Fast Directed Model Checking Via Russian Doll Abstraction

Sebastian Kupferschmid[1], Jörg Hoffmann[2], and Kim G. Larsen[3]

[1] University of Freiburg, Germany
kupfersc@informatik.uni-freiburg.de
[2] University of Innsbruck, STI, Austria
joerg.hoffmann@sti2.at
[3] Aalborg University, Denmark
kgl@cs.aau.dk

Abstract. Directed model checking aims at speeding up the search for bugs in a system through the use of heuristic functions. Such a function maps states to integers, estimating the state's distance to the nearest error state. The search gives a preference to states with lower estimates. The key issue is how to generate good heuristic functions, i. e., functions that guide the search quickly to an error state. An arsenal of heuristic functions has been developed in recent years. Significant progress was made, but many problems still prove to be notoriously hard. In particular, a body of work describes heuristic functions for model checking timed automata in UPPAAL, and tested them on a certain set of benchmarks. Into this arsenal we add another heuristic function. With previous heuristics, for the largest of the benchmarks it was only just possible to find some (unnecessarily long) error path. *With the new heuristic, we can find provably shortest error paths for these benchmarks in a matter of seconds.* The heuristic function is based on a kind of Russian Doll principle, where the heuristic for a given problem arises through using UPPAAL itself for the complete exploration of a simplified instance of the same problem. The simplification consists in removing those parts from the problem that are distant from the error property. As our empirical results confirm, this simplification often preserves the characteristic structure leading to the error.

1 Introduction

When model checking safety properties, the ultimate goal is to prove the absence of error states. This can be done by exploring the entire reachable state space. UPPAAL is a tool doing this for networks of extended timed automata. It has a highly optimized implementation, but still the reachable state space often is too large in realistic applications. A potentially much easier task is to try to *falsify* the safety property, by identifying an error path: for this, we can use a *heuristic* that determines in what order the states are explored. In our work, we enhance error detection in UPPAAL following such a strategy.

A heuristic, or *heuristic function*, is a function h that maps states to integers, estimating the state's distance to the nearest error state. The heuristic is called *admissible* if it provides a lower bound on the real error state distance. The search gives a preference to states with lower h value. There are many different ways of doing the latter.

The A^* method, where the search queue is a priority queue over start state distance plus the value of h, guarantees to find an optimal (shortest possible) error path if the heuristic is admissible. An alternative is *greedy best-first search*. There, the search queue is a priority queue over the value of h. This does not give any guarantee on the solution length, but is often (yet not always) faster than A^* in practice. Note that short error paths are important in practice, since the error path will be used for debugging purposes. The application of heuristic search to model checking was introduced a few years ago by Edelkamp et al. [1,2], naming this research direction *directed model checking*, and inspiring various other approaches of this sort, e. g. [3,4,5,6,7]. The main difference between these approaches is how they define and compute the heuristic function: *How does one estimate the distance to an error state?*

The following gives an overview of the heuristic functions defined so far. Edelkamp et al. [1] base their heuristics on the "graph distance" within each automaton – the number of edge traversals needed, disregarding synchronization and all state variables. This yields a rather simplistic estimation, but can be computed very quickly. Groce and Visser [3] define heuristics inspired from the area of testing, with the idea to prefer covering yet unexplored branches in the program. Qian and Nymeyer [4,8] ignore some of the state variables to define heuristics which are then used in a pattern database approach (see below). Kupferschmid et al. [5] adapt a heuristic method from the area of AI Planning, based on a notion of "monotonicity" where it is assumed that a state variable accumulates, rather than changes, its values. Dräger et al. [6] iteratively "merge" a pair of automata, i. e., compute their product and then merge locations until there are at most N locations left, where N is an input parameter. The heuristic function is read off the overall merged automaton. Hoffmann et al. [7] compute the state space of a predicate abstraction of the system to be checked, and use a mapping from real states into abstract states to compute the heuristic values.

We add another kind of heuristic functions into the above arsenal. Like Qian and Nymeyer's [4] techniques, our heuristic functions belong into the family of *pattern databases* (PDB), which were first explored in AI [9], more precisely for hard search problems in single agent games such as Rubik's Cube. A PDB heuristic function abstracts a problem by ignoring some of the relevant symbols, e. g., some of the state variables [4]. The state space of the abstracted problem is built completely as a preprocess to search, and is used as a look-up table for the heuristic values during search.

The main question to answer is, of course, which symbols should be ignored? How should we abstract the problem to obtain our PDB? In AI, see e. g. [9,10], most strategies are aimed at exploiting parts of the problem that are largely independent – the idea being to generate a separate PDB for each part, and accumulate the heuristic values. Indeed, Edelkamp et al.'s [1,2] heuristic can be viewed as an instance of this, where each PDB ignores all symbols except the program counter of one single automaton.

In our work, we extend and improve upon a new kind of strategy to choose a PDB abstraction. The strategy is particularly well suited for model checking; a first version of it was explored by Qian and Nymeyer [8]. It is based on what we call a *Russian Doll* principle. Rather than trying to split the entire system up into (more or less) independent parts, one homes in on the part of the system that is most relevant to the safety property, and *leaves that part entirely intact.*[1] Intuitively, this is more suitable for model checking than traditional AI techniques because *a particular combined behavior of the automata nearest to the safety property is often essential in how the error arises.* The child Russian Doll preserves such combined behaviors, and should hence provide useful search guidance. The excellent results we obtained in our benchmarks indicate that this is indeed the case, even with rather small abstractions/"child dolls".

Given the key idea of the Russian Doll strategy – keep all and only symbols that are of "immediate relevance" to the safety property to be checked – the question remains what is "relevant". Answering this question precisely involves solving the problem in the first place. However, one can design computationally easy strategies that are intuitively very adequate for model checking. The basic idea is to do some form of abstract cone-of-influence [11] computation, and ignore those symbols that do not appear in the cone-of-influence. Qian and Nymeyer [8] use a simple syntactic backward chaining process that iteratively collects variable names and requires the user to specify a threshold on the maximal considered "distance" – number of iterations – of the kept variables from the safety property. In our work, we use a more sophisticated procedure based on the abstraction techniques of Kupferschmid et al. [5]. The procedure selects a subset of the relevant symbols (automata, synchronization actions, clock variables, integer variables) based on an abstract error path. No user input is required. Once it is decided which parts to keep, our implementation outputs those parts in UPPAAL input language. In Russian Doll style, UPPAAL itself is then used to compute the entire state space of the abstracted problem, and that state space is stored and used as a look-up table for heuristic values during search.

