Using Predicate Abstraction to Generate Heuristic Functions in UPPAAL

Jörg Hoffmann[1*], Jan-Georg Smaus[2], Andrey Rybalchenko[3,4],
Sebastian Kupferschmid[2], and Andreas Podelski[2]

[1] Digital Enterprise Research Institute (DERI), Innsbruck, Austria
[2] University of Freiburg, Germany
[3] Max Planck Institute for Computer Science, Saarbrücken, Germany
[4] Ecole Polytechnique Fédérale de Lausanne, Switzerland

Abstract. We focus on checking safety properties in networks of extended timed automata, with the well-known UPPAAL system. We show how to use predicate abstraction, in the sense used in model checking, to generate search guidance, in the sense used in Artificial Intelligence (AI). This contributes another family of heuristic functions to the growing body of work on *directed model checking*. The overall methodology follows the *pattern database* approach from AI: the abstract state space is exhaustively built in a pre-process, and used as a lookup table during search. While typically pattern databases use rather primitive abstractions ignoring some of the relevant symbols, we use *predicate abstraction*, dividing the state space into equivalence classes with respect to a list of logical expressions (predicates). We empirically explore the behavior of the resulting family of heuristics, in a meaningful set of benchmarks. In particular, while several challenges remain open, we show that one can easily obtain heuristic functions that are competitive with the state-of-the-art in directed model checking.

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.[5] UPPAAL has a highly optimized implementation, but still the reachable state space often is prohibitively 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. If the heuristic is perfect, then only the states on a shortest error path will have to be explored.

More formally, a heuristic, or *heuristic function*, is a function that maps states to integers, estimating the state's distance to the nearest error state. The heuristic is called

[*] Contact author. Email: joerg.hoffmann@deri.org

[5] Such (networks of) automata feature synchronization, integer variables, and real-valued clock variables. We assume the reader is (vaguely) familiar with these concepts; a brief explanation will be given. The ideas and results of the paper should be easily accessible without detailed background knowledge.

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) solution (error) path if the heuristic is admissible. Herein, we instead use *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 tends to be much faster than A^* in practice.

The application of heuristic search to model checking was pioneered a few years ago by Edelkamp et al [6, 7], christening this research direction *directed model checking*, and inspiring various other approaches of this sort, e.g. [5, 9, 13, 15]. The main difference between all the approaches is how they define and compute the heuristic function: *How does one estimate the distance to an error state?*

A brief overview of the heuristic functions defined so far is this. Edelkamp et al [6] 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 [9] define heuristics inspired from the area of testing, with the idea to prefer covering yet unexplored branches in the program. Qian and Nymeyer [15] ignore some of the state variables to define heuristics which are then used in a pattern database approach (see below). Kupferschmid et al [13] 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 [5] 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.

We add another family of heuristic functions into the above arsenal, based on an abstraction method already established quite broadly in model checking – *predicate abstraction* [8]. A predicate abstraction of a system is defined by a set of logical expressions – the predicates. In general, one could use arbitrary expressions; herein, we consider expressions of the form $\mathit{lfn}(X) \bowtie c$ where $\mathit{lfn}(X)$ is a linear function in variable set X, $\bowtie \in \{<, \leq, =, \geq, >\}$, and c is a constant (a number). The idea is to divide the state space into equivalence classes with respect to the truth values of the predicates: the abstraction of a system state s is a tuple $\bar{b}$ of truth values, stating which of the predicates are true or false in s; we have an abstract transition $\bar{b} \rightarrow \bar{b}'$ for every transition $s \rightarrow s'$ of the original system. The abstract system thus over-approximates the real system, which enables us to analyze the abstract system in order to draw (certain kinds of) conclusions about the real system. If an error state (condition) is not reachable in the abstract system, then it is neither reachable in the real system. Such methods have been extremely successful in the verification of temporal safety properties, e.g. [1, 2, 10].

Herein, predicate abstraction is, for the first time as far as we are aware, used to define heuristic functions instead. In a manner reminiscent of the *pattern database* approach [3], we build the (entire) abstract state space before search starts; during search, the abstract state space is used as a lookup table, i.e., states are mapped onto their abstract counterparts, and the error distance of the counterpart is taken as the heuristic estimate. In difference to our approach, pattern databases traditionally use simple abstractions, mostly (like [15] above) based on ignoring some of the relevant symbols.

An important characteristic of our method (which it shares with traditional pattern databases) is that it yields a very large family of heuristics, rather than just a single one. Every different set of predicates yields a different abstract state space, which gives a different heuristic function. The main question is: *How should we choose the predicates?* This is the same "main" question as in the standard use of predicate abstraction. However, in our approach the abstraction does not have to be precise enough to verify the property of interest, in order to be useful. So we got much more freedom of design. Herein, we explore two approaches. The first one simply collects the predicates from the syntax (e.g. transition guards) of the automata network, which is not likely to be property-preserving. The second one uses the standard *error path guided abstraction refinement* life cycle: (1) start with an empty set of predicates; (2) find an error path; (3) check if the error is real or spurious; (3a) if it is real, stop; (3b) if it is spurious, *analyze the error path* to create new predicates that exclude this path, and goto (2). For our purposes, we can stop the process at any time – we do not have to wait until a real error path is found. In our experiments, we simply fix a number of refinement iterations (which becomes an input parameter).[6] Generating the predicates using abstraction refinement is, we think, particularly promising: this technique has the power to adapt the heuristic function quite flexibly and intelligently to the individual problem instance at hand. Surprisingly, we weren't yet able to obtain entirely convincing results with the technique; we believe there is hope for the future. This will be discussed in detail later.

Apart from the parameterization given by the choice of predicates, we explore another parameter defining how the automata network is *split* into several parts. It turns out that predicate abstraction is much too time-consuming when done on the entire network. So we apply another abstraction method beforehand. We define a partitioning of the set of automata (we "split" the network), and hand each part to the predicate abstraction engine *in separate*. The splits are made so that few potential "interactions" are violated. The transition guards responsible for the violated interactions are removed. During search, a heuristic value is looked up in each part (in the corresponding abstract state space). These values are aggregated by taking their sum as the overall heuristic value. Since we removed the guards responsible for violated interactions, this aggregated heuristic value is still admissible. There are many possible strategies for making the split. We use a simple greedy strategy parameterized by the *split bound* b – the maximal number of automata in a single part of the partitioning (the maximal number of automata considered within a single predicate abstraction).

