expanding subroutine, that generates a list of all successors of a system state. In its current implementation it provides *blind search algorithms* like depth-first search (DFS) and breadth-first search (BFS), *heuristic search algorithms* like best-first search, A* and IDA*, and *local search algorithms* like hill-climbing and genetic algorithms. The set of evaluation function ranges from *property specific* heuristics via *user heuristics* to *structural* ones. Moreover, exploration refinements like state compression, bit-state hashing, partial order and symmetry reduction have been successfully integrated to this algorithm portfolio. HSF-SPIN can handle a significant fraction of Promela and works with the same trace format as SPIN.

To use general abstraction functions we combined α-SPIN and HSF-SPIN as follows. First, HSF-SPIN has been extended to generate abstraction databases. With each state we attached a list of all predecessors on which a state is encountered. In fact this reflects the inverse of the underlying state space graph. In case of an error, the traversal is not terminated but the abstract error states are collected in a (priority) queue.

Next, backward traversal is invoked on the implicitly represented inverse of the (possibly weighted) state space graph, starting with the queued set of abstract error states. Since we need estimates for each abstract states, we chose Dijkstra's single source all target shortest path algorithm. The established shortest path distances to the abstract error state are associated with each state in the hash table, yielding our *abstraction database*.

The most involved technical question was how to maintain the distance information that was established in a complete run in the abstract model as a look-up table in the overall search. The mere transfer of the abstraction database information from the abstract to the original space appeared not to be difficult: the entire hash table with the encoded state information is flushed to and read from disk in raw binary format. As *abstraction databases* reside on disk, we can easily access their sizes. Moreover, external databases can be reused for different runs.

To address the data in the abstract hash table, as required for the heuristic or local search algorithm in the original state space was the core problem. By using the XML abstraction function from the α-SPIN tool set the state abstraction seemed not available for the internal state description of SPIN. The key to the solution of this problem was to include dummy assignments of the form `variable := variable` into the original Promela source. The abstraction mechanism in α-SPIN results in a Promela abstractions that has changed this assignment to `variable := map(variable)`.

When compiling the model with the validator SPIN or HSF-SPIN this assignment is available in the file `pan.m` produced by either SPIN or HSF-SPIN in native C. This source fragment it is used to devise the abstraction function for each state. To address the abstraction mapping when computing the heuristic estimate of a state, we execute a state transition with a *do-transit* operation. Consequently, the source fragment is included as the abstraction function for computating the estimate in the original model.

7 Experiments

In the experiments we choose two problems that come with the α-SPIN tool. By the novelty of the product the library of available abstractions and corresponding models is small. We took the model of the air condition system with n users in the room, everyone trying to change the temperature. We selected $n \in \{2, 3\}$.

The two models contained no bug, so that we had to seed one. We choose a simple assertion to be violated by the system in large depths. Two abstraction database heuristics h_p and h_{p^*} were chosen. The first one (h_p) corresponds to an abstraction function that we devised by data abstraction for a selected variable. To illustrate the maximal reduction of our method and to indicate the range of different estimator functions that can be established, we also conducted experiments with abstraction databases for the original model, which was possible, since the state spaces were small enough.

We compared the results of the two pattern database heuristics with BFS ($h \equiv 0$) and with previously published heuristics like the formula-based heuristic (h_f) and the FSM distance (h_m) that takes a given error trail as an additional input. Tables 1 and 2 summarize the outcome of our small set of experiments. As expected, more space is needed to memorize the databases.

However, the abstraction database heuristics significantly reduced the number of states that are explored during search. Compared to the other heuristics the savings of h_p in the number of states are larger by a factor of about 5. For $n = 2$ an abstraction database was built, containing 15,858 states found while exploring the inverse graph structure. The initial state was re-encountered in depth 18 of 28 in total. For $n = 3$ the final databases had 28,782 entries, generated in 32 layers with the initial state re-established in depth 19. On our 248 MHz SUN Ultra the time to create the databases accumulated to about 10s for $n = 2$, and to about 6 min for $n = 3$.

