Abstractions and Pattern Databases: The Quest for Succinctness and Accuracy

Sebastian Kupferschmid and Martin Wehrle

University of Freiburg
Department of Computer Science
Freiburg, Germany
{kupfersc, mwehrle}@informatik.uni-freiburg.de

Abstract. Directed model checking is a well-established technique for detecting error states in concurrent systems efficiently. As error traces are important for debugging purposes, it is preferable to find as short error traces as possible. A wide spread method to find provably *shortest* error traces is to apply the A* search algorithm with distance heuristics that never overestimate the real error distance. An important class of such distance estimators is the class of *pattern database* heuristics, which are built on abstractions of the system under consideration. In this paper, we propose a systematic approach for the construction of pattern database heuristics. We formally define a concept to measure the accuracy of abstractions. Based on this technique, we address the challenge of finding abstractions that are succinct on the one hand, and accurate to produce informed pattern databases on the other hand. We evaluate our approach on large and complex industrial problems. The experiments show that the resulting distance heuristic impressively advances the state of the art.

1 Introduction

When model checking safety properties of large systems, the ultimate goal is to prove the system correct with respect to a given property. However, for practically relevant systems, this is often not possible because of the state explosion problem. Complementary to proving a system correct, model checking is often used to detect reachable *error states*. To be able to debug a system effectively, it is important to have *short* or preferably *shortest possible* error traces in such cases. Directed model checking (DMC) is a well established technique to find reachable error states in concurrent systems and has recently found much attention [4, 6, 7, 10, 14, 15, 19–23]. The main idea of DMC is to focus on those parts of the state space that show promise to contain reachable error states. Thus, error states can often be found by only exploring a small fraction of the entire reachable state space. DMC achieves this by applying a *distance heuristic* that estimates for each state encountered during the search the distance to a nearest error state. These heuristics are usually based on abstractions and computed fully automatically. One important discipline in DMC is to find *optimal*, i. e., *shortest* possible error traces. This can be achieved by applying an *admissible* distance heuristic, i. e., a heuristic that never overestimates the real error distance, with the A* search algorithm [8, 9].

An important class of admissible distance heuristics is based on *pattern databases* (PDB). They have originally been introduced in the area of Artificial Intelligence [2, 5]. A *pattern* in this context is a subset of the variables or the predicates that describe the original system $\mathcal{M}$. A PDB is essentially the state space of an abstraction of $\mathcal{M}$ based on the selected pattern. The heuristic estimate for a state encountered during the model checking process is the error distance of the corresponding abstract state. The most crucial part in the design of a pattern database heuristic is the choice of the pattern, which determines the heuristic's behavior and therefore the overall quality of the resulting heuristic. Ultimately, one seeks for patterns that are as small as possible to be able to handle large systems on the one hand. On the other hand, the corresponding abstract system that is determined by the pattern should be as similar to the original system as possible to appropriately reflect the original system behavior. This is equivalent to maximizing the distance estimates of the pattern database, as higher distance values lead to more informed admissible heuristics. More precisely, for admissible heuristics h_1 and h_2, it is known that A* with h_1 performs better than with h_2 if h_1 delivers greater distance values than h_2 [18]. Therefore, it is desirable to have admissible heuristics that deliver as high distance values as possible.

In this paper, we present *downward pattern refinement*, a systematic approach to the pattern selection problem. Complementary to other approaches, we successively abstract the original system as long as only little spurious behavior is introduced. For this purpose we developed a suitable notion to measure the similarity of systems which corresponds to the accuracy of abstractions. This often results in small patterns that still lead to very informed pattern database heuristics. We demonstrate that downward pattern refinement is a powerful approach, and we are able to handle large problems that could not be solved optimally before. In particular, we show that the resulting pattern database heuristic recognizes many dead end states. Correctly identifying dead end states is useful to reduce the search effort significantly, since such states can be excluded from the search process without losing completeness. This even allows us to efficiently verify *correct* systems with directed model checking techniques.

The remainder of this paper is structured as follows. Section 2 provides notations and the necessary background in directed model checking. In Sec. 3 we detail the main part of our contribution. This is followed by a discussion of related work. Afterwards, in Sec. 5 we empirically evaluate our approach by comparing it to other state-of-the-art PDB heuristics. Section 6 concludes the paper.

2 Preliminaries

In this section we introduce the notations used throughout this paper. This is followed by a short introduction to directed model checking and pattern database heuristics.

2.1 Notation

The approach we are presenting here is applicable to a broad class of transition systems, including systems featuring parallelism and interleaving, shared variables or binary synchronization. For the sake of presentation, we decided to define systems rather gener-

ally. The systems considered here consist of parallel processes using a global synchronization mechanism. Throughout this paper, Σ denotes a finite set of synchronization labels.

Definition 1 (Process). *A process $p = \langle L, L^*, T\rangle$ is a labeled directed graph, where $L \neq \emptyset$ is a finite set of locations, $L^* \subseteq L$ is a set of error locations, and $T \subseteq L \times \Sigma \times L$ is a set of local transitions.*

For a local transition $(l, a, l') \in T$ we also write $l \xrightarrow{a} l'$. The systems we are dealing with are running in lockstep. This means that a process can only perform a local transition, if all other processes of the system simultaneously perform a local transition with the same label. In this paper, a system $\mathcal{M}$ is just the parallel composition of a finite number of processes $p_1, \ldots, p_n$, the components of $\mathcal{M}$.

Definition 2 (Parallel composition). *Let $p_1, \ldots, p_n$ be a finite number of processes with $p_i = \langle L_i, L_i^*, T_i\rangle$ for $i \in \{1, \ldots, n\}$. The parallel composition $p_1 \parallel \ldots \parallel p_n$ of these processes is the process $\langle S, S^*, \Delta\rangle$, where $S = L_1 \times \cdots \times L_n$, $S^* = L_1^* \times \cdots \times L_n^*$, and $\Delta \subseteq S \times \Sigma \times S$ is a transition relation. There is a transition $(l_1, \ldots, l_n) \xrightarrow{a} (l_1', \ldots, l_n') \in \Delta$, if and only if $l_i \xrightarrow{a} l_i' \in T_i$ for each $i \in \{1, \ldots, n\}$.*

Note that parallel composition as defined above is an associative and commutative operation. This gives rise to the following definition of systems.

