Relaxation Refinement: A New Method to Generate Heuristic Functions

Jan-Georg Smaus[1] and Jörg Hoffmann[2]

[1] University of Freiburg, Germany, smaus@informatik.uni-freiburg.de
[2] SAP Research, Karlsruhe, Germany, joe.hoffmann@sap.com

Abstract. In artificial intelligence, a *relaxation* of a problem is an overapproximation whose solution in every state of an explicit search provides a heuristic solution distance estimate. The heuristic guides the exploration, potentially shortening the search by exponentially many search states. The big question is how a good relaxation for the problem at hand should be derived. In model checking, overapproximations are called *abstractions*, and *abstraction refinement* is a powerful method developed to derive approximations that are sufficiently precise for *verifying* the system at hand. In our work, we bring these two paradigms together. We pioneer the application of (predicate) abstraction refinement for the generation of heuristic functions that are intelligently adapted to the problem at hand. We investigate how an abstraction refinement process for generating heuristic functions should differ from the process used in the verification context. We do so in the context of DMC of timed automata. We obtain a variety of interesting insights about this approach.
Keywords: Directed model checking, abstraction refinement, predicate abstraction, timed automata

1 Introduction

In artificial intelligence (AI), a *relaxation* of a problem is an overapproximation of the problem. During an explicit search in the state space of the problem, in each state s the relaxed problem is solved starting from s, and the length of the relaxed solution is used as a heuristic distance estimate, i.e., an estimate of the distance from s to the nearest solution state. States with lower estimated distance are explored first. It is well known that this strategy can exponentially decrease the explored part of the state space (e.g. [22]). Recently, the same idea has been applied for *falsification*, namely in *directed model checking* (*DMC*) [11], which is the search for errors using a heuristic function.

Both in AI and in DMC, the main question to be addressed is how to define the approximation that underlies the heuristic function. Different definitions yield different heuristics, and this makes all the difference between being and not being able to find an error. Intuitively, the heuristic function should capture sufficient information for predicting whether a state is likely to lead to an error. Doing so requires knowledge about the reasons for the error in the particular system considered. The "hot trail" we follow is to use methods coming from the field of model checking to learn this kind of knowledge.

In model checking, overapproximations are called *abstractions*, and have been used for *verification.* Applying the formalism of *abstract interpretation* [7], an abstraction of a system is designed so that if some undesirable state is unreachable in the abstraction, then it is surely unreachable in the original system. A highly successful method in this context is *abstraction refinement* [2, 6, 23]: starting from some trivial abstraction, the abstract state space is computed. If there is no path to the undesirable state (error path), the system is safe and one can stop. Otherwise, one determines if the path is *spurious*, i.e., if it does not correspond to a path in the original system. If it is spurious, one examines the path and tries to refine the abstraction in order to exclude this spurious error path in the next iteration. If the path is not spurious, an error has been found. The process of repeated refinement iterations is called *refinement loop.*

Given these facts, the obvious question that springs to mind is: can, and how can, we use abstraction refinement to devise heuristic functions that are intelligently adapted to the system at hand? This question may be of relevance in many different contexts, ranging from DMC in various model checking formalisms to the various search problems traditionally considered in AI. Here, we investigate this question in the context of DMC of timed automata.

In our own previous work [16], we have shown how one can use predicate abstraction [14] to generate heuristic functions for DMC of timed automata using UPPAAL [4]. Following the *pattern database* approach [8], the abstract state space is built before the actual search for an error starts; during the search, the abstract state space is used as a look-up table for retrieving the heuristic values. The abstraction predicates in that work were mostly generated by reading the predicates directly off transition guards in the system. We also made an initial experiment with abstraction refinement, realised via the *abstraction refinement model checker* (*ARMC*) [23]. The conclusion from the latter experiment was that off-the-shelf abstraction refinement is *not* a good method for deriving heuristic functions. An intuitive explanation is that in off-the-shelf abstraction refinement, the abstraction is tailored to capture a lot of information about one particular error path; other regions of the state space unrelated to that path are abstracted coarsely. When used as a heuristic function, such an abstraction yields precise estimates for states near the error path, but imprecise – overly optimistic – estimates otherwise. This can have the unwanted effect that states unrelated to the error path obtain lower heuristic estimates, and are preferred in the search. In addition, using off-the-shelf abstraction refinement, we have insufficient control to be able to tune the balance between heuristic quality and heuristic cost.

In this paper, while being inspired by ARMC, we do not use ARMC off-the-shelf, because we want to experiment with different methods of doing abstraction refinement in the context of heuristic generation, to find a method is more suitable for defining useful heuristic functions. Let us briefly explain the strategies we tried. The error state in our benchmarks is given by a formula $(