With half of the related work discussed above, namely [5,6,7], we share the fact that we are working with UPPAAL, and we also share the set of benchmarks with these works. The benchmarks are meaningful in that they stem from two industrial case studies [12,13]. Table 1 gives a preview of our results with our "Russian Doll" approach; we re-implemented the two heuristic functions defined in [1]; for each of [5,6,7], we could run the original implementation; finally, we implemented the abstraction strategy of [8], for comparison with our more sophisticated abstraction strategy (we created the pattern database with UPPAAL for our strategy). Every entry in Table 1 gives the total runtime (seconds), as well as the length of the found error path. The result shown is the best one that could be achieved, on that instance, with the respective technique: from the data points with shortest error path length, we selected the one with the smallest runtime (detailed empirical results are given in Section 5). A dash means the technique runs out of memory on a 4 GByte machine. Quite evidently, our approach drastically outperforms all the other approaches. This signifies a real boost in the performance of directed model checking, at least on these benchmarks.

The paper is organized as follows. Section 2 introduces notations. Section 3 explains some technicalities regarding possible sets of symbols to be ignored, and regarding the generation of a pattern database using UPPAAL. Section 4 introduces our Russian Doll strategy for choosing the symbols to be ignored. Section 5 contains our empirical evaluation, Section 6 discusses related work, and Section 7 concludes.

[1] We chose the name "Russian Doll" based on the intuition that the part left intact resembles the child Russian Doll, which is smaller but still characteristically similar to the parent.

Table 1. Results preview: total runtime / error path length

Exp.	[1]-best	[5]-best	[6]-best	[7]-best	[8]-best	Russian Doll
C_5	114.2 / 57	114.1 / 57	21.8 / 57	13.7 / 57	121.5 / 57	1.1 / 57
C_6	–	1211.7 / 57	291.5 / 57	85.2 / 57	–	1.3 / 57
C_7	–	–	309.1 / 855	204.5 / 1064	–	2.1 / 57
C_8	–	427.0 / 433	293.8 / 707	153.5 / 976	–	2.2 / 57
C_9	–	875.8 / 614	–	–	–	2.1 / 58

2 Notations

We assume the reader is roughly familiar with timed automata (TA) and their commonly used extensions; however, an in-depth familiarity is not necessary to understand the key contribution of this paper. Here, we give a brief description of the TA variant treated in our current implementation loosely following the terminology given by Behrmann et al. [14].

We treat networks of timed automata with binary synchronisation and integer variables. Our notations are as follows (all sets are finite). Each automaton i is a tuple $(L_i, X_i, V_i, A_i, E_i)$ where L_i is a set of locations, X_i is a set of clock variables, V_i is a set of integer variables, A_i is a set of actions, and E_i is a set of edges; these constructs will be explained below. The network consists of a set I of automata. By X, V, and A we denote $\bigcup_{i \in I} X_i$, $\bigcup_{i \in I} V_i$, and $\bigcup_{i \in I} A_i$, respectively. Importantly, each $x \in X$, $v \in V$, and $a \in A$ may appear in more than one automaton $i \in I$. In our Russian Doll abstractions, as stated, we ignore a set of "symbols". More precisely, such an *abstraction set* $\mathcal{A}$ will be a subset of $I \cup X \cup V \cup A$.

To denote the current locations of the automata, we assume a location variable loc_i for each $i \in I$, where the range of loc_i is L_i. A *state*, or *system state*, of the network is then given by a valuation of the variables loc_i, X, and V. Each $x \in X$ ranges over the non-negative reals. Each $v \in V$ has a finite domain dom_v. The action set A contains the internal action τ, and for each action $a? \in A$ there is a corresponding co-action $a! \in A$; for $a \in A$, we denote the co-action with $\overline{a}$. For each $i \in I$, the edges E_i are given as a subset of $L_i \times L_i$. Each edge $e \in E_i$ is annotated with an action $a_e \in A$, with a guard g_e, and with an effect f_e. The guard is a conjunction of conditions, each having the form of either $x \bowtie c$, or $x - y \bowtie c$, or $\mathit{lfn}(V') \bowtie c$, where $x, y \in X_i$, $\bowtie \in \{<, \leq, =, \geq, >\}$, c is a constant (a number), and $\mathit{lfn}(V')$ is a linear function in a variable set $V' \subseteq V_i$. The effect is a list of assignments, each of which either has the form $x := c$ or $v := \mathit{lfn}(V') + c$, where $v \in V_i$ and the other notations remain the same. Each variable $x \in X_i$ and $v \in V_i$ occurs on the left hand side of one such assignment at most. The semantics are defined as usual. Transitions are either asynchronous and triggered by an edge e where $a_e = \tau$, or synchronous and triggered by two edges $e \in E_i$ and $e' \in E_j$, $i \neq j$, so that $a_e = a?$ and $a_{e'} = a!$ for some $a?, a! \in A$. Each $i \in I$ has a start location $l_i^0 \in L_i$; each $v \in V$ has a start value $c_v^0 \in dom_v$; the start value of all clocks is 0.

As stated, we address the falsification of safety properties, also commonly referred to as invariants; in CTL, these properties take the form $AG\phi$. In our current implementation, ϕ takes the form $g \wedge (\bigvee_{i \in I'} \neg loc_i = l_i)$ where g has the same form as a guard, $I' \subseteq I$, and $l_i \in L_i$. A path of transitions is called an *error path* if it leads from the start state to a state that satisfies $\neg\phi$. An error path is *optimal* if there is no other error path that contains less transitions.