Definition 3 (System). *A system $\mathcal{M} = \{p_1, \ldots, p_n\}$ is a set of processes, the components of $\mathcal{M}$. The semantics of $\mathcal{M}$ is given as the composite process $\langle S, S^*, \Delta\rangle = p_1 \parallel \ldots \parallel p_n$. We use $S(\mathcal{M})$ and $\Delta(\mathcal{M})$ to denote the set of system states and global transitions, respectively. We denote the set of error states with $S^*(\mathcal{M})$.*

To distinguish between local states of an atomic process and the global state of the overall system, we use the term *location* for the former and *state* for the latter. The problem we address in this paper is the detection of reachable error states $s \in S^*(\mathcal{M})$ for a given system $\mathcal{M}$. Formally, we define model checking tasks as follows.

Definition 4 (Model checking task). *A model checking task is a tuple $\langle \mathcal{M}, s_0\rangle$, where $\mathcal{M}$ is a system and $s_0 \in S(\mathcal{M})$ is the initial state. The objective is to find a sequence $\pi = t_1, \ldots, t_n$, of transitions, so that $t_i = s_{i-1} \xrightarrow{a_i} s_i \in \Delta(\mathcal{M})$ for $i \in \{1, \ldots, n\}$ and $s_n \in S^*(\mathcal{M})$.*

We call a sequence π of successively applicable transitions leading from $s \in S(\mathcal{M})$ to $s' \in S(\mathcal{M})$ a trace. If $s' \in S^*(\mathcal{M})$, then π is an *error trace*. The length of a trace, denoted with $\|\pi\|$, is the number of its transitions, e. g., the length of π from the last definition is n.

We conclude this section with a short remark on the close correspondence between solving model checking tasks and the nonemptiness problem for intersections of regular automata. From this perspective, a system component corresponds to a regular automaton and the error locations correspond to accepting states of the automaton. Parallel composition corresponds to language intersection. This view is not necessarily useful for efficiently solving model checking tasks, but it shows that deciding existence of error traces is PSPACE-complete [11].

2.2 Directed Model Checking

Directed model checking (DMC) is the application of heuristic search [18] to model checking. DMC is an explicit search strategy which is especially tailored to the fast detection of reachable error states. This is achieved by focusing the search on those parts of the state space that show promise to contain error states. More precisely, DMC applies a *distance heuristic* to influence the order in which states are explored. The most successful distance heuristics are fully automatically generated based on a declarative description of the given model checking task. A distance heuristic for a system $\mathcal{M}$ is a function $h : S(\mathcal{M}) \to \mathbb{N}_0 \cup \{\infty\}$ which maps each state $s \in S(\mathcal{M})$ to an integer, estimating $d(s)$, the distance from s to a nearest error state in $S^*(\mathcal{M})$. When we want to stress that $h(s)$ is a heuristic estimate for $s \in S(\mathcal{M})$, we write $h(s, \mathcal{M})$. Typically, heuristics in DMC are based on abstractions, i. e., the heuristic estimate for a states s is the length of a corresponding error trace in an abstraction of $\mathcal{M}$. During search, such a heuristic is used to determine which state to explore next. There are many different ways how to prioritize states, e. g., the wide-spread methods A^* [8, 9] and *greedy search* [18]. In the former, states are explored by increasing value of $c(s) + h(s)$, where $c(s)$ is the length of the trace on that state s was reached. If h is *admissible*, i. e., if it never overestimates the real error distance, then A* is guaranteed to return a shortest possible error trace. In greedy search, states are explored by increasing value of $h(s)$. This gives no guarantee on the length of the detected error trace, but tends to explore fewer states in practice. Figure 1 shows a basic directed model checking algorithm.

```
1  function dmc(M, h):
2      open = empty priority queue
3      closed = ∅
4      open.insert(s_0, priority(s_0))
5      while open ≠ ∅ do:
6          s = open.getMinimum()
7          if s ∈ S*(M) then:
8              return False
9          closed = closed ∪ {s}
10         for each transition t = s -a-> s' ∈ Δ(M) do:
11             if s' ∉ closed then:
12                 open.insert(s', priority(s'))
13     return True
```

Fig. 1. A basic directed model checking algorithm

The algorithm takes a system $\mathcal{M}$ and a heuristic h as input. It returns *False* if there is a reachable error state, otherwise it returns *True*. The state s_0 is the initial state of $\mathcal{M}$. The algorithm maintains a priority queue *open* which contains visited but not yet explored states. When *open.getMinimum* is called, *open* returns a minimum element, i. e., a state with minimal priority value. States that have been explored are stored in *closed*. Every encountered state is first checked if it is an error state. If this is not the case, its successors are computed. Every successor that has not been visited before is

inserted into *open* according to its priority value. The *priority* function depends on the applied version of directed model checking, i. e., if applied with A* or greedy search (cf. [8, 18]). As already mentioned, for A*, *priority*(s) returns $h(s) + c(s)$, where $c(s)$ is the length of the path on which s was reached for the first time. For greedy search, it simply evaluates to $h(s)$. When every successor has been computed and prioritized, the process continues with the next state from *open* with lowest priority value.

2.3 Pattern Database Heuristics

Pattern database (PDB) heuristics [2] are a family of abstraction-based heuristics. Originally, they were proposed for solving single-agent problems. Today they are one of the most successful approaches for creating admissible heuristics.