We observe a time slow-down in the exploration with respect to BFS and A* search and the other heuristic estimates. Time consumption for exploration increased from about 7s to about 9s ($n = 2$) and from about 1 min to about 1:30 min ($n = 3$). The profiler showed that almost all of the extra time is due to file scanning.

That we have not established a larger time gain during search is due to the additional

heuristic	stored states	time spend	space consumed
h_p	21,987	2.1s + 6.9s	8,449 KB
h_{p*}	381	13.0s + 0.01s	13 MB
$h \equiv 0$	26,087	6.6s	5,953 KB
h_m	25,195	6.6s	7,129 KB
h_f	25,936	7.1s	5.937 KB

Table 1: Using A* with abstraction database and other heuristics in the air condition model with $n = 2$. Abstraction database search time is divided into file reading time and pure search time.

heuristic	stored states	time spend	space consumed
h_p	149,818	22s + 1:07 min	35 MB
h_{p^*}	532	1:45 min + 2.2s	94 MB
$h \equiv 0$	183,475	1:01 min	24 MB
h_m	176,382	1:02 min	24 MB
h_f	182,961	1:07 min	24 MB

Table 2: Using A* with abstraction database and other heuristics in the air condition model with $n = 3$. Abstraction database search time is divided into file reading time and pure search time.

time for the search in the (abstract) hash table. Twice as many calls to the hash table and state equality checks were needed. For $n = 2$ and BFS, we observed 124,463 table look-ups, while the abstraction database heuristic yields 222,925 calls. For $n = 3$ we found a discrepancy of about 1.2 million to about 2.03 million calls.

For perfect databases (h_{p^*}) the exploration results were by far better. The exploration turned out to be optimal in the number of state expansions. Search was no longer harmed by additional hash table operations. With 1ms ($n = 2$) and 2.2s ($n = 3$) the pure search time was smaller than all other approaches by orders of magnitude.

For $n = 3$ the construction of the perfect database with 805,760 entries took about 1.5 hours. For $n = 2$ the database with 113,072 entries was build in less than a minute.

8 Conclusions

This work combines abstraction reduction with heuristic search in order to improve the bug-finding capabilities of model checking. Automatically created heuristics in form of pre-computed heuristic databases are inferred in a backward traversal of the abstract state space. The entries are retrieved as estimate values for solution finding.

The general scheme allows to refine heuristics for explicit and symbolic state model checkers, where in this paper we concentrated on the explicit state model checking. The abstraction process that comes along with every heuristic estimation can be automated and is suited to general model checking. The approach is not limited to data abstraction, any simulation with an abstract state space that fits into main memory can be used to enhance error detection. The new aspect is not only to take the existence of behaviors in abstract space but also distance information into account.

We successfully implemented the approach in *HSF-SPIN* exploiting the α-Spin tool and presented first experimental results on the few models that are available. Since we have started off with this research by utilizing two considerably small models, the results are indeed preliminary. While the savings in the number of expansions are significant, total time and space consumption are not.

The increase in time is mostly due to reading the databases from a file, but also to additional retrieval time in the hash table. Compared to the determination of more involved estimates to be computed on-line for each state, however, hashing is fast in theory and practice. The increase in space is inherent to the approach, but the case studies in challenging AI search domains show that investing memory to improve the heuristic is better than using ordinary hashing of states for duplicate detection. This is especially true in *iterative-deepening* explorations that are usually considered together with pattern database search and into which we have not yet looked at.

Databases can be reused when having a different initial situation in the same model and yield admissible heuristics and allow to find counterexamples of optimal cost. Since the heuristics are monotone, A* has to consider each state at most once. The only other one heuristic that is admissible - the FSM distance estimate - can be interpreted as a special case of an abstraction databases. In this context we have not yet exploited the option of disjoint abstraction databases to improve the estimate.

By the smaller number of node expansions it is possible to analyze models that have not been analyzed before. Storing and retrieving the abstraction state space data from disk is not mandatory. Despite other heuristics, many abstraction functions can be scaled, so that in the limit we have a perfect heuristic with optimal performance.