The above notations correspond to a subset of the UPPAAL input language; that language allows more powerful constructs such as non-binary synchronization, committed locations, and array manipulations. It is important to note that the restrictions imposed by the language subset are by no means inherent to our approach. Indeed, the only "language bottleneck" in our current implementation is the method choosing the abstraction set $\mathcal{A}$; as detailed in Section 4, this is based on methods from [5] which are as yet restricted to the above input language. Once $\mathcal{A}$ is chosen, UPPAAL itself is used to solve the abstracted problem, and so of course the whole of UPPAAL's input language can be handled. Hence, one can extend our technique simply by devising more generally applicable techniques for choosing $\mathcal{A}$.

3 Russian Doll Abstraction

This section presents the technicalities of generating the simplified problem in UPPAAL input language, and using UPPAAL itself to compute the heuristic function. We show how the simplified problem is generated based on an abstraction set $\mathcal{A}$, how the pattern database is built and used, and that the resulting heuristic estimates are admissible (i. e., lower bounds) provided $\mathcal{A}$ satisfies a certain property.

3.1 Abstraction Sets

Assume a network I of timed automata with the notations as specified, and a safety property $AG\phi$. As mentioned, an abstraction set is a set $\mathcal{A} \subseteq I \cup X \cup V \cup A$. The abstracted problem is generated as follows.

Definition 1. *Given a network I of timed automata and an abstraction set $\mathcal{A}$, the* abstraction of I under $\mathcal{A}$, *$\mathcal{A}(I)$, is defined as*

$$\{(L_i, X_i \setminus \mathcal{A}, V_i \setminus \mathcal{A}, A_i \setminus \mathcal{A}, \{\mathcal{A}(e) \mid e \in E_i\}) \mid i \in I \setminus \mathcal{A}\}$$

where $\mathcal{A}(e)$ is initialized to be equal to e and then modified as follows: if $a_e \in \mathcal{A}$ or $\overline{a_e} \in \mathcal{A}$, then $a_{\mathcal{A}(e)} := \tau$; if $x \in \mathcal{A}$ or $y \in \mathcal{A}$ for a guard or effect $x \bowtie c$, $x - y \bowtie c$, or $x := c$, then this guard/effect is removed; if $(\{v\} \cup V') \cap \mathcal{A} \neq \emptyset$ for a guard or effect lfn$(V') \bowtie c$ or $v :=$ lfn$(V') + c$, then this guard/effect is removed.

Given a safety property $AG\phi$, $\phi = g \wedge (\bigvee_{i \in I'} \neg loc_i = l_i)$, the abstraction of ϕ under $\mathcal{A}$, *$\mathcal{A}(\phi)$, is defined as $\mathcal{A}(g) \wedge (\bigvee_{i \in I' \setminus \mathcal{A}} \neg loc_i = l_i)$, where $\mathcal{A}(g)$ is defined as for guards above.*

In words, given an abstraction set $\mathcal{A}$, we simply ignore any automaton that appears in $\mathcal{A}$, as well as any guards or effects that involve variables or actions from $\mathcal{A}$.

It is important to note that this simple strategy does not always have the desired effect. Consider the case where automaton i has an edge e where $a_e = a?$ and automaton j has an edge e' where $a_{e'} = a!$. Say $i \in \mathcal{A}$ but $a! \notin \mathcal{A}$. Then potentially j can never traverse the edge e' because there is no one to synchronize with. A similar situation arises if f_e sets $v := v'$ and $g_{e'}$ demands $v = 7$, but $v' \in \mathcal{A}$ and $v \notin \mathcal{A}$. The following is a sufficient condition on $\mathcal{A}$ ensuring that such things do not happen.

Definition 2. *Given a network I of timed automata and an abstraction set $\mathcal{A}$, $\mathcal{A}$ is* closed *iff all of the following hold:*

- *If $i \in I \cap \mathcal{A}$ and $a \in A_i$, then $\overline{a} \in \mathcal{A}$*
- *If $i \in I \cap \mathcal{A}$ and $e \in E_i$ so that f_e sets $x := c$, then $x \in \mathcal{A}$*
- *If $i \in I \cap \mathcal{A}$ and $e \in E_i$ so that f_e sets $v :=$ lfn$(V') + c$, then $v \in \mathcal{A}$*
- *If $i \in I \setminus \mathcal{A}$ and $e \in E_i$ so that f_e sets $v :=$ lfn$(V') + c$ where $V' \cap \mathcal{A} \neq \emptyset$, then $v \in \mathcal{A}$*

We will see below that closed $\mathcal{A}$ yield admissible heuristic functions. Obviously, any $\mathcal{A}$ can be closed by extending it according to Definition 2.

3.2 Pattern Databases

As explained, pattern databases in our approach are obtained as the result of a complete state space exploration using UPPAAL. One subtlety to consider here is that, due to the continuous nature of the set of possible system states in timed automata, UPPAAL's search space does *not* coincide with the set of possible system states. Rather, each state s that UPPAAL considers corresponds to a set of system states where all automata locations and integer variables are fixed but the clock valuation can be any of a particular clock *region*. A clock region is given in the form of a (normalized) set of unary or binary constraints on the clock values, called *difference bound matrix*, which we denote by DBM_s. By $[s]$, we denote the set of system states corresponding to s.[2]

Our basic notions regard state spaces and error distances.

Definition 3. *Given a network I of timed automata, the* UPPAAL state space for I, *$\mathcal{S}(I)$, is a tuple (S, T, s_0), where S is the set of search states explored by* UPPAAL *when verifying a safety property $AG\phi$ with $\phi \equiv \top$, $T \subseteq S \times S$ are the possible transitions between those search states, and $s_0 \in S$ is the start state.*

Given also a safety property $AG\phi$, an error state *is a state $s \in S$ so that $s \models \neg\phi$. Given an arbitrary state $s \in S$, the* error distance of s in I with ϕ, *$d^{I,\phi}(s)$, is the length of a shortest path in (S, T) that leads from s to an error state, or $d^{I,\phi}(s) = \infty$ if there is no such path.*

Given Definition 3, it is now easy to state precisely what our pre-process to search does, when given a network I and a safety property $AG\phi$. First, an abstraction set $\mathcal{A}$ is chosen (with the techniques detailed below in Section 4). Then, UPPAAL is called to generate $\mathcal{S}(\mathcal{A}(I))$. The resulting tuple (S', T', s'_0) is redirected into a file, in a simple format. Once UPPAAL has stopped, an external program finds all error states in S', and computes $d^{\mathcal{A}(I),\mathcal{A}(\phi)}(s')$ for all $s' \in S'$, using a version of Dijkstra's algorithm with multiple sources. In other words, UPPAAL computes the state space of the abstracted problem, and an external program finds the distances to the abstracted error states.