For a given system $\mathcal{M} = \{p_1, \ldots, p_n\}$ a PDB heuristic is defined by a *pattern* $\mathcal{P} \subseteq \mathcal{M}$, i. e., a subset of the components of $\mathcal{M}$. The pattern $\mathcal{P}$ can be interpreted as an abstraction of $\mathcal{M}$. To stress that $\mathcal{P}$ is an abstraction of $\mathcal{M}$, we will denote this system with $\mathcal{M}|_{\mathcal{P}}$. It is not difficult to see that this kind of abstraction is a projection, and the abstract system is an overapproximation of $\mathcal{M}$. A PDB is built prior to solving the actual model checking task. For this, the entire reachable state space of the abstract system is enumerated. Every reachable abstract state is then stored together with its abstract error distance in some kind of lookup table, the so-called *pattern database*. Typically, these distances are computed by a breadth-first search. Note that abstract systems have to be chosen so that they are much smaller than their original counterparts and hence are much easier to solve.

When solving the original model checking task with such a PDB, distances of concrete states are estimated as follows. A concrete state s is mapped to its corresponding abstract state $s|_{\mathcal{P}}$; the heuristic estimate for s is then looked up in the PDB. This is given formally in the definition of PDB heuristics.

Definition 5 (Pattern database heuristic). *Let $\mathcal{M}$ be a system and $\mathcal{P}$ be a pattern for $\mathcal{M}$. The heuristic value for a state $s \in S(\mathcal{M})$ induced by $\mathcal{P}$ is defined as follows:*

$$h^{\mathcal{P}}(s) = \min\{\|\pi\| \mid \pi \textit{ is error trace of model checking task } \langle \mathcal{M}|_{\mathcal{P}}, s|_{\mathcal{P}} \rangle\},$$

where $s|_{\mathcal{P}}$ is the projection of s onto $\mathcal{P}$.

The main problem with PDB heuristics is the *pattern selection problem*. The informedness of a PDB heuristic crucially depends on the selected pattern. For instance, if the selected pattern $\mathcal{P}$ contains all processes, i. e., $\mathcal{M} = \mathcal{M}|_{\mathcal{P}}$, then we obtain a *perfect heuristic*, i. e., a heuristic that returns the real error distance for all states of the system. On the other hand, since we have to enumerate the reachable abstract state space exhaustively, which coincides with the concrete one, this exactly performs like blind breadth-first search. On the other end of an extreme spectrum, the PDB heuristic induced by the empty pattern can be computed very efficiently, but on the negative side, the resulting heuristic constantly evaluates to zero and thus also behaves like uninformed search. Good patterns are somewhere in between and how to find good patterns is the topic of this paper.

3 Pattern Selection based on Downward Refinement

In this section, we describe our approach for the pattern selection that underlies the pattern database heuristic. On the one hand, as the abstract state space has to be searched exhaustively to build the pattern database, the ultimate goal is to find patterns that lead to *small* abstractions to be able to handle large systems. On the other hand, the patterns should yield abstractions that are as *similar* to the original system as possible to retain as much of the original system behavior as possible. An obvious question in this context is the question about similarity: what does it mean for a system to be "similar" to an abstract system? In Sec. 3.1 and Sec. 3.2, we derive precise, but computationally hard properties of similarity of abstract systems. Furthermore, based on these considerations, we provide ways to efficiently approximate these notions in practice. Based on these techniques, we finally describe an algorithm for the pattern selection based on *downward pattern refinement* in Sec. 3.3.

3.1 Sufficiently Accurate Distance Heuristics

In this section, we derive a precise measure for abstractions to obtain informed pattern database heuristics. As already outlined above, the most important question in this context is the question about similarity. At the extreme end of the spectrum of possible abstractions, one could choose a pattern that leads to bisimilar abstractions to the original system. This yields a pattern database heuristic $h^{\mathcal{P}}$ that is *perfect*, i. e., $h^{\mathcal{P}}(s) = d(s)$ for all states s, where d is the real error distance function. However, apart from being not feasible in practice, we will see that this condition is stricter than needed for obtaining perfect search behavior. It suffices to require $h^{\mathcal{P}}(s) = d(s)$ only for states s that are possibly explored by A*. In this context, Pearl [18] gives a necessary and sufficient condition for a state to be explored by A*. Consider a model checking task $\langle \mathcal{M}, s_0 \rangle$ and let $d(s_0)$ denote the length of a shortest error trace of $\mathcal{M}$. Recall that the priority function of A* is $priority(s) = h(s) + c(s)$, where $c(s)$ is the length of a shortest trace from s_0 to s. Pearl shows that if $priority(s) < d(s_0)$, then s is necessarily explored by A*, whereas exploring s implies that $priority(s) \leq d(s_0)$. This gives rise to the following definition for a distance heuristic to be *sufficiently accurate*.

Definition 6 (Sufficiently accurate). *Let $\mathcal{M}$ be a system with shortest error trace length $d(s_0)$, $\mathcal{P} \subseteq \mathcal{M}$ be a pattern of $\mathcal{M}$, and $h^{\mathcal{P}}$ be the pattern database heuristic for $\mathcal{P}$. If $h^{\mathcal{P}}(s) = d(s)$ for all states s with $h^{\mathcal{P}}(s) + c(s) \leq d(s_0)$, then $\mathcal{M}|_{\mathcal{P}}$ is called a* sufficiently accurate abstraction *of $\mathcal{M}$, and $h^{\mathcal{P}}$ is called* sufficiently accurate distance heuristic *for $\mathcal{M}$.*

Obviously, the requirement for a distance heuristic $h^{\mathcal{P}}$ to be sufficiently accurate is weaker than the requirement $h^{\mathcal{P}}(s) = d(s)$ for *all* possible states. However, with the results given by Pearl, we still know that A* with a sufficiently accurate distance heuristic delivers perfect search behavior, i. e., the same search behavior as that of A* with d. This justifies Def. 6 and is stated formally in the following proposition.