For testing, we use a set of benchmarks that is meaningful in that it includes examples from two industrial case studies [4, 12]. Table 1 gives a summary of our results. Examples "FAn" and "FBn" are variants of the Fischer protocol for mutual exclusion, examples "Mi", "Ni", and "Ci" come from the industrial case studies (some more details are given later). In Table 1, "RDF" is standard UPPAAL's fastest search strategy, a randomized depth-first search. "[13]-best" gives, for each example, the better result of

[6] We will see that, by using abstraction refinement for a heuristic – combining abstraction refinement and state space search – we can solve examples that cannot be solved by either method (abstraction refinement or blind state space search) alone.

Exp.	RDF	[13]-best	[5]-N50	Syntax-b2	ARMC-b3-r4
FA5	526/0.0	27/0.0	80/0.1	0.0/80/0.1	0.3/29/0.3
FA10	6371/0.4	42/0.0	130/0.2	0.0/130/0.1	0.4/44/0.4
FA15	20010/1.3	57/0.0	180/0.5	0.1/180/0.1	0.6/59/0.6
FB5	356/0.0	74/0.0	21/0.0	0.1/21/0.1	0.5/23/0.5
FB10	7885/0.5	274/0.0	36/0.1	0.3/36/0.3	0.5/38/0.5
FB15	58793/3.8	599/0.1	51/0.4	0.5/51/0.5	0.5/53/0.5
M1	29607/0.7	5656/0.7	19063/0.9	1.1/23257/1.8	3.9/12121/4.1
M2	118341/3.3	30743/2.8	46545/1.5	1.2/84475/3.9	4.0/50599/5.1
M3	102883/4.6	18431/2.0	64522/2.0	1.1/92548/4.4	4.0/28022/4.4
M4	543238/6.1	88537/9.9	168692/3.3	1.3/311049/11.4	4.5/116570/5.8
N1	41218/2.0	16335/2.4	27275/2.0	1.0/36030/3.8	7.3/12439/8.1
N2	199631/10.1	132711/9.9	102097/5.2	1.4/178333/14.0	7.8/97219/11.3
N3	195886/9.7	28889/3.3	135783/6.8	1.3/196535/15.4	7.5/43159/9.6
N4	878706/43.5	226698/23.1	483623/23.1	1.2/983344/75.1	8.2/469363/24.4
C1	25219/0.9	2368/0.8	871/0.5	2.8/1588/2.8	10.8/1172/10.8
C2	65388/1.0	5195/1.5	1600/0.6	3.0/3786/3.1	12.2/3256/12.2
C3	85940/1.7	6685/1.7	2481/0.8	2.6/3846/2.7	12.5/4278/12.5
C4	892327/9.7	55480/6.5	22223/1.4	4.6/30741/5.0	14.3/46837/14.8
C5	8.0e+6/81.9	465796/41.9	160951/3.2	5.0/185730/8.0	15.4/473470/20.0
C6	–	4.5e+6/353.0	1.7e+6/15.5	5.2/1.9e+6/33.9	16.5/2.6e+6/42.5

Table 1. Summary of results. Number of explored states/runtime (sec.) for RDF, [13]-best, and [5]-b50. Overhead/number of explored states/runtime (sec.) for Syntax-b2 and ARMC-b3-r4. Explanation see text.

the two heuristic functions defined in [13]. "[5]-N50" gives the result for the heuristic from [5] with $N = 50$.[7] "Syntax-b2" is predicate abstraction based on automata syntax, with split bound 2. "ARMC-b3-r4" is predicate abstraction based on abstraction refinement, with split bound 3 and 4 refinement iterations. The implementation interfaces to ARMC [14, 16], which does the abstraction refinement. The "overhead" given for "Syntax-b2" and "ARMC-b3-r4" is the total time taken to build abstract state spaces.

"Syntax-b2" and "ARMC-b3-r4" are, overall, the most successful configurations, of the many possible configurations of our code, that we found in our experiments. The experiments, in particular the effects of the different parameters on performance, are described in detail later (Sections 4 and 5). From a quick glance at Table 1, one sees that our new heuristics are indeed competitive with the heuristics defined by [13] and [5] (which, in turn, have been shown to outperform [6] and some other simple heuristics in these examples). "[13]-best" explores very few states in the "Mi" and "Ni" examples, "[5]-N50" and "Syntax-b2" are best in the "Ci" examples. Note that the blind search "RDF" cannot solve C6 (consumes more than 1GB memory). It is also notable that "[13]-best" takes more time to compute than the other heuristics – e.g., this can be seen in C5 where "[13]-best" takes 41.9 seconds to explore 465796 states, but "ARMC-b3-r4" takes only 4.6 seconds (20.0-15.4) to explore a slightly larger number of states.[8]

The main surprise in the data, for us, was the good performance of syntax-based predicate abstractions, like "Syntax-b2": we didn't expect that one can do so good with

[7] This is the sweet-spot; when increasing N, the number of explored states does not decrease much, but the overhead for merging the automata increases a lot.

[8] "[13]-best" is the only one of the tested heuristics that is *not* organized as a lookup table (the abstract problem has to be solved in every search state). Note that the results reported in [13] are better than those in Table 1; this is due to the additional "bitstate hashing" technique used there. Here, we focus exclusively on the heuristic functions.

such a simple form of abstraction predicates. In particular, we expected abstraction refinement to yield much better heuristics. The reason why this is not (yet) the case appears to lie in the following oddity. One would expect that a more refined heuristic yields a smaller search space.[9] However, in disquietingly many cases, refining the abstraction yielded a *larger* search space in our experiments. We believe there is hope to overcome this with modified refinement strategies; see Section 5.

The paper is organized as follows. The next section provides a brief background of, and some notations for, predicate abstraction. Section 3 summarizes the technical details of our approach: the formal definition of the predicate abstraction heuristic, a method to implement the needed "lookup tables" efficiently, and the technicalities of our network split operation. Section 4 gives detailed empirical observations for syntax-based predicate abstractions, Section 5 gives detailed empirical observations for predicate abstractions based on abstraction refinement. Section 6 concludes.