It remains to specify how $\mathcal{S}(\mathcal{A}(I))$ and the $d^{\mathcal{A}(I),\mathcal{A}(\phi)}(s')$ are used to implement a heuristic function for solving I and $AG\phi$. The core operation is to map a state in $\mathcal{S}(I)$ onto a set of corresponding states in $\mathcal{S}(\mathcal{A}(I))$. For a UPPAAL state s, by $[s]|_{\mathcal{A}}$ we denote the projection of the system states in $[s]$ onto the variables not contained in $\mathcal{A}$.

[2] For the reader unfamiliar with timed automata, we want to add that our techniques apply also to discrete state spaces, in a manner that should become obvious in the following.

Definition 4. *Given a network I of timed automata with $\mathcal{S}(I) = (S, T, s_0)$, an abstraction set $\mathcal{A}$ with $\mathcal{S}(\mathcal{A}(I)) = (S', T', s'_0)$, and a state $s \in S$,* the abstraction of s under $\mathcal{A}$, *$\mathcal{A}(s)$, is defined as $\{s' \in S' \mid [s'] \cap [s]|_{\mathcal{A}} \neq \emptyset\}$. Given a safety property $AG\phi$, the* heuristic value of s under $\mathcal{A}$, *$h^{\mathcal{A}}(s)$, is defined as $min\{d^{\mathcal{A}(I),\mathcal{A}(\phi)}(s') \mid s' \in \mathcal{A}(s)\}$.*

Note that $[s'] \cap [s]|_{\mathcal{A}} \neq \emptyset$ may be the case for more than one s' because, and only because, UPPAAL's search states do not commit to one particular clock valuation. We have $[s'] \cap [s]|_{\mathcal{A}} \neq \emptyset$ if and only if s' and s agree completely on the automata locations of $I \setminus \mathcal{A}$ and on the values of $V \setminus \mathcal{A}$, and $DBM_{s'}$ is consistent with DBM_s.[3] Testing consistency of two DBMs is a standard operation for which UPPAAL provides a highly efficient implementation. Consequently, in our implementation, we store $\mathcal{S}(\mathcal{A}(I))$ in a hash table indexed on $I \setminus \mathcal{A}$ and $V \setminus \mathcal{A}$, where each table entry contains a list of DBMs, one for each corresponding abstract state s'; of course, $d^{\mathcal{A}(I),\mathcal{A}(\phi)}(s')$ is also stored in each list entry. Lookup of heuristic values is then realized via hash table lookup plus DBM consistency checks in the list, selecting the smallest $d^{\mathcal{A}(I),\mathcal{A}(\phi)}(s')$ of those s' for which the check succeeded.

Lemma 1. *Let I be a network of timed automata with $\mathcal{S}(I) = (S, T, s_0)$, let $\mathcal{A}$ be a closed abstraction set $\mathcal{A}$, and let $s \in S$ be a state. Then $h^{\mathcal{A}}(s) \leq d^{I,\phi}(s)$.*

Proof Sketch: Let $\mathcal{S}(\mathcal{A}(I)) = (S', T', s'_0)$. The key property is that, in the terms of [15], (S', T', s'_0) *approximates* (S, T, s_0): for any transition $(s_1, s_2) \in T$, either s and s' agree on the symbols not in $\mathcal{A}$, or there is a corresponding transition $(s'_1, s'_2) \in T'$. So transitions are preserved, and error path length can only get shorter in the abstraction. ■

Lemma 1 does *not* hold if $\mathcal{A}$ is not closed. This can be seen easily based on examples like those mentioned above Definition 2, where a symbol that is abstracted away can contribute to changing the status of a symbol that is not abstracted away. The importance of Lemma 1 is that, plugging our heuristic function into A^*, we can guarantee to find a shortest possible – an optimal – error path.

4 Choosing Abstraction Sets

Having specified how to proceed once an abstraction set $\mathcal{A}$ is chosen, it remains to clarify how that choice is made. In AI, the traditional design principle for pattern databases is to look for different parts of the problem that are largely independent, and to construct a separate pattern database for each of them, accumulating the heuristic values. This principle has been shown to be powerful (see e. g. [16,10]). Now, consider this design principle in model checking. An error typically arises due to some complex interaction between several automata. If one tears those automata apart, the information about this interaction is lost. A different approach, first mentioned by Qian and Nymeyer [8], is to keep only one pattern database that includes as much as possible of those parts of the network that are of immediate relevance to the safety property. The intuition is that the particular combined behavior responsible for the error should be preserved.

[3] In a discrete state space, s' and s agree completely on all non-abstracted variables, and so the mapping becomes simpler.

To realize this idea, one needs a definition of what is "close" to the safety property, and what is "distant". The notion of cone-of-influence [11] computation lends itself naturally to obtain such a definition. Qian and Nymeyer [8] use a simple method based on syntactic backward chaining over variable names. Herein, we introduce a more sophisticated method based on the abstraction techniques of Kupferschmid et al. [5]. As we shall see, this method leads to much better empirical behavior, at least in our tests with UPPAAL on networks of timed automata.

Qian and Nymeyer's [8] method starts with the symbols – automata, variables – mentioned in the safety property; this set of symbols forms layer 0. Then, iteratively, new layers are added, where layer $t + 1$ arises from layer t by including any symbol y that does not occur in a layer $t' \leq t$, and that may be involved in modifying the status of a symbol x in layer t, e. g., x and y may be variables and there may exist an assignment $x := exp(\bar{y})$ where $y \in \bar{y}$. The abstraction set is then chosen based on a cut-off value d supplied by the user: $\mathcal{A}$ will contain (exactly) all the symbols in layers $t > d$.

Intuitively, the problem with this syntactic backward chaining is that it is not discriminative enough between transitions that are actually relevant for violating the error property, and transitions that are not. In our experiments, we observed that, typically, the layers t converge to the entire set of symbols very quickly (in our largest benchmark example, this is the case at $t = 5$); when cutting off very early (at $t = 2$, e. g.), one misses some symbols that are important, and at the same time one includes many symbols that are not important.