Proposition 1. *Let $\mathcal{M}$ be a system, $h^{\mathcal{P}}$ be a distance heuristic that is sufficiently accurate for $\mathcal{M}$. Then the set of explored states with A* applied with d is equal to the set of explored states of A* applied with $h^{\mathcal{P}}$.*

Proof. The claim follows immediately from the results given by Pearl [18]. As $h^{\mathcal{P}}$ is sufficiently accurate, we know that for every state s that is possibly explored by A* applied with $h^{\mathcal{P}}$ it holds $h^{\mathcal{P}}(s) = d(s)$. Therefore, the behavior of A* with d and $h^{\mathcal{P}}$ is identical.

As a first and immediate result of the above considerations, it suffices to have patterns that lead to sufficiently accurate distance heuristics to obtain perfect search behavior with A*. On the one hand, this notion is intuitive and reasonable. On the other hand, it is still of rather theoretical nature. It should be obvious that a sufficiently accurate heuristic is hard to compute as it relies on *exact* error distances d; as a side remark, if d was given, the overall model checking problem would be already solved, and there would be no need to compute a pattern database heuristic. However, Def. 6 also provides a first intuitive way for approximating this property, an approximation which is described next.

According to Def. 6, an abstraction $\mathcal{M}|_{\mathcal{P}}$ is sufficiently accurate if $h^{\mathcal{P}}(s) = d(s)$ for all states s that are possibly explored by A*. In this case, the pattern database heuristic based on $\mathcal{M}|_{\mathcal{P}}$ is sufficiently accurate for $\mathcal{M}$. For the following considerations, note that $h^{\mathcal{P}}(s) = d(s|_{\mathcal{P}}, \mathcal{M}|_{\mathcal{P}})$, and therefore, a direct way to approximate this test is to use a *distance heuristic* h instead of d. This is reasonable as distance heuristics are designed exactly for the purpose of approximating d, and various distance heuristics have been proposed in the directed model checking literature. Furthermore, as checking *all* states that are possibly explored by A* is not feasible either, we check this property only for the *initial* system state. This is the only state for which we know a priori that it is explored by A*. Overall, this gives rise to the following definition of *relatively accurate* abstractions.

Definition 7 (Relatively accurate). *Let $\langle \mathcal{M}, s_0 \rangle$ be a model checking task with system $\mathcal{M}$ and initial state s_0. Further, let $\mathcal{P} \subseteq \mathcal{M}$ be a pattern of $\mathcal{M}$, and let $\mathcal{M}|_{\mathcal{P}}$ be the corresponding abstraction to $\mathcal{P}$ with abstract initial state $s_0|_{\mathcal{P}}$. Furthermore, let h be a distance heuristic, $h(s_0, \mathcal{M})$ the distance estimate of $s_0 \in S(\mathcal{M})$, and $h(s_0|_{\mathcal{P}}, \mathcal{M}|_{\mathcal{P}})$ the distance estimate of $s_0|_{\mathcal{P}} \in S(\mathcal{M}|_{\mathcal{P}})$. If*

$$h(s_0, \mathcal{M}) = h(s_0|_{\mathcal{P}}, \mathcal{M}|_{\mathcal{P}}),$$

then $\mathcal{M}|_{\mathcal{P}}$ is called the relatively accurate abstraction *of $\mathcal{M}$ induced by h and $\mathcal{P}$.*

Obviously, the quality of this approximation strongly depends on the quality of the applied distance heuristic h. We want to emphasize that, according to the above definition, we apply a *second* distance heuristic h to determine our pattern database heuristic $h^{\mathcal{P}}$. In the experimental section, we will see that even this rather simple approximation of sufficient accurateness yields very informed abstract systems for sophisticated h.

3.2 Concretizable Traces and Safe Abstractions

In addition to the criterion from the last section, we derive a sufficient criterion for a distance heuristic to be sufficiently accurate that is still weaker than the requirement $h^{\mathcal{P}}(s) = d(s)$ for *all* states s. It is based on the observation that abstract systems where every spurious error trace is longer than $d(s_0)$ are not harmful. This is stated formally in the following proposition.

Proposition 2. *Let $\mathcal{M}$ be a system, $\mathcal{P} \subseteq \mathcal{M}$ be a pattern such that every spurious error trace π in the corresponding abstraction $\mathcal{M}|_{\mathcal{P}}$ is longer than a shortest possible error trace in $\mathcal{M}$, i. e., $\|\pi\| > d(s_0)$. Then $h^{\mathcal{P}}$ is sufficiently accurate for $\mathcal{M}$.*

Proof. First, recall that $h^{\mathcal{P}}(s) \leq d(s)$ for all states $s \in S(\mathcal{M})$ because $\mathcal{M}|_{\mathcal{P}}$ is an overapproximation of $\mathcal{M}$. We show that $h^{\mathcal{P}}(s) + c(s) > d(s_0)$ for all states $s \in S(\mathcal{M})$ with $h^{\mathcal{P}}(s) < d(s)$. Assume $h^{\mathcal{P}}(s) < d(s)$ for a state $s \in S(\mathcal{M})$. Let $s|_{\mathcal{P}} \in S(\mathcal{M}|_{\mathcal{P}})$ be the corresponding abstract state to s. As $h^{\mathcal{P}}(s) < d(s)$, there is an abstract trace $\pi_{\mathcal{P}}$ that is spurious and contains $s|_{\mathcal{P}}$. As all spurious error traces are longer than $d(s_0)$ by assumption, we have $\|\pi_{\mathcal{P}}\| > d(s_0)$. Therefore, $\|\pi_{\mathcal{P}}\| = c^{\mathcal{P}}(s|_{\mathcal{P}}) + d^{\mathcal{P}}(s|_{\mathcal{P}}) > d(s_0)$, where $c^{\mathcal{P}}(s|_{\mathcal{P}})$ denotes the length of a shortest abstract trace from the initial abstract state to $s|_{\mathcal{P}}$, and $d^{\mathcal{P}}(s|_{\mathcal{P}})$ denotes the abstract error distance of $s|_{\mathcal{P}} \in S(\mathcal{M}|_{\mathcal{P}})$. As $d^{\mathcal{P}}(s|_{\mathcal{P}}) = h^{\mathcal{P}}(s)$ and $c(s) \geq c^{\mathcal{P}}(s|_{\mathcal{P}})$, we have $c(s) + h^{\mathcal{P}}(s) > d(s_0)$.