Compared to earlier work on pattern databases that use inverse operators from the set of goal states we contributed a forward construction algorithms. During forward chaining in abstract space we construct the inverse graph structure by setting explicit links. The inverse graph is then traversed starting with the collected set of abstract error states. Moreover, by the compression of atomic state regions we are able to analyze and abstract weighted graph structures. Last but not least, we used ordinary instead of perfect hashing, so that the same hash function can be in original and abstract space.

The relation of pattern databases to state space and domain abstractions for classical AI problems has been identified in [13]. Moreover, the successful application of symbolic pattern databases in AI planning [7] makes the potential for devising symbolic abstraction databases abundant clear: by the larger number of represented states in abstract space, larger and better informed estimates are to be expected.

There is some related work on the combination of abstraction and guided search in model checking. It includes the FLAVERS tool with a heuristic was chosen that reduced infeasible execution sequences [5]. In the *Java Path Finder* directed search was incorporated with a heuristic that prefers the search in the abstract space and that does not contain any nondeterministic choices due to abstraction [10]. If an error is found in this portion of the state space, then it is an error also in the original one.

Some current software model checkers, like the Berkeley Lazy Abstraction Software Verification Tool (BLAST) [12], use abstractions on-the-fly in an *abstract-check-*

refine loop. Lazy abstraction continuously builds and refines a single abstract model on demand, driven by the model checker, so that different parts of the model may exhibit different degrees of precision. The model checking algorithms are applied to checking safety properties in C programs, while providing sufficient conditions for termination.

BLAST, among others may be viewed as heuristics that use abstract spurious counterexamples to guide the refinement process, i.e., ultimately the analysis of the concrete system. However *abstraction databases* are different, since the analysis is not iterative in nature. Instead of refining the abstraction for the error trace of a spurious error, in our approach we use abstraction to accelerate the overall search process.

The tool *C2BP* [1] applies automatic predicate abstraction to the validation of C programs. Given a C program and a set $p_1, \ldots, p_n$ of boolean C-expressions over the variables of P, predicate abstractions conservatively preserves the control flow and produces a boolean program containing only n boolean variables $b_1, \ldots, b_n$, where each b_i represents the truth predicate b_i. The result is then checked with the model checker BeBop [2], an apparent future aspirant to apply the proposed technique.

In summary, none of the related approaches in model checking applies databases of state distances in abstract space.

References

[1] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation*, pages 203–213, 2001.

[2] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Model Checking Software (SPIN)*, pages 113–130, 2000.

[3] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification (CAV)*, pages 154–169, 2000.

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

[5] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *International Conference on Software Engineering (ICSE)*, pages 37–46, 2001.

[6] J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

[7] S. Edelkamp. Symbolic pattern databases in heuristic search planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 195–204, 2002.

[8] S. Edelkamp. Promela planning. In *Model Checking Software (SPIN)*, pages 197–212, 2003.

[9] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *Model Checking Software (SPIN)*, pages 57–79, 2001.

[10] A. Groce and W. Visser. Heuristic model checking for java programs. *Software Tools for Technology Transfer*, 2004. To appear.

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

[12] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Model Checking Software (SPIN)*, pages 235–239, 2003.

[13] R. C. Holte and I. T. Hernadvölgyi. A space-time tradeoff for memory-based heuristics. In *National Conference on Artificial Intelligence (AAAI)*, pages 704–709, 1999.

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

[15] R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI)*, pages 700–705, 1997.

[16] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1-2):9–22, 2002.

[17] P. Merino, M. del Mar Gallardo, J. Martinez, and E. Pimentel. aSpin: Extending spin with abstraction. In *Model Checking Software (SPIN)*, pages 254–258, 2002.

[18] R. Milner. An algebraic definition of simulation between programs. In *Joint Conference of Artificial Intelligence (IJCAI)*, pages 481–489, 1995.

[19] C. Pasareanu, M. B.Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 284–298, 2001.

[20] J. Pearl. *Heuristics*. Addison-Wesley, 1985.

[21] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, pages 72–83, 1997.

[22] H. Saidi. Model checking guided abstraction and analysis. In *Static Analysis Symposium (SAS)*, pages 377–396, 2000.