Our key idea for improving on these difficulties is to do a more informed relevance analysis. We abstract the problem according to Kupferschmid et al. [5]: we compute an abstract error path with those authors' techniques, and set $\mathcal{A}$ to those symbols that are not affected by any of the transitions contained in the abstract error path. This way, we get a fairly targeted notion of what is relevant for reaching an error and what is not. The abstraction of Kupferschmid et al. [5] does not require any parameters, and hence as a side effect we also get rid of the need to request an input parameter from the user; i. e., our method for choosing $\mathcal{A}$ is fully automatic.

Describing Kupferschmid et al.'s [5] techniques in detail would breach the space limits of this paper and cannot be its purpose. For the sake of self-containedness, the following is a summary of the essential points. Kupferschmid et al.'s abstraction is based on the simplifying assumption that state variables accumulate rather than change, their values. The value $s(v)$ of a variable v in a state s is now a subset rather than an element, of v's domain. If v obtains a new value c, then c is included into $s(v)$ without removing any old values, i. e., the new value subset is defined by $s(v) := s(v) \cup \{c\}$. Hence the value range of each state variable grows monotonically over transitions, and hence Kupferschmid et al. call this the *monotonicity abstraction*.

Of course, the interpretation of formulas, such as transition guards, must be adapted to the new notion of states. This is done by existentially quantifying the state variables in the formula where each quantifier ranges over the value subset assigned to the respective variable in the state. It is easy to see that this abstraction is an over-approximation in the sense that the shortest abstract error path is never longer than the shortest real error path; it may be shorter.

The following example describes a situation where no real error path exists but only an absone. Say we have an integer variable v and one transition with guard $v = 0$ and effect $v := v + 1$. The start state is $v = 0$, and the safety property is $AGv < 2$. Obviously, the safety property is valid, i. e., there is no error path. However, such a path does exist in the abstraction. The abstract start state is $\{0\}$ which after one transition becomes $\{0, 1\}$. Since the transition guard is abstracted to $\exists c \in s(v) : c = 0$, the transition can be applied a second time and we get the state $\{0, 1, 2\}$: the new values obtained for v are 1 (inserting 0 into the effect right hand side) and 2 (inserting 1). The negated safety property, which is abstracted to $\exists c \in s(v) : c \geq 2$, is satisfied in that state.

Kupferschmid et al. develop a method that finds abstract error paths in time that is exponential only in the maximum number of variables of any linear expression over integer variables; i. e., the only exponential parameter is $max\{|V'| \mid ex.\ i, e : i \in I, e \in E_i, (\mathit{lfn}(V') \bowtie c) \in g_e$ or $(v := \mathit{lfn}(V') + c) \in f_e\}$. The method consists of two parts, a forward chaining and a backward chaining step. The forward chaining step simulates the simultaneous execution of all transitions in parallel, starting from the start state. In a layer-wise fashion, this computes for every state variable – i. e., for the location variables loc_i, $i \in I$, as well as the integer variables $v \in V$ and the clock variables $x \in X$ – what the subset of reachable values is. The forward step stops when it reaches a layer where the negation of the safety condition can be true. The backward step then starts at the state variable values falsifying the safety condition; it selects transitions that can be responsible for these values. The guards of these transitions yield new state variable values that must be achieved at an earlier layer. The process is iterated, selecting new transitions to support the new values and so on. The outcome of the process is a sequence $\langle t_1, \ldots, t_n \rangle$ of transitions that leads from the start state to a state falsifying the safety property, when executed under the monotonicity abstraction.

In our method for choosing the abstraction set $\mathcal{A}$, we execute Kupferschmid et al.'s algorithm exactly once to obtain an abstract error path $\bar{t} = \langle t_1, \ldots, t_n \rangle$ for the problem.[4] We then collect all symbols not affected by this path:

$$\begin{aligned}
\mathcal{A}_0 :=& \{i \in I \mid \text{not ex. } e \in \bar{t} \text{ s. t.} e \in E_i\} \cup \\
& \{a \in A \mid \text{not ex. } e \in \bar{t} \text{ s. t.} a_e = a\} \cup \\
& \{x \in X \mid \text{not ex. } e \in \bar{t}, c \text{ s. t.} (x := c) \in f_e, \text{ and} \\
& \qquad \text{not ex. } i \in \mathcal{A}_0, e \in E_i, c \text{ s. t.} (x \bowtie c) \in g_e, \text{ and} \\
& \qquad \text{not ex. } i \in \mathcal{A}_0, e \in E_i, y, c \text{ s. t.} (x - y \bowtie c) \in g_e\} \cup \\
& \{v \in V \mid \text{not ex. } e \in \bar{t}, \mathit{lfn}(V'), c \text{ s. t.} (v := \mathit{lfn}(V') + c) \in f_e, \text{ and} \\
& \qquad \text{not ex. } i \in \mathcal{A}_0, e \in E_i, \mathit{lfn}(V'), c \text{ s. t.} (\mathit{lfn}(V') \bowtie c) \in g_e \text{ and } v \in V'\}.
\end{aligned}$$

In this notation, $e \in \bar{t}$ is of course a shorthand for asking whether any of the transitions t_i involves e. In words, we keep all automata, actions, clock variables and integer variables that are modified on the path, and we keep all clock and integer variables that are relevant to a guard in an automaton that we keep. We obtain our final abstraction set $\mathcal{A}$ by closing $\mathcal{A}_0$ according to Definition 2.

[4] Actually we use a slightly modified version of the described backward chaining procedure, not considering indirect variable dependencies. We found this method to yield better performance, by selecting more relevant variable subsets.