Again, identifying abstractions with the property given by the above proposition is computationally hard as it relies on checking *all* possible spurious error traces. In the following, we show that a subclass of abstractions for a slightly stronger condition can be identified efficiently. To be more precise, we focus on abstractions that only introduce spurious error traces that can be *concretized* in the following sense.

Definition 8 (Concretizable Trace). *Let $\mathcal{M}$ be a system, $\mathcal{P} \subseteq \mathcal{M}$ be a pattern, and $\mathcal{M}|_{\mathcal{P}}$ be the corresponding abstraction of $\mathcal{M}$. Let $\pi_{\mathcal{P}} = t_1^{\#}, \ldots, t_n^{\#}$ be an abstract error trace of $\mathcal{M}|_{\mathcal{P}}$ with corresponding concrete transitions $t_1, \ldots, t_n$ of $\mathcal{M}$. Let $\pi_{\mathcal{P}}$ be spurious, i. e., $t_1, \ldots, t_n$ is not a concrete error trace of $\mathcal{M}$. The error trace $\pi_{\mathcal{P}}$ is* concretizable *in $\mathcal{M}$ if and only if there is a concrete error trace*

$$\pi = \pi_0, t_1, \pi_1, t_2, \pi_2, \ldots, \pi_{n-1}, t_n, \pi_n$$

in $\mathcal{M}$ that embeds $t_1, \ldots, t_n$. The π_i are traces in $\mathcal{M}$ with $\|\pi_i\| \geq 0$, for $i \in \{0, \ldots, n\}$.

Informally speaking, an abstract trace $\pi_{\mathcal{P}}$ in $\mathcal{M}|_{\mathcal{P}}$ is concretizable in $\mathcal{M}$ if there is a concrete trace in $\mathcal{M}$ so that the corresponding abstract trace in $\mathcal{M}|_{\mathcal{P}}$ is equal to $\pi_{\mathcal{P}}$. Note that from the above definition, concretizable error traces are a subclass of spurious error traces; as a side remark, these are exactly those error traces that preserve dead-ends in $\mathcal{M}$, i. e., states from which no error state is reachable. In the following, we focus on finding abstractions that do not introduce error traces that are not concretizable. We observe that *safe abstraction* is an effective technique for this purpose.

Safe abstraction for directed model checking has been introduced by Wehrle and Helmert [21]. Essentially, processes identified by safe abstraction can change their locations independently of and without affecting any other process, and every possible location is reachable. We briefly give the formal definitions in the following, starting with the notion of *independent* processes.

Definition 9 (Independent process). *Let $\mathcal{M}$ be a system and let $p \in \mathcal{M}$ be a process. Process $p = \langle L, L^*, T \rangle$ is* independent *in $\mathcal{M}$ if for every $(l_1, a, l_2) \in T$ with $l_1 \neq l_2$ and every process $\langle L', L^{*\prime}, T' \rangle = p' \in \mathcal{M} \setminus \{p\}$, the following two conditions hold: For every $l' \in L'$, there is $(l', a, l'') \in T'$, and for every $(l', a, l'') \in T'$: $l' = l''$.*

According to the above definition, independent processes p can change their locations independently of the current locations of other processes, and changing locations in p has no side effect on other processes either. Based on this notion, we define safe processes as follows.

Definition 10 (Safe process). *Let $\mathcal{M}$ be a system and let $\langle L, L^*, T\rangle = p \in \mathcal{M}$ be a process. Process p is* safe *in $\mathcal{M}$ if p is independent in $\mathcal{M}$, and for all locations $l, l' \in L$ there is a sequence of local transitions in T that leads from l to l', i. e., p is strongly connected.*

Safe processes can be efficiently identified by an analysis of the system processes' causal dependencies. Wehrle and Helmert exploited this property by performing directed model checking directly on the safe abstracted system. Corresponding abstract error traces in $\mathcal{M}|_{\mathcal{P}}$ have been finally extended to a concrete error trace in $\mathcal{M}$. Doing so, however, is not optimality preserving: shortest abstract error traces in $\mathcal{M}|_{\mathcal{P}}$ may not correspond to *any* shortest error trace in $\mathcal{M}$.

In this work, we use safe abstraction in a different context, namely to select patterns for a pattern database. For the following proposition, we assume without loss of generality that for every process p the target location set for p is not empty, and each label a that occurs in a transition of any process also occurs in a transition of p. Under these assumptions, every abstract error trace can be concretized. This is summarized in the following proposition. A proof is given by Wehrle and Helmert [21].

Proposition 3. *Let $\mathcal{M}$ be a system and let $p \in \mathcal{M}$ be a safe process of $\mathcal{M}$. Let $\mathcal{P} = \mathcal{M}\backslash\{p\}$ be the pattern obtained by removing p from $\mathcal{M}$, and $\mathcal{M}|_{\mathcal{P}}$ be the corresponding abstract system. Then every abstract error trace in $\mathcal{M}|_{\mathcal{P}}$ is concretizable.*

We observe that under the assumptions of Prop. 3, the set of unconcretizable abstract error traces is empty, and of course, so is the set of shorter or equally long abstract traces $\{\pi_{\mathcal{P}} \mid \pi_{\mathcal{P}}$ is not concretizable and $\|\pi_{\mathcal{P}}\| \leq d(s_0)\}$. In other words, abstracting safe variables does not introduce error traces that are longer than $d(s_0)$ and are not concretizable. We observe that safe abstraction provides an effective technique to approximate Prop. 2, where the condition of spuriousness is strengthened to concretizablility. The causal analysis required for safe abstraction can be done statically, is cheap to compute, and identifies processes that have the property that corresponding abstract systems approximate the conditions of Prop. 2. At this point, we emphasize again that we do not claim to introduce safe abstraction; however, we rather *apply* this technique for a different purpose than it was originally introduced.