2 Predicate Abstraction

In principle, the idea behind this sort of abstraction is very simple. Say we have a transition system, specified declaratively via a set of transition rules, and a set X of variables. The state space of the system is the directed graph $\mathcal{S}$ where the nodes are all states (variable valuations) s, and the edges are the possible state transitions, as induced by the transition rules; we will use $\mathcal{S}$ also to denote the set of states. A predicate abstraction is defined by a finite set $\mathcal{P}$ of predicates over X. In our context, as said earlier, each $p \in \mathcal{P}$ has the form $\mathit{lfn}(X) \bowtie c$. Denote by a *bitvector* for $\mathcal{P}$ any conjunction that contains (exactly) each $p \in \mathcal{P}$, possibly negated. For a bitvector $\bar{b}$, denote by $[\bar{b}]$ the *extension* of $\bar{b}$, $[\bar{b}] := \{s \mid s \in \mathcal{S}, s \models \bar{b}\}$. The $[\bar{b}]$ are equivalence classes in $\mathcal{S}$; for $s \in [\bar{b}]$, we denote $[s] := [\bar{b}]$. The *abstract state space* for $\mathcal{P}$, denoted $[\mathcal{S}]^{\mathcal{P}}$, is the directed graph where the nodes are all bitvectors for $\mathcal{P}$, and there is an edge from $\bar{b}_1$ to $\bar{b}_2$ iff there exist $s_1 \in [\bar{b}_1]$ and $s_2 \in [\bar{b}_2]$ so that there is an edge from s_1 to s_2 in $\mathcal{S}$. Obviously, $[\mathcal{S}]^{\mathcal{P}}$ is an over-approximation of $\mathcal{S}$: if s_2 is reachable from s_1 in $\mathcal{S}$, then $[s_2]$ is reachable from $[s_1]$ in $[\mathcal{S}]^{\mathcal{P}}$.

Matters get a little more complicated once one starts to think about how to actually handle this sort of abstraction. When building the abstract state space, one has to frequently decide if there is an edge from a bitvector $\bar{b}_1$ to a bitvector $\bar{b}_2$. Enumerating $[\bar{b}_1]$ and $[\bar{b}_2]$ is, of course, nonsense. Instead, one formulates the transition rules of the real system as (conjunctions of) constraints on variable values (before and after the transition), so that the needed test comes down to the satisfiability of a conjunction of constraints. Both our own implementation and ARMC use this method, regressing from the error condition in a breadth-first manner to build the fraction of the abstract state space that is reachable from that condition. Concretely, both methods repeatedly consider a formula ϕ that formulates the properties of the state(s) that should be inserted next into the state space: ϕ is initially the error condition, and later the conjunction of the constraints given by the regressed abstract state, and the transition. The precise method to find the corresponding abstract states would be to enumerate all bitvectors

[9] It is easy to see that a more refined heuristic *dominates* a less refined heuristic; see Section 3.

and check if they are satisfiable in conjunction with ϕ. Instead, both our own implementation and ARMC additionally use a "cartesian" abstraction, and set the resulting state to $\bigwedge\{p \mid p \in \mathcal{P}, \phi \models p\} \wedge \bigwedge\{\neg p \mid p \in \mathcal{P}, \phi \models \neg p\}$. That is, one just checks which "bits" are definitely implied, and leaves the others unspecified. We denote such partial bitvectors with $\bar{c}$, keeping the notation $[\bar{c}] := \{s \mid s \in \mathcal{S}, s \models \bar{c}\}$. By $[c\mathcal{S}]^{\mathcal{P}}$ we denote the abstract state space – the graph of partial bitvectors – built in this way.

3 Technicalities

Before we start explaining the technical details of our approach, we fill in some details on the framework. As said, we consider networks of extended timed automata, which are finite automata (whose "states" are called "locations" in here) annotated with: 1. Effects and guards (constraints on edge executability) on integer variables; 2. Effects and guards on real-valued clock variables. The latter are a restricted form of real-valued variables: they always increase with the same speed as a linear function over time; their only allowed guards take the form $x \bowtie c$ or $x - y \bowtie c$; their only allowed effects take the form $x := c$ (where c denotes a constant). Networks of extended timed automata feature a set of such automata, that can share global integer and clock variables, and that can synchronize via synchronization actions. The latter actions generally take the form "send signal a" or "receive signal b". If (and only if) "a" and "b" match, the two edges can (and must) be taken simultaneously, as a single transition of the system. Arbitrarily many edges may be involved in such a transition if, e.g., the "send" action is a broadcast. In our implementation, so far we allow binary synchronization only, involving exactly two automata (one sender, one receiver).

Our safety properties take the form of formulas ϕ. In our implementation, so far we restrict to reachability, i.e., the question of whether a state fulfilling ϕ is reachable, where ϕ specifies a set of *target locations* and/or conditions on the clocks and integer variables.

UPPAAL tests safety properties in networks of extended timed automata by enumerating the space of reachable states. The search states correspond not to single system states, but to sets of such states. Namely, instead of concrete clock valuations, of which there are infinitely many, UPPAAL considers clock regions, whose (relevant) number is finite. 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*.

3.1 Predicate Abstraction Heuristics

To turn a predicate abstraction into a heuristic function, we simply map the state into the abstract state space, and read the error distance from there. Precisely, if ϕ is the error condition, $\mathcal{P}$ is the predicate set, and s is a system state, we get:

$$h^{\mathcal{P}}(s) := min\{dist^{\mathcal{P}}(\bar{c}_1, \bar{c}_2) \mid \bar{c}_1, \bar{c}_2 \in [c\mathcal{S}]^{\mathcal{P}}, s \in [\bar{c}_1], \exists s' \in [\bar{c}_2] : s' \models \phi\}.$$