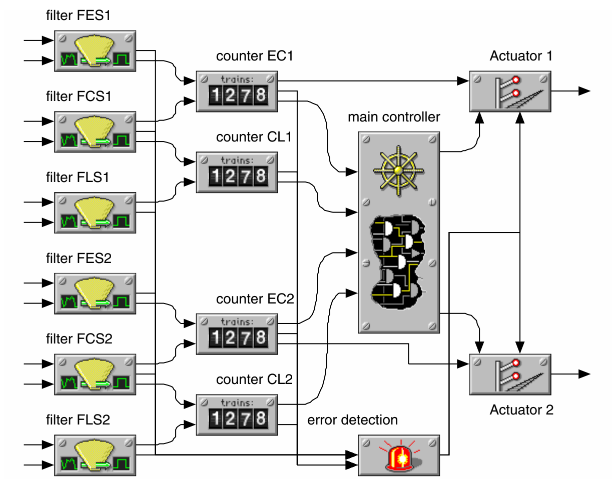

Fig. 1. The Single-tracked Line Segment case study

Kupferschmid et al.'s techniques form an appropriate basis for choosing $\mathcal{A}$ because they are computationally efficient, and they do provide useful information about relevance in the problem. Let us consider an example to illustrate this. Figure 1 illustrates one of our two industrial case studies, called "Single-tracked Line Segment". This study stems from an industrial project partner of the UniForM-project [12]. It concerns the design of a real-time controller for a segment of tracks where trams share a piece of track; each end of the shared piece of track is connected to two other tracks. The property to be checked requires that never both directions are given permission to enter the shared segment simultaneously. That property is not valid because some of the temporal conditions in the control automaton are not strict enough.

Let us consider Figure 1 in some more detail. As one would expect, **Actuator 1** and **Actuator 2** are the two automata in direct control of the signals allowing (signal up) or disallowing (signal down) a tram to enter the shared track. In particular, the safety property expresses that always at most one of those signals is up. The **main controller** automaton contains the (faulty) control logic that governs how the signals are set. The four **counter** automata count how many trains have passed on each of the four tracks that connect to the shared segment. The **error detection** detects inconsistencies between the counts, meaning that a train that should have left the shared segment is actually still inside it. Finally, each **filter** automaton receives an input variable from a sensor, and removes the noise from the signal by turning it into a step function based on a simple threshold test (so as to avoid, e. g., mistaking a passing truck for a tram).

The advantage of Kupferschmid et al.'s abstract error path for this example is that it touches only **Actuator 1**, **Actuator 2**, and the **control unit**. That is, the abstract error path involves exactly those automata that are immediately responsible for the error. Further, the abstract error path involves exactly the variables that are crucial in obtaining the error. The other – irrelevant – variables and automata have only an indirect influence on the error path, and need not be touched to obtain an error under the monotonicity abstraction. On the other hand, consider what happens if we apply Qian and Nymeyer's [8] syntactic backward chaining instead. In the start layer, indexed 0, of the chaining, we have only **Actuator 1** and **Actuator 2**. In the next layer, indexed 1, we correctly get the **control unit** – but we also get **error detection** and all of the **counter** automata. In just one more step, at layer 2, we get every automaton in the whole network. As if that wasn't bad enough, the relevant *variables* involved in producing the error appear much later, some of them in layer 5 only. Hence, based on this information, there is no way of separating the relevant symbols from the irrelevant ones.

5 Empirical Results

We ran experiments on an Intel Xeon 3.06 Ghz system with 4 GByte of RAM. We compare our heuristic to those of Edelkamp et al. [1] and Qian and Nymeyer [8] (both re-implemented), as well as those of Kupferschmid et al. [5], Dräger et al. [6], and Hoffmann et al. [7] (all in the original implementation). We further include results for UPPAAL's breadth-first search, which we abbreviate *BF*, and for UPPAAL's randomised depth-first search, abbreviated *rDF*. We distinguish between *optimal search* and *greedy search*. The former is BF, or A^* with an admissible (lower-bound) heuristic function; the latter is rDF, or greedy best-first search with any (possibly non-admissible) heuristic function. Table 2 shows the results for optimal search, Table 3 shows the results for greedy search. In the figures, our Russian Doll technique is indicated with *RD*. All other techniques are indicated in terms of the respective citations. If a technique requires a parameter setting, then we choose the setting that performs best in terms of total runtime; importantly, this does not compromise the other performance parameters: search space size and memory usage correlate positively with runtime, and error path length behavior does not vary significantly over parameter settings.

The "C_i", $i = 1, \ldots, 9$, examples in the figures come from the Single-tracked Line Segment case study that was explained above. Examples "M_i" and "N_i", $i = 1, \ldots, 4$, come from a study called "Mutual Exclusion". This study models a real-time protocol to ensure mutual exclusion of states in a distributed system via asynchronous communication. The protocol is described in full detail in [13]. The specifications are flawed due to an overly generous time bound. In all of the C_i, M_i, and N_i test beds, the size of the network scales with increasing i.

Consider first Table 2. The results for the C_i examples are striking. While all other techniques suffer from severe scalability issues, we can find the error in even the largest example in basically no time at all (C_2 is somewhat of an outlier). This is due to the quality of the heuristic function, which is clearly indicated in the number of search states explored by UPPAAL (note the direct effect that a smaller number of search states has on the peak memory usage). In the M_i and N_i examples, our technique is less dominant, but still performs better than the other techniques. The only somewhat bad cases are the smaller examples where the overhead for computing the Russian Doll pattern database does not pay off in terms of total runtime. Note that this is benign – what matters are the hard cases. It is remarkable that, consistently, our method explores at least one order of magnitude less search states than any of the others. This clearly indicates that, again,

Table 2. Results for optimal search. Notations: "runtime" is total runtime (including any pre-processes) in seconds; "search space" is the number of states UPPAAL explored before finding an error; "memory" is peak memory usage in MByte; "trace" is the length of the found error path; x e+y means $x \cdot 10^y$