Overall, based on the computationally hard notion of sufficiently accurate distance heuristics and Prop. 2, we have introduced ways to find abstract systems that are similar to the original system. Based on these techniques, we propose an algorithm for the pattern selection in the next section.

3.3 An Algorithm for Pattern Selection Based on Downward Refinement

In this section, we put the pieces together. So far, we have identified the notion of sufficiently accurate abstractions, and proposed techniques for approximating these concepts. Based on these techniques, we introduce an algorithm for the pattern selection

which we call *downward pattern refinement*. It starts with the full pattern, and iteratively refines it as long as the confidence is high enough that the resulting abstraction yields an informed pattern database. The algorithm is shown in Figure 2.

```
1 function dpr(M, s_0, h):
2     P := M \ {p | p safe process in M}
3     for each p ∈ P do:
4         if h(s_0, M) = h(s_0|_{P\{p}}, M|_{P\{p}}) then:
5             P := P \ {p}
6             goto 3
7 return P
```

Fig. 2. The downward pattern refinement algorithm

Roughly speaking, the overall approach works as follows. We start with the pattern that contains all system processes. In this case, the resulting pattern database heuristic would deliver perfect search behavior. However, as we have already discussed, such systems usually become too huge and cannot be handled in general due to the state explosion problem. Therefore, we iteratively remove processes such that the resulting abstraction is still similar to the original system.

We start with identifying all processes that do not introduce error traces that are not concretizable. Therefore, we remove all processes that are safe according to the safe abstraction approach (line 2). From the resulting abstract system $\mathcal{M}|_{\mathcal{P}}$, we iteratively remove processes p that lead to relatively accurate abstractions for the given distance heuristic h, i. e., for which the distance estimate of the initial abstract state does not decrease (lines 3–6). In particular, in line 4, we check for the current abstraction $\mathcal{M}|_{\mathcal{P}}$ if it can be further abstracted without reducing the distance estimation provided by h. The search stops when no more processes can be removed without decreasing h, i. e., when a fixpoint is reached. Termination is guaranteed after at most $|\mathcal{P}|$ iterations as we remove one process from the pattern in each iteration. We finally return the obtained pattern (line 7). We remark that the order in which the processes are considered may influence the resulting pattern. However, in our experiments, we observed that the pattern is invariant with respect to this order.

4 Related Work

Directed model checking has recently found much attention in different versions to efficiently detect error states in concurrent systems [4, 6, 7, 10, 14, 15, 19–23]. In the following, we give a very brief comparison of downward pattern refinement with other PDB heuristics. Overall, they mainly differ from our approach in the pattern selection scheme. The h^{rd} heuristic [15] uses a counterexample-guided pattern selection scheme, where those variables that occur in a certain abstract error trace are selected. The pattern selection mechanism of the h^{coi} heuristic [19] is based on a cone of influence analysis. It is based on the idea that variables that occur "closer" to those variables of the property

are more important than other ones. The h^{pa} heuristic [10] splits a system into several parts and uses predicate abstraction to build a PDB heuristic for each of the parts. The resulting heuristic is the sum of all these heuristics.

Further admissible non-PDB heuristics are the h^L and h^{aa} heuristics. The underlying abstraction of the h^{aa} heuristic [4] is obtained by iteratively replacing two components of a system by an overapproximation of their cross product. The h^L heuristic is based on the monotonicity abstraction [14]. The main idea of this abstraction is that variables are set-valued and these sets grow monotonically over transition application. In the following section, we will provide an experimental comparison of these heuristics with our approach.

5 Evaluation

We have implemented downward pattern refinement into our model checker MCTA [16] and empirically evaluated its potential on a range of practically relevant systems coming from an industrial case study. We call the resulting heuristic h^{dpr}. We compare it with various other admissible distance heuristics as implemented in the directed model checking tools UPPAAL/DMC [13] and MCTA.

5.1 Implementation Details

As outlined in the preliminaries, we chose our formalism mainly to ease presentation. Actually, our benchmarks are modeled as timed automata consisting of finitely many parallel automata with clocks and bounded integer variables. The downward pattern refinement algorithm works directly on timed automata. In a nutshell, automata and integer variables correspond to processes in our formalism. As it is not overly important how abstractions of timed automata systems are built, we omit a more detailed description here for lack of space. We remark that this formalism is handled as in Kupferschmid et al.'s Russian doll approach [15].

To identify relatively accurate abstractions, we use the (inadmissible) h^U heuristic [14] for the following reasons. First, it is one of MCTA's fastest to compute heuristics for this purpose. This is an important property since the heuristic is often called for different patterns during the downward pattern refinement procedure. Second, among MCTA's fastest heuristics, the h^U heuristic is the most informed one. The more informed a heuristic is the better it is suited for the evaluation of patterns. As in the computation of the h^U heuristic clocks are ignored, we always include all clocks from the original system in the selected pattern. By doing so, the resulting h^{dpr} is able to reason about clocks. We will come back to this in Sec. 6.

To identify safe variables, each automaton and each bounded integer variable corresponds essentially to a process p in the sense of Def. 1. Both kinds of processes can be subject to safe abstraction as described in Sec. 3.2.

5.2 Experimental Setup

We evaluate the h^{dpr} distance heuristic with A* search by comparing it with other distance heuristics implemented in MCTA or UPPAAL/DMC. In more details, we compare

to h^{rd}, h^{coi}, h^{pa}, h^{aa} and h^{L} heuristics as described in the related work section. Furthermore, we compare to UPPAAL's[1] breadth-first search (*BFS*) as implemented in the current version (4.0.13). Note that we do not compare our method with inadmissible heuristics like the h^{U} heuristic, as we do not necessarily get shortest error traces when applied with A*. All experiments have been performed on an AMD Opteron 2.3 GHz system with 4 GByte of memory.