Here, $dist^{\mathcal{P}}(.,.)$ is graph distance in $[c\mathcal{S}]^{\mathcal{P}}$, which is ∞ if there is no path from the first to the second argument. Note here that, since the bitvectors $\bar{c}$ in $[c\mathcal{S}]^{\mathcal{P}}$ are partial, there may be several $\bar{c}_1$ so that $s \in [\bar{c}_1]$. $h^{\mathcal{P}}(s) = \infty$ if no error state is reachable in $[c\mathcal{S}]^{\mathcal{P}}$ from any such $\bar{c}_1$, which implies that no error state is reachable in $\mathcal{S}$ from s. Since we minimize over all $\bar{c}_1$ (and $\bar{c}_2$), $h^{\mathcal{P}}$ is admissible:

Proposition 1. *For any non-temporal formula ϕ, predicate set $\mathcal{P}$, and $s \in \mathcal{S}$, we have $h^{\mathcal{P}}(s) \leq min\{dist(s, s') \mid s' \in \mathcal{S}, s' \models \phi\}$, where $dist(.,.)$ is graph distance in $\mathcal{S}$.*

Another interesting property of this kind of heuristics is that they are monotone in the predicate set, in the following sense:

Proposition 2. *For any non-temporal formula ϕ, predicate sets $\mathcal{P}_1$ and $\mathcal{P}_2$ such that $\mathcal{P}_1 \subseteq \mathcal{P}_2$, and $s \in \mathcal{S}$, we have $h^{\mathcal{P}_1}(s) \leq h^{\mathcal{P}_2}(s)$.*

This is simply because, in particular, $[\mathcal{S}]^{\mathcal{P}_2}$ makes all distinctions that $[\mathcal{S}]^{\mathcal{P}_1}$ makes. What it tells us is that, if we refine a predicate set $\mathcal{P}$ by inserting new predicates into it, we obtain a heuristic function that *dominates* the previous one, in that it provides a (potentially) better lower bound.

For use in UPPAAL, we have to modify the definition of $h^{\mathcal{P}}$ to work on UPPAAL *search* states – which correspond to *sets* of system states. Let s be a UPPAAL search state, and $[s]$ be the corresponding set of system states. We define:

$$h^{\mathcal{P}}(s) := min\{dist^{\mathcal{P}}(\bar{c}_1, \bar{c}_2) \mid \bar{c}_1, \bar{c}_2 \in [c\mathcal{S}]^{\mathcal{P}}, [s] \cap [\bar{c}_1] \neq \emptyset, \exists s' \in [\bar{c}_2] : s' \models \phi\}.$$

Obviously, again this leads to an admissible heuristic function, and the monotonicity in the predicate set is preserved. While the definition looks fairly complicated, we will see in the next section that, once the abstract state space $[c\mathcal{S}]^{\mathcal{P}}$ is built, the function can be implemented quite efficiently.

3.2 Predicate Abstraction Pattern Databases

For every search state UPPAAL encounters, the heuristic function must be computed. This makes it time-critical to implement that function efficiently. Our two main tricks are: 1. We formulate the mapping of search states into $[c\mathcal{S}]^{\mathcal{P}}$ as a "bitset" inclusion problem; 2. We use a tree data structure to address that inclusion problem efficiently.

Remember that the abstract states in $[c\mathcal{S}]^{\mathcal{P}}$ are partial bitvectors $\bar{c}$: sets of possibly negated predicates from $\mathcal{P}$ – sets of bits. Given a UPPAAL search state s, by $\phi(s)$ we denote the constraint conjunction corresponding to s: location and integer valuations plus difference bound matrix. We define the following bitset:

$$\bar{c}(s) := \{p \mid \phi(s) \models p\} \cup \{\neg p \mid \phi(s) \models \neg p\} \cup \{p, \neg p \mid \phi(s) \not\models p, \phi(s) \not\models \neg p\}.$$

In words, $\bar{c}(s)$ contains all bits that may possibly be true in s (that are satisfied by at least one system state in $[s]$). While the definition of $\bar{c}(s)$ involves entailment checks, due to our particular circumstances $\bar{c}(s)$ can be computed efficiently. First, the UPPAAL search state contains precise valuations for all locations and integer variables; the uncertainty is exclusively about the clocks. So predicates not involving clocks can simply be evaluated in s. Second, since UPPAAL itself allows only clock constraints of the form

$x \bowtie c$ or $x - y \bowtie c$, it is reasonable to also restrict to such constraints in the predicate sets. Whether such a predicate is implied by s or not can be read off from a single pass over the difference bound matrix of s. We observe:

$$[s] \cap [\bar{c}_1] \neq \emptyset \Leftrightarrow \bar{c}(s) \supseteq \bar{c}_1.$$

This is because $[s] \cap [\bar{c}_1] \neq \emptyset$ iff $\bar{c}_1$ contains no bit that is known to be false in s – in other words, if all bits contained in $\bar{c}_1$ may be true in s. We obtain:

$$h^{\mathcal{P}}(s) = min\{dist^{\mathcal{P}}(\bar{c}_1, \bar{c}_2) \mid \bar{c}_1, \bar{c}_2 \in [c\mathcal{S}]^{\mathcal{P}}, \bar{c}(s) \supseteq \bar{c}_1, \exists s' \in [\bar{c}_2] : s' \models \phi\}.$$

This is a syntactic characterization except for $\exists s' \in [\bar{c}_2] : s' \models \phi$; but that we have dealt with already when building $[c\mathcal{S}]^{\mathcal{P}}$: we simply mark, during that pre-process, the respective $\bar{c}_2$ (namely, the start state in our backward search) as error states. Of course, we also annotate each state $\bar{c}$ with its distance to the nearest error state (building $[c\mathcal{S}]^{\mathcal{P}}$ backward breadth-first, we get that distance for free).

We are left with the problems to: 1. In the pre-process, store $[c\mathcal{S}]^{\mathcal{P}}$ as a set of bitsets annotated with their error distance; 2. During search, quickly find all bitsets that are contained in $\bar{c}(s)$. Both can be accomplished using a data structure called *Unlimited Branching Tree* [11]. In a nutshell, this is a tree structure that stores sets of sets, exploiting shared elements between the sets to optimize space usage and access time for answering subset queries of the precise kind we need here. The details are not essential for the paper at hand, so we omit them. (A node in the tree may have as many branches as there are distinct elements in the sets, hence the name.)