	runtime				search space				memory				trace
	BF	[1]	[5]	[6]	BF	[1]	[5]	[6]	BF	[1]	[5]	[6]	
M$_1$	0.8	0.7	0.3	0.3	50001	50147	24035	19422	7	9	9	9	48
M$_2$	3.1	3.3	1.4	1.1	223662	223034	101253	77523	11	14	12	11	51
M$_3$	3.3	3.3	1.6	1.4	234587	231357	115008	94882	11	14	12	12	51
M$_4$	13.6	13.8	6.5	6.9	990513	971736	468127	436953	29	33	23	23	54
N$_1$	5.2	5.6	3.3	2.7	100183	99840	59573	46920	9	11	10	10	50
N$_2$	25.6	25.7	15.5	12.7	442556	446465	273235	211132	18	21	16	16	53
N$_3$	26.4	27.0	17.1	13.6	476622	473117	301963	238161	17	20	16	16	53
N$_4$	120.0	118.3	79.0	68.2	2.0e+6	2.0e+6	1.3e+6	1.1e+6	65	57	40	39	56
C$_1$	0.3	0.2	0.6	0.1	35325	35768	17570	9784	7	11	10	10	55
C$_2$	0.9	0.8	1.5	0.4	109583	110593	46945	34644	10	18	13	12	55
C$_3$	1.2	1.1	1.8	0.5	143013	144199	53081	40078	11	21	14	13	55
C$_4$	10.8	10.6	14.8	2.9	1.4e+6	1.4e+6	451755	324080	78	124	52	41	56
C$_5$	114.0	114.2	114.1	21.8	1.2e+7	1.2e+7	3.4e+6	2.4e+6	574	927	329	246	57
C$_6$	–	–	1211.7	291.5	–	–	3.2e+7	2.4e+7	–	–	2880	2402	57
C$_7$	–	–	–	–	–	–	–	–	–	–	–	–	57
C$_8$	–	–	–	–	–	–	–	–	–	–	–	–	57
C$_9$	–	–	–	–	–	–	–	–	–	–	–	–	58

	runtime			search space			memory			trace
	[7]	[8]	RD	[7]	[8]	RD	[7]	[8]	RD	
M$_1$	1.4	0.7	4.8	22634	28788	190	9	12	13	48
M$_2$	2.8	2.9	5.0	94602	121594	4417	12	23	13	51
M$_3$	3.2	3.1	5.2	121559	131482	11006	12	24	15	51
M$_4$	9.0	12.8	6.2	466967	543872	41359	24	67	20	54
N$_1$	4.6	4.9	26.8	46966	61830	345	10	21	21	50
N$_2$	13.7	27.2	17.7	211935	271912	3811	16	71	21	53
N$_3$	14.6	30.0	22.4	233609	298208	59062	16	74	33	53
N$_4$	58.6	154.3	55.5	1.0e+6	1.2e+6	341928	39	305	105	56
C$_1$	3.5	0.4	1.0	7088	30201	130	10	14	9	55
C$_2$	3.7	1.0	1.7	15742	95560	89813	11	25	27	55
C$_3$	3.7	1.4	0.9	15586	127327	197	12	31	9	55
C$_4$	6.1	12.4	1.0	108603	1.2e+6	1140	23	181	10	56
C$_5$	13.7	121.5	1.1	733761	1.1e+7	7530	194	1479	11	57
C$_6$	85.2	–	1.3	7.3e+6	–	39435	745	–	16	57
C$_7$	–	–	2.1	–	–	149993	–	–	32	57
C$_8$	–	–	2.2	–	–	158361	–	–	34	57
C$_9$	–	–	2.1	–	–	127895	–	–	39	58

our approach yields the best quality search information (the relatively high memory usage for N$_4$ is mostly due to the size of the pattern database).

Consider now Table 3, the data for the greedy searches. The techniques by Kupferschmid et al. [5], Dräger et al. [6], and Hoffmann et al. [7] all perform much better,

Table 3. Results for greedy search. Notations as in Table 2; "K" means thousand.

	runtime				search space				memory				trace			
Exp	rDF	[1]	[5]	[6]	rDF	[1]	[5]	[6]	rDF	[1]	[5]	[6]	rDF	[1]	[5]	[6]
M_1	0.8	0.4	0.0	0.3	29607	31927	5656	21260	7	9	8	9	1072	1349	169	90
M_2	3.1	2.8	0.3	1.0	118341	203051	30743	78117	10	15	10	10	3875	7695	431	102
M_3	2.8	1.6	0.2	1.1	102883	174655	18431	85301	9	12	9	10	3727	5412	231	105
M_4	12.7	7.3	1.2	3.8	543238	579494	122973	287122	22	28	15	16	15K	5819	849	124
N_1	1.9	1.3	0.4	1.2	41218	42931	16335	30970	7	9	9	9	1116	1695	396	110
N_2	9.3	9.5	2.4	5.8	199631	264930	88537	149013	13	17	12	12	4775	9279	990	127
N_3	8.4	4.9	0.6	6.0	195886	134798	28889	158585	12	14	10	12	3938	1656	324	108
N_4	40.9	52.1	4.9	31.6	878706	1.5e+6	226698	785921	39	60	20	32	18K	1986	1199	147
C_1	0.8	0.1	0.1	0.1	25219	19263	2368	2025	7	11	9	10	1056	794	95	149
C_2	1.0	0.4	0.2	0.2	65388	68070	5195	4740	8	15	9	10	875	962	84	198
C_3	1.1	0.6	0.3	0.2	85940	97733	6685	6970	10	19	9	10	760	916	109	198
C_4	8.4	6.1	2.5	0.5	892327	979581	55480	31628	43	96	16	12	1644	2305	142	173
C_5	72.4	69.4	20.8	2.6	8.0e+6	8.8e+6	465796	260088	295	734	68	37	2425	2708	330	268
C_6	–	–	177.4	23.3	–	–	4.5e+6	2.9e+6	–	–	519	303	–	–	490	377
C_7	–	–	–	309.1	–	–	–	2.9e+7	–	–	–	2600	–	–	–	855
C_8	–	–	427.0	293.8	–	–	1.2e+7	2.8e+7	–	–	1266	2608	–	–	433	707
C_9	–	–	875.8	–	–	–	2.0e+7	–	–	–	1946	–	–	–	614	–