As benchmarks, we use the *Single-tracked Line Segment* case study, which comes from an industrial project partner of the UniForM-project [12]. The case study models a distributed real-time controller for a segment of tracks where trams share a piece of track. A distributed controller has to ensure that never two trams are simultaneously in the critical section driving in different directions. The controller was modeled in terms of PLC automata [3], which is an automata-like notation for real-time programs. With the tool MOBY/RT [17], we transformed the PLC automata system into abstractions of its semantics in terms of timed automata [1]. For the evaluation of our approach we chose the property that never both directions are given permission to enter the shared segment simultaneously. We use three problem families to evaluate our approach, denoted with C, D, and E. They have been obtained by applying different abstractions to the case study. For each of them, we constructed nine models of increasing size by decreasing the number of abstracted variables. Note that all these problems are very large. The number of variables in the C instances ranges from 15 to 28, the number of automata ranges from 5 to 10. The corresponding numbers in the D problems range from 29 to 54 (variables) and from 7 to 13 (automata). The E instances have 44 to 54 variables and 9 to 13 automata. We injected an error into the C and D examples by manipulating an upper time bound. The E instances are correct with respect to the chosen property.

5.3 Experimental Results

Our experimental results are presented in Table 1. We compare h^{dpr} with the other heuristics and UPPAAL's breadth-first search (*BFS*) in terms of total runtime (including the preprocessing to build the pattern database for the PDB heuristics) and in terms of number of explored concrete states during the actual model checking process. The results are impressive. Most strikingly, h^{dpr} is the only heuristic that is able to solve every (erroneous and correct) problem instance. Looking a bit more closely, we also observe that h^{dpr} is always among the fastest approaches. In the C instances, only h^{rd} is faster, whereas in the smaller D instances, h^{dpr} outperforms the other approaches except for D_1. The larger D instances cannot be handled by any of the other heuristics at all. Moreover, we observe that the pruning power of h^{dpr} is high, and hence, we are able to verify correct systems that are even out of scope for the current version of UPPAAL. In many cases, the initial system state s_0 is already evaluated to infinity; this means that there is provably no concrete error trace from s_0 and there is no need to search in the concrete system at all. In particular, this results in a total number of explored states of zero. We will discuss these points in more details in the next section.

[1] `http://www.uppaal.com/`

Table 1. Experimental results for A* search. Abbreviations: "runtime": overall runtime including any preprocessing in seconds, "explored states": number of explored states before an error state was encountered or the instance was proven correct, dashes indicate out of memory (> 4 GByte)

	runtime in s							explored states							trace
Inst.	h^{dpr}	h^{rd}	h^{coi}	h^{pa}	h^{aa}	h^{L}	*BFS*	h^{dpr}	h^{rd}	h^{coi}	h^{pa}	h^{aa}	h^{L}	*BFS*	length
erroneous instances															
C_1	1.8	0.7	0.6	1.1	0.1	0.1	0.2	55	130	130	7088	8649	8053	21008	54
C_2	2.3	1.0	1.6	1.2	0.4	0.2	0.4	55	89813	187	15742	21719	21956	55544	54
C_3	2.2	0.6	2.6	1.2	0.4	0.4	0.6	55	197	197	15586	28753	24951	74791	54
C_4	2.5	0.7	23.4	2.2	1.9	2.3	5.3	253	1140	466	108603	328415	170325	553265	55
C_5	2.6	0.9	223.9	6.8	12.6	18.2	46.7	1083	7530	2147	733761	2.5e+6	1.2e+6	4.0e+6	56
C_6	4.1	0.8	227.4	53.6	176.2	165.2	464.7	2380	39436	6229	7.4e+6	2.5e+7	1.0e+7	3.4e+7	56
C_7	3.7	1.3	227.7	–	–	–	–	3879	149993	16357	–	–	–	–	56
C_8	3.7	1.3	182.3	–	–	–	–	5048	158361	16353	–	–	–	–	56
C_9	3.3	1.3	–	–	–	–	–	12651	127895	–	–	–	–	–	57
D_1	5.9	81.1	217.6	9.0	2.3	28.0	76.6	2450	4.6e+6	414	475354	2.6e+6	888779	4.1e+6	78
D_2	6.7	218.5	213.5	35.6	11.5	134.0	458.4	4401	4223	4223	2.5e+6	1.4e+6	4.0e+6	2.2e+7	79
D_3	6.7	222.7	215.0	36.3	11.8	152.3	466.5	4713	2993	2993	2.5e+6	1.4e+6	4.6e+6	2.2e+7	79
D_4	7.1	218.7	216.3	27.9	9.9	79.7	404.4	979	2031	2031	2.0e+6	1.3e+6	2.4e+6	1.8e+7	79
D_5	48.9	–	–	–	–	–	–	75631	–	–	–	–	–	–	102
D_6	52.6	–	–	–	–	–	–	255486	–	–	–	–	–	–	103
D_7	55.5	–	–	–	–	–	–	131275	–	–	–	–	–	–	104
D_8	52.6	–	–	–	–	–	–	22267	–	–	–	–	–	–	104
D_9	55.3	–	–	–	–	–	–	11960	–	–	–	–	–	–	105
error-free instances															
E_1	5.9	–	1.5	1.6	0.2	0.3	0.3	0	–	59210	22571	24842	18533	43108	n/a
E_2	23.4	–	–	91.6	65.7	140.1	157.0	0	–	–	6.1e+6	6.4e+6	4.6e+6	1.1e+7	n/a
E_3	53.1	–	–	–	–	–	–	1	–	–	–	–	–	–	n/a
E_4	156.1	–	–	–	–	–	–	1	–	–	–	–	–	–	n/a
E_5	158.0	–	–	–	–	–	–	0	–	–	–	–	–	–	n/a
E_6	161.9	–	–	–	–	–	–	0	–	–	–	–	–	–	n/a
E_7	168.1	–	–	–	–	–	–	0	–	–	–	–	–	–	n/a
E_8	172.8	–	–	–	–	–	–	13	–	–	–	–	–	–	n/a
E_9	180.1	–	–	–	–	–	–	0	–	–	–	–	–	–	n/a

5.4 Directed Model Checking for Correct Systems?

As outlined in the introduction, directed model checking is tailored to the fast detection of reachable error states. The approach is sound and complete as only the order is influenced in which the states are explored. However, one may wonder why a technique that influences the order of explored states is also capable of efficiently proving a system correct. The answer is that admissible distance heuristics like h^{dpr}, i. e., heuristics h with $h(s) \leq d(s)$ for all states s and the real error distance function d, also admit pruning power in the following sense. If $h(s) = \infty$ for an admissible heuristic h and a state s, then there is no abstract error trace that starts from the corresponding abstract state of s. Therefore, s can be pruned without losing completeness because $d(s) = \infty$ as well, as there is no concrete error trace starting from s either. Therefore, the absence of error states might be shown without actually exploring the entire reachable state space. In our experiments, we observe that h^{dpr} is very successful for this purpose as well. This

is caused by the suitable abstraction found by our downward refinement algorithm that preserves much of the original system behavior. The other distance heuristics do not perform as well in this respect. This is either because the underlying abstraction is too coarse (and hence, not many states are recognized that can be pruned), or it is too large such that no pattern database could be built because of lack of memory. Obviously, the abstractions of h^{dpr} identify a sweet spot of the trade-off to be as succinct as possible on the one hand, and as accurate as possible on the other hand.

6 Conclusions

We have introduced an approach to find abstractions and to build pattern database heuristics by systematically exploiting a tractable notion of system similarity. Based on these techniques, we presented a powerful algorithm for selecting patterns based on downward refinement. The experimental evaluation shows impressive performance improvements compared to previously proposed, state-of-the-art distance heuristics on a range of large and complex real world problems. In particular, we have learned that directed model checking with admissible distance heuristics can also be successfully applied to verify correct systems. For both erroneous and correct systems, we are able to solve very large problems that could not be optimally solved before. Overall, we observe that directed model checking with abstraction based distance heuristics faces similar problems as other (abstraction based) approaches to solve model checking tasks. In all these areas, the common problem is to find abstractions that are both succinct and accurate. This is also reflected in the future work, where it will be interesting to further investigate the class of pattern database heuristics and, in particular, to find suitable abstractions for pattern databases. In this context, counterexample-guided abstraction refinement could serve as a technique to further push our approach. Moreover, for the class of timed automata, we expect that the development of heuristics that consider clocks in the computation of heuristic values (rather than ignoring them) will improve our approach as such heuristics are better suited for the evaluation of patterns.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS, `http://www.avacs.org/`).

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
2. Culberson, J.C., Schaeffer, J.: Pattern databases. Comp.Int. 14(3), 318–334 (1998)
3. Dierks, H.: Time, Abstraction and Heuristics – Automatic Verification and Planning of Timed Systems using Abstraction and Heuristics. Habilitation thesis, University of Oldenburg, Germany (2005)
4. Dräger, K., Finkbeiner, B., Podelski, A.: Directed model checking with distance-preserving abstractions. STTT 11(1), 27–37 (2009)

5. Edelkamp, S.: Planning with pattern databases. In: Proc. ECP. pp. 13–24 (2001)
6. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. STTT 5(2), 247–267 (2004)
7. Edelkamp, S., Schuppan, V., Bosnacki, D., Wijs, A., Fehnker, A., Aljazzar, H.: Survey on directed model checking. In: Proc. MOCHART. pp. 65–89. Springer (2008)
8. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Systems Science and Cybernetics 4(2), 100–107 (1968)
9. Hart, P.E., Nilsson, N.J., Raphael, B.: Correction to a formal basis for the heuristic determination of minimum cost paths. SIGART Newsletter 37, 28–29 (1972)
10. Hoffmann, J., Smaus, J.G., Rybalchenko, A., Kupferschmid, S., Podelski, A.: Using predicate abstraction to generate heuristic functions in Uppaal. In: Proc. MOCHART. pp. 51–66. Springer (2007)
11. Kozen, D.: Lower bounds for natural proof systems. In: Proc. FOCS. pp. 254–266. IEEE Computer Society (1977)
12. Krieg-Brückner, B., Peleska, J., Olderog, E.R., Baer, A.: The UniForM workbench, a universal development environment for formal methods. In: Proc. MF. pp. 1186–1205. Springer (1999)
13. Kupferschmid, S., Dräger, K., Hoffmann, J., Finkbeiner, B., Dierks, H., Podelski, A., Behrmann, G.: Uppaal/DMC – abstraction-based heuristics for directed model checking. In: Proc. TACAS. pp. 679–682. Springer (2007)
14. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an AI planning heuristic for directed model checking. In: Proc. SPIN. pp. 35–52. Springer (2006)
15. Kupferschmid, S., Hoffmann, J., Larsen, K.G.: Fast directed model checking via russian doll abstraction. In: Proc. TACAS. Springer (2008)
16. Kupferschmid, S., Wehrle, M., Nebel, B., Podelski, A.: Faster than Uppaal? In: Proc. CAV. Springer (2008)
17. Olderog, E.R., Dierks, H.: Moby/RT: A tool for specification and verification of real-time systems. J. UCS 9(2), 88–105 (2003)
18. Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley (1984)
19. Qian, K., Nymeyer, A.: Guided invariant model checking based on abstraction and symbolic pattern databases. In: Proc. TACAS. pp. 497–511. Springer (2004)
20. Smaus, J.G., Hoffmann, J.: Relaxation refinement: A new method to generate heuristic functions. In: Proc. MOCHART. pp. 146–164. Springer (2008)
21. Wehrle, M., Helmert, M.: The causal graph revisited for directed model checking. In: Proc. SAS. pp. 86–101. Springer (2009)
22. Wehrle, M., Kupferschmid, S.: Context-enhanced directed model checking. In: Proc. SPIN. Springer (2010)
23. Wehrle, M., Kupferschmid, S., Podelski, A.: Transition-based directed model checking. In: Proc. TACAS. pp. 186–200. Springer (2009)