3.3 Network Splitting

A very simple abstraction method is to partition the set of automata contained in a network. As said, we use this abstraction prior to predicate abstraction, in order to make the latter feasible. One simply considers each part of the partitioning – a subset of the automata – in separate. The only problem with this approach is that, of course, the automata typically interact with each other in various ways, and cannot be split without violating such interactions. We identify a possible definition of what “interaction” means. We approximate that definition to obtain an admissible splitting strategy.

Let e be an edge of automaton a, and e' be an edge of automaton $a' \neq a$. Let ψ be an effect of e (an assignment to a variable, or a synchronization action), and let ϕ be a guard of e' (a constraint over variables, or a synchronization action). We say that ψ *affects* ϕ if there is an execution trace P (a path in $\mathcal{S}$) so that: e occurs before e' on P; when removing ψ from e and simulating the execution of P by ignoring the guards between e and e', ϕ is no longer satisfied at the point where e' should be executed. Similarly, this definition is made also for location invariants ϕ. We say that an automaton a affects an automaton a' if there is an effect of an edge in a that affects a guard of an edge, or a location invariant, in a'. We say that a and a' *interact* if a affects a', or vice versa.

Proposition 3. *Say we have a network with automata A, a set of target locations ϕ, and a set $A_1 \subseteq A$ such that no automaton in $A \setminus A_1$ affects any automaton in A_1. Then,* *for any $s \in \mathcal{S}$, we have $min\{dist_{|1}(s_{|1}, s') \mid s' \in \mathcal{S}_{|1}, s' \models \phi_{|1}\} \leq min\{dist(s, s') \mid s' \in \mathcal{S}, s' \models \phi\}$, where $s_{|1}$ is s restricted to the locations and variables mentioned in A_1, $\mathcal{S}_{|1}$ is the state space of A_1, $\phi_{|1}$ is the subset of ϕ mentioned in A_1, and $dist(.,.)$ ($dist_{|1}(.,.)$) is graph distance in $\mathcal{S}$ ($\mathcal{S}_{|1}$).*

In words, the isolated automata A_1 provide an admissible distance estimate. The reason for this is, simply, that we can take any solution path for A, and restrict it onto the edges present in A_1, to obtain a solution for A_1 in isolation – otherwise, if that restricted path wasn't a solution for A_1, a constraint in A_1 would be unsatisfied, and we could construct a contradiction since an edge on the sub-path for $A \setminus A_1$ would have to affect that constraint. Note that, in particular, Proposition 3 says that, if A is solvable, then A_1 in isolation is also solvable. We further have:

Proposition 4. *Say we have a network with automata A, a set of target locations ϕ, and a partioning $A_1, \dots, A_m$ of A so that no pair of automata $a \in A_i$, $a' \in A_j$, $i \neq j$, interacts. Then, for any $s \in \mathcal{S}$, we have $\sum_{i=1}^{m} min\{dist_{|i}(s_{|i}, s') \mid s' \in \mathcal{S}_{|i}, s' \models \phi_{|i}\} \leq min\{dist(s, s') \mid s' \in \mathcal{S}, s' \models \phi\}$, where the notations are as in Proposition 3.*

This tells us that we can safely add the individual heuristic values. The reason is that we can partition any solution path for A into (independent) solution paths for each of $A_1, \dots, A_m$.

What we have just seen is not yet practical since there normally is no split that doesn't violate *any* interaction (otherwise there would be no point in posing both parts of the network within the same problem). We become practical by finding *potential* interactions, and simply removing guards that constitute violated potential interactions. Concretely, we use the simplistic notion saying that effect ψ can not affect condition ϕ if the variable x affected by ψ, and any variable that can, transitively, be affected by the value of x, does not appear in ϕ. For example, $x := 1$ can affect $x + y > 2$. On the other hand, $x := 1$ can affect $y > 2$ if there also is an effect $y := x$ somewhere, but not if there is no chain of variables from x to y. In our pre-process, we simply consider all pairs of occuring ψ and ϕ, and see if they satisfy this criterion; if not, we say that they have a potential interaction. We then greedily put automata together (into one part of the partitioning) so that few potential interactions to automata in other parts remain. For those interactions that do remain, we remove the responsible ϕ. Note that the latter will, in particular, remove synchronization actions that also occur in other parts.

With Proposition 4, our resulting heuristic function is still admissible, *except* for the effects of synchronization. In a solution path to A, a set of synchronized edges will be counted as a single transition, while in the partitioned network every edge will be counted separately.[10] We feel that this potential non-admissibility is benign. For example, if only binary synchronization is allowed, then the real error state distance is over-estimated by at most a factor 2 – in the *worst case*, which one can realistically expect to be far away from the typical case. We ran a number of tests using our heuristics with A^*, and never obtained a sub-optimal solution.

[10] Note that the automata network "A" we have here, in the application of Proposition 4, is no longer the original network, but one where several synchronization actions have been removed.

Exp.	Syntax-b1	Syntax-b2	Syntax-b3	Syntax-b4	Syntax-all
FA5	0.1/80/0.1	0.0/80/0.1	0.3/80/0.3	0.2/80/0.2	1.5/27/1.5
FA10	0.1/130/0.1	0.2/130/0.2	0.4/130/0.4	0.6/130/0.6	7.4/42/7.4
FA15	0.1/180/0.1	0.3/180/0.3	0.7/180/0.7	1.2/180/1.2	28.8/57/28.8
FB5	0.1/21/0.1	0.1/21/0.1	0.2/21/0.2	0.3/21/0.3	0.7/21/0.7
FB10	0.1/36/0.1	0.3/36/0.3	0.4/36/0.4	0.5/36/0.5	2.1/36/2.1
FB15	0.1/51/0.1	0.5/51/0.5	0.8/51/0.8	0.9/51/0.9	3.4/51/3.4
M1	0.3/16446/0.8	1.1/23257/1.8	6.3/12780/6.8	6.6/12780/7.0	
M2	0.3/68956/3.5	1.2/84475/3.9	9.6/37780/10.3	37.1/34947/37.8	
M3	0.3/62731/2.6	1.1/92548/4.4	9.1/55726/11.5	36.5/55098/37.8	
M4	0.4/275433/7.0	1.3/311049/11.4	13.5/198407/20.5	57.3/139875/62.6	–/
N1	0.5/22304/2.7	1.0/36030/3.8	8.8/17357/9.9	8.5/17357/10.2	
N2	0.5/122398/8.1	1.4/178333/14.0	11.7/87471/19.8	47.8/63596/53.3	
N3	0.5/140201/8.7	1.3/196535/15.4	11.2/115074/21.7	46.8/96202/56.7	
N4	0.6/738680/37.9	1.2/983344/75.1	16.3/720350/78.8	70.1/445359/120.3	–/
C1	0.9/1455/1.1	2.8/1588/2.8	9.5/4698/9.6	–/	–/
C2	1.4/3273/1.4	3.0/3786/3.1	9.5/10843/9.6	35.7/10507/35.9	–/
C3	1.4/5879/1.5	2.6/3846/2.7	9.7/10375/9.8	35.7/10195/35.9	–/
C4	1.6/44837/2.3	4.6/30741/5.0	21.2/66336/22.7	104.3/66761/105.7	–/
C5	1.8/301065/6.4	5.0/185730/8.0	21.7/436678/34.0	109.4/435309/121.5	–/
C6	1.6/2.8e+6/39.8	5.2/1.9e+6/33.9	26.7/4.1e+6/137.2	103.2/3.8e+6/230.0	–/

Table 2. Results for syntax-based abstractions. Notation as in Table 1; empty entries are identical to their left neighbour; "–/" means time-out during pre-processing.

4 Syntax-Based Abstractions

In our syntax-based abstractions, as indicated before, the abstraction predicates are simply read off the description of the automata network. Given a set A of automata, the created set of predicates consists of all expressions e that appear as a transition guard, or as a location invariant, of some automaton in A. Further, the abstraction distinguishes between the locations (which is equivalent to including the predicate $loc(a) = l$ for each $a \in A$ and location l of a).

Simply collecting all mentioned constraints, there is no parameterization to this sort of abstraction – except the need to split the automata network before the abstraction is applied. Table 2 shows our results. The data, and all other data reported herein, were obtained on a PC running at 1.2GHz with 1GB main memory and 1024KB cache running Linux. We set a time-out of 300 seconds for pre-processing. Search always ran out of memory before we ran out of patience (within a few minutes, that is).

Before considering the data, let us describe the examples in some more detail. Recall that our aim is *falsification* of safety properties, i.e., the systems we consider are unsafe in that an error state is reachable. Since the examples we considered were originally safe, we *injected* an error into them.

Examples "FAn" and "FBn" are variants of the Fischer protocol for mutual exclusion, which asks if at least two of n similar automata can be in a certain location simultaneously. We made the error possible by weakening one of the temporal conditions in the automata (from "$>$" to "$\geq$"). The variants differ in the way they encode the error condition. Variant A adds additional automata with synchronisation. Variant B selects and specifies two of the automata for the error condition. Examples "Mi" and "Ni", $i = 1, \ldots, 4$, 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 [4]. By increasing an upper time bound in the model we got a flawed specification that we transformed into its timed automata semantics by applying various abstractions techniques. Examples "Ci", $i = 1, \ldots, 6$, come from a case study called "Single-tracked Line Segment". This study stems from an industrial project partner of the UniForM-project [12] and the problem is to design a distributed real-time controller for a segment of tracks where trams share a piece of track. A distributed controller was modeled in terms of PLC-Automata [4, 12], and translated into timed automata. We injected an error by manipulating a delay such that the asynchronous communication between some automata is faulty. The given set of PLC-Automata had eight input variables and we constructed six models with decreasing size by abstracting more and more of these inputs.

In Table 2, the split bound increases from left to right as obvious; in "Syntax-all" there is no split bound, meaning that the entire network is handed to the abstraction engine in one piece. The foremost observation is that the latter is *bad* – except in the Fischer toy examples and the smaller "M" and "N" cases, the abstract state space could never be built within the alloted time (300 sec). ("M1" .. "M3" and "N1" .. "N3" have ≤ 4 automata so there is no change from "Syntax-b4" to "Syntax-all".) Consider the "M" and "N" examples, and what happens as b increases from 1 to 4. The overhead increases sharply, quickly becoming larger than the time spent in search. Strangely, the number of explored states also grows, from $b = 1$ to $b = 2$, before decreasing again from $b = 2$ to $b = 4$. It is unclear to us what causes this behavior. The smallest search spaces are obtained with $b = 4$, the smallest overall runtimes are obtained with $b = 1$. Note that the "M" and "N" examples can all be solved quite quickly even with a blind search, c.f. Table 1. The "C" examples are more interesting in that respect (blind search scales badly). They exhibit very similar behavior in terms of the overhead.[11] The number of explored states now decreases sharply from $b = 1$ to $b = 2$, and increases sharply from $b = 2$ to $b = 3$. Again, the reason for this behavior is unclear. C6 is the first example (in the scaling pattern here) where the larger overhead pays off in runtime – "Syntax-b2" is faster than "Syntax-b1" due to the reduced search space. Note here that C6 is the most relevant example.

All in all, the syntax-based abstractions give surprisingly good performance – e.g. "Syntax-b2" is very competitive in the "C" examples – but they don't seem to have much potential for further improvements. One could try to allow more freedom in the selection of the predicates, but such an approach is likely to be wild guesswork, at least without a deeper analysis of the system. An idea worth trying is to integrate syntax-based predicate selection into abstraction refinement: as a start, one could select (amongst others) the guards that are not satisfied by the spurious error path.

5 Abstraction Refinement

We implemented this sort of abstraction via an interface to the ARMC tool [14, 16]. This is a recent model checker based on error path guided abstraction refinement. Predicates are generated from spurious error paths by an analysis using a constraint based

[11] C1 is exceptionally hard for the pre-process since there are only 4 automata, which have a large abstract state space together; in C2 . . . C6, one of these automata is split away.

interpolation [17] to find a concise reason for the failure (the spuriousness) of the path. We modified ARMC to feature a maximal number of iterations as an input parameter. If ARMC finds a correct abstraction (no error paths), it stops with no output, causing our overall program to terminate – if there is no abstract error path, then there is no real one. (In our examples, of course this did not happen.) If ARMC finds a feasible (real) error path, it stops and outputs the abstract state space.[12] The same happens otherwise, i.e. if the maximum iteration is reached. The abstract state space is read in, and stored in a UBTree structure for lookup.

The configuration of our heuristic function now has two parameters: the split bound, and the number of refinement iterations. This makes our data field 3-dimensional. Table 3 restricts to the "C" examples – which are the most relevant anyway – to save space. The data are arranged in a slightly unusual way, grouped by example rather than by configuration parameters, to ease observing how the behavior for an example changes as a function of the configuration parameters.

Let us start with some of the simpler observations to be made in Table 3. First, we see that, like for the syntax-based abstractions, without network splitting we don't get far. The overhead needed with "-b all" is huge with 0 or 1 refinement iterations ("-r 0" or "-r 1") already, exhausting the available runtime for C6 respectively C5.[13] With 4 or more refinement iterations, runtime is exhausted in even the smallest example C1. Second, for "-b 1" and "-b 2", the table entry with "-r 7" is always identical to that with "-r 4". This is because ARMC finds feasible error paths. Precisely, with "-b 1" ARMC finds feasible error paths, in all examples and in all parts of the partitionings (meaning, in each single automaton), in 4 refinement iterations. So increasing the maximum number of refinement iterations beyond 4 does not have any effect. With $b = 2$, ARMC finds feasible error paths in 3 refinement iterations already.

Now, consider what happens as we let the configuration parameters vary. Consider first the split bound "-b": compare the data when moving up or down in the table. Like for the syntax-based abstractions in Table 2, the overhead consistently grows over growing split bound, particularly much when the number of refinement iterations is high.[14] The number of explored search states, on the other hand, behaves more stably, and a little more expectedly, than for the syntax-based abstractions in Table 2. In most cases, the number stays the same, or decreases, over increasing split bound. Particularly with many refinement iterations, there is a relatively sharp monotonic decrease over increasing split bound. Notable exceptions to this rule are a few configurations for C1, and the increase from 2.6e+6 states to 3.8e+6 states when moving from "-b 3" to "-b 4" in C6. We observe that the decreased search space size never pays off in runtime: when moving downwards in a column within one part of the table (within one example), the runtime (almost) always increases monotonically.

[12] Since ARMC gets only parts of the network, i.e. due to our splitting operation, a feasible error path here is *not* a solution to the overall problem.

[13] The observations for C6 here show that combining ARMC and UPPAAL enables us to solve examples that neither tool can solve on its own. ARMC, *if* it can solve C6, would definitely run for a *very* long time. UPPAAL, as said before, runs out of memory on this example.

[14] Regarding the huge overhead for C1 with split bound 4, see Footnote 11.

-b \ -r	0	1	4	7
C1				
1	0.8/19778/1.0	0.8/17330/1.0	1.6/2806/1.6	
2	1.9/8769/2.0	2.6/8861/3.1	6.0/1508/6.0	
3	1.7/8769/1.8	3.0/8861/3.1	10.8/1172/10.8	36.2/6362/36.2
4	10.0/16291/10.3	32.9/12044/33.0	176.0/3630/174.3	–/
all	30.2/8769/30.2	120.2/9627/120.2	–/	–/
C2				
1	1.0/62046/1.4	1.2/59031/1.7	1.5/8143/1.6	
2	1.7/39710/2.0	3.1/35245/3.4	6.3/4898/6.3	
3	1.7/39710/2.0	4.1/35245/4.4	12.2/3256/12.2	38.2/25601/38.4
4	2.1/39710/2.6	5.2/35245/5.5	17.2/3256/17.2	63.8/25601/63.9
all	43.0/39710/43.1	189.1/40122/189.8	–/	–/
C3				
1	0.9/88015/1.6	0.9/89194/1.6	1.6/10191/1.7	
2	1.6/67166/2.2	3.1/53616/3.4	6.4/5583/6.4	
3	1.6/67166/2.2	3.8/53616/4.2	12.5/4278/12.6	40.2/30407/40.4
4	2.0/67166/2.5	5.2/53616/5.6	18.0/4278/18.0	67.6/30407/67.8
all	44.6/67166/45.0	198.2/52042/198.9	–/	–/
C4				
1	1.2/897900/6.8	1.5/872580/7.2	2.0/79069/2.7	
2	2.1/516282/5.8	3.0/511180/6.8	6.8/41831/7.1	
3	2.1/516282/5.8	4.8/511180/8.6	14.3/46837/14.8	45.6/279374/47.4
4	2.1/516282/5.8	6.0/511180/9.9	20.6/46837/21.0	73.3/279374/74.9
all	67.6/516282/71.2	288.9/540013/291.7	–/	–/
C5				
1	1.1/9.0e+6/64.1	1.3/8.4e+6/61.0	2.4/1.1e+6/11.8	
2	2.2/4.9e+6/38.8	3.2/6.8e+6/57.0	7.6/425264/11.6	
3	2.2/4.9e+6/38.8	5.2/6.8e+6/59.1	15.4/473470/20.0	48.7/2.3e+6/66.2
4	2.2/4.9e+6/38.8	7.0/6.8e+6/62.2	24.1/288614/26.7	79.9/1.9e+6/94.3
all	94.1/4.9e+6/132.2	–/	–/	–/
C6				
1	/–	/–	2.4/4.4e+6/40.5	
2	/–	/–	7.9/2.8e+6/34.1	
3	/–	/–	16.5/2.6e+6/42.5	/–
4	/–	/–	30.5/3.8e+6/63.4	/–
all	/–	–/	–/	–/

Table 3. Results for abstraction refinement based abstractions. Notation as in Table 1; "-r": number of refinement iterations in ARMC; "-b": split bound; empty entries are identical to their left neighbour; "–/" means time-out during pre-processing; "/–" means out of memory during search.

In terms of runtime, the number of refinement iterations definitely is a better parameter to invest overhead in: most notably, C6 is not solved with less than 4 refinement iterations. Then again, refining "too much" apparently isn't a good idea, either. We get back to this below. First, observe that, when moving from left to right in the table (when increasing the number of refinement iterations), as expected we get a consistent increase of overhead. The number of explored search states consistently (with few exceptions) decreases a little when stepping from "-r 0" to "-r 1", decreases sharply when stepping from "-r 1" to "-r 4", and increases sharply when stepping from "-r 4" to "-r 7". In terms of runtime, the decrease in search space size does not pay off (due to the larger overhead) in C1 .. C4, but does pay off in C5 and C6, when the search spaces explode.

The most curious observation in this data is definitely the *increase* in search space size that we often get when we make the abstraction more refined. Particularly, this is a surprise since, c.f. Section 3.1, every time we refine the abstraction we obtain a heuristic that dominates the previous one. At first sight, we were irritated this is even possible –

a heuristic that dominates another one, but yields a larger search space. Looking more closely, however, this is possible quite naturally. Imagine a state has two successors s and s', of which s leads to an error state on a narrow path (not much branching) of length 10, while s' is the start of a huge part of the state space containing no error state at all. Let's say $h(s) = 5$ and $h(s') = 8$. Let's further say $h'(s) = 9$ and $h'(s') = 8$. Obviously, h' is a more precise heuristic than h – it refined h in its judgement of s – but will yield a much larger search space. The "mistake" made in the refinement step here is that the focus of the refinement is exclusively on s, not on s'. This sort of thing may be precisely what happens (sometimes) when we do a refinement step with ARMC. In fact, the defining characteristic of the refinement step is that it *excludes the detected spurious error path* – which means, one shortest spurious error path (ARMC does a breadth-first search) is removed. But other spurious error paths of the same length may remain. The heuristic values "along" the removed spurious error path will increase (that region of the state space is refined), but the heuristic values "along" the other spurious error paths will remain the same. If the removed spurious error path happens to be the (only) one that actually corresponds to a real solution, then the refinement will increase our search space in pretty much the way as illustrated with s and s' in the example above.

Our intuition was confirmed quite clearly when we ran the following test on example C4 with split bound 3 (middle row of C4 part of Table 3). We incrementally increased the number of refinement iterations, and measured the search space size as well as the length of the shortest error path found in the abstractions at the maximum level. The partitioning for this example in this setting has two parts, giving us two abstractions. The data we obtained are as follows. "-r 0": 516282 nodes, error path lengths 6 and 5. "-r 1": 511180, lengths 9 and 7. "-r 2": 49384, lengths 13 and 10. "-r 3": 56081, lengths 13 and 12. At this point, we first notice the hypothesized effect. In difference to before, the error path length of the first abstraction didn't increase, and promptly the search space size went (slightly) up. The error path in the second abstraction became feasible at this level, so it stays fixed and we don't report it from now on. "-r 4": 46837, length 13. "-r 5": 63606, length 13. "-r 6": 279374, length 13 – this step seems to correspond most closely to the example above. "-r 7": 279374, length 13. "-r 8": 279374, length 13. "-r 9": 361555, length 13. "-r 10": 50060, length 16. In this step, the error path length finally increases again, and promptly the search space size goes sharply down. The error path found in refinement step 10 is feasible, so here our experiment stops.

One can try to overcome this phenomenon by making the refinement mechanism less focussed on a single error path, trying to exclude the spurious error paths more "broadly". The straightforward idea is to introduce predicates so that *all* shortest spurious error paths are removed (not just a single one). This may require too much overhead. It remains to be seen if one can define successful selection heuristics and greedy strategies that remove the most "relevant" error paths, and/or that introduce only the most "relevant" predicates (an idea for the latter may be to define relevance of a predicate based on how many error paths it serves to remove). Another idea might be to use a sort of perimeter search within the abstraction, where the error condition would be "broadened" to the final layer of a depth-bounded backwards breadth-first search. Alternatively, of course, one can use our above observations simply to design an automatic selection of the number of refinement steps: refine until, in an iteration k, the length of the shortest spurious error path does, for the first time, not increase; take the heuristic function defined by the abstract state space from iteration $k-1$.

6 Conclusion

There clearly is promise in defining heuristic functions for model checking based on predicate abstraction. It is straightforward to spell this idea out formally. Apart from the idea, we have contributed a method to efficiently store and query the heuristic information, a method to split an automata network without losing admissibility, and a first empirical exploration. Our (empirical) results are not yet at a level that would be thoroughly satisfying, but we are competitive with the other techniques that have been tried so far. It remains to be seen if, how, and in what sort of applications this kind of heuristic can be made more efficient. We are optimistic that a refinement-based approach will eventually turn out to be quite useful.

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.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI’2001: Programming Language Design and Implementation*, pages 203–213, 2001.

[2] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE’2003: Int. Conf. on Software Engineering*, pages 385–395, 2003.

[3] J. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[4] H. Dierks. Comparing model-checking and logical reasoning for real-time systems. *Formal Aspects of Computing*, 16(2):104–120, 2004.

[5] K. Dräger, B. Finkbeiner, and A. Podelski. Directed model checking with distance-preserving abstractions. In *13th International SPIN Workshop on Model Checking of Software (SPIN’2006)*, 2006.

[6] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *8th International SPIN Workshop on Model Checking of Software (SPIN’2001)*, pages 57–79, 2001.

[7] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 2004.

[8] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV'1997: Computer Aided Verification*, pages 72–83, 1997.

[9] A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *International Symposium on Software Testing and Analysis*, pages 12–21, 2002.

[10] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL'2004: Principles of Programming Languages*, pages 232–244, 2004.

[11] J. Hoffmann and J. Koehler. A new method to query and index sets. In *16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 462–467, 1999.

[12] B. Krieg-Brückner, J. Peleska, E. Olderog, and A. Baer. The UniForM Workbench, a universal development environment for formal methods. In *FM'99 – Formal Methods*, pages 1186–1205, 1999.

[13] S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann. Adapting an AI planning heuristic for directed model checking. In *13th International SPIN Workshop on Model Checking of Software (SPIN'2006)*, 2006.

[14] Andreas Podelski and Andrey Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL'2007: Practical Aspects of Declarative Languages*, pages 245–259, 2007.

[15] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-04)*, pages 497–511, 2004.

[16] A. Rybalchenko. A model checker based on abstraction refinement. Master's thesis, Universität des Saarlandes, 2002.

[17] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI'07: International Conference on Verification, Model Checking and Abstract Interpretation*, 2007.