	runtime			search space			memory			trace		
Exp	[7]	[8]	RD	[7]	[8]	RD	[7]	[8]	RD	[7]	[8]	RD
M_1	1.4	0.2	4.8	23257	11284	249	9	9	13	51	169	56
M_2	2.4	1.0	4.8	84475	59667	495	12	15	13	53	476	77
M_3	2.5	1.4	4.9	92548	85629	993	12	17	13	56	589	54
M_4	5.6	3.3	5.1	311049	216938	3577	24	32	13	56	419	106
N_1	3.2	0.5	26.5	31593	13902	242	10	11	21	55	159	57
N_2	8.7	3.8	17.7	172531	93467	470	16	28	20	58	624	64
N_3	8.2	5.3	15.4	167350	104104	1787	16	28	19	58	493	71
N_4	39.4	30.7	10.3	975816	422499	10394	39	93	19	61	242	81
C_1	3.2	0.3	0.9	1588	23173	130	10	13	9	159	65	55
C_2	3.5	0.8	1.2	3786	75111	56894	10	21	21	181	77	128
C_3	3.6	1.1	1.0	3846	101049	290	10	26	9	187	75	57
C_4	4.9	8.8	1.1	30741	1.0e+6	1163	14	151	10	241	86	58
C_5	7.1	84.3	1.4	185730	9.1e+6	39837	31	1075	18	423	124	76
C_6	23.6	–	1.7	1.9e+6	–	80878	195	–	25	757	–	65
C_7	204.5	–	6.7	1.8e+7	–	697116	1591	–	129	1064	–	65
C_8	153.5	–	10.4	1.4e+7	–	1.1e+6	1282	–	194	976	–	98
C_9	–	–	20.0	–	–	2.2e+6	–	–	355	–	–	109

compared to the optimal search in Table 2, in terms of runtime, search space size, and peak memory usage. This improvement is bought at the cost of significantly overlong error paths; in most cases, the returned error paths are more than an order of magnitude longer than the shortest possible error path. For rDF and the heuristic functions by Edelkamp et al. [1], the path length increase is even more drastic, by another order of magnitude, and with only a moderate gain in runtime. Qian and Nymeyer's [8] heuristic function yields much improved runtime behavior in M_i and N_i at the cost of significantly overlong error paths; in C_i, greedy search does not make much of a difference.

Finally, consider our RD technique. In M_i and N_i, the search space size performance is drastically improved now beating the other techniques quite convincingly (but not as convincingly in terms of runtime, where [8] is very competitive except in N_4). In C_i, the search spaces become a little larger; it is not clear to us what the reason for that is. The loss in error path quality is relatively minor.

In summary, the empirical results clearly show how superior our Russian Doll heuristic function is, on these examples, in comparison to previous techniques.

6 Related Work

We have already listed the previous methods for generating heuristic functions for directed model checking [3,4,8,5,6,7]. By far the closest relative to our work is the work by Qian and Nymeyer [8] which uses an intuitively similar strategy for generating pattern database heuristics. As we have shown, our improved strategy yields much better heuristic functions, at least in our suite of benchmarks. It remains to be seen whether that is also the case for other problems. It should also be noted that Qian and Nymeyer [8] use their heuristic function in a rather unusual BDD-based iterative deepening A^* procedure, and compare that to a BDD-based breadth-first search. As the authors state themselves, it is not clear in this configuration how much of their empirically observed improvements is due to the heuristic guidance, and how much of it is due to all the other differences between the two search procedures. In our work, we use standard heuristic search algorithms. We finally note that Qian and Nymeyer [8] state as the foremost topic for future work to find better techniques choosing the abstraction; this is exactly what we have done in this paper.[5]

7 Conclusion

We have explored a novel strategy for generating pattern database heuristics for directed model checking. As it turns out, this strategy results in an unprecedented efficiency of detecting error paths, solving within a few seconds, and to optimality, several benchmarks that were previously hardly solvable at all.

Our empirical results must of course be related to the benchmarks on which they were obtained, and it is a priori not clear to what extent they will carry over to other model checking problems. However, there certainly is a non-zero chance that they *will* carry over. This makes the further exploration of this kind of strategy an exciting direction, which we hope will inspire other researchers as well.

Acknowledgements

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `http://www.avacs.org/` for more information.

[5] We remark on the side that we developed our technique independently from Qian and Nymeyer [8], and only became aware of their work later.

References

1. Leue, S., Edelkamp, S., Lluch Lafuente, A.: Directed Explicit Model Checking with HSF-SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 57–79. Springer, Heidelberg (2001)
2. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. STTT 5, 247–267 (2004)
3. Groce, A., Visser, W.: Model checking Java programs using structural heuristics. In: Proc. ISSTA, pp. 12–21. ACM, New York (2002)
4. Nymeyer, A., Qian, K.: Guided Invariant Model Checking Based on Abstraction and Symbolic Pattern Databases. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 497–511. Springer, Heidelberg (2004)
5. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an AI Planning Heuristic for Directed Model Checking. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 35–52. Springer, Heidelberg (2006)
6. Drãger, K., Finkbeiner, B., Podelski, A.: Directed model checking with distance-preserving abstractions. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 19–34. Springer, Heidelberg (2006)
7. Hoffmann, J., Smaus, J.G., Rybalchenko, A., Kupferschmid, S., Podelski, A.: Using predicate abstraction to generate heuristic functions in UPPAAL. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS (LNAI), vol. 4428, pp. 51–66. Springer, Heidelberg (2007)
8. Qian, K., Nymeyer, A., Susanto, S.: Abstraction-guided model checking using symbolic ida* and heuristic synthesis. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 275–289. Springer, Heidelberg (2005)
9. Culberson, J., Schaeffer, J.: Pattern databases. Comp. Int. 14, 318–334 (1998)
10. Haslum, P., Botea, A., Helmert, M., Bonet, B., Koenig, S.: Domain-independent construction of pattern database heuristics for cost-optimal planning. In: Proc. AAAI (2007)
11. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
12. Krieg-Brückner, B., Peleska, J., Olderog, E., Baer, A.: The $UniForMWorkbench$, a universal development environment for formal methods. In: Woodcock, J.C.P., Davies, J., Wing, J.M. (eds.) FM 1999. LNCS, vol. 1709, Springer, Heidelberg (1999)
13. Dierks, H.: Comparing Model-Checking and Logical Reasoning for Real-Time Systems. Formal Aspects of Computing 16, 104–120 (2004)
14. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
15. Clarke, E.M., Grumberg, O., Long, D.E.: Model Checking and Abstraction. ACM Transactions on Programming Languages and Systems 16, 1512–1542 (1994)
16. Korf, R.E., Felner, A.: Disjoint pattern database heuristics. AIJ 134, 9–22 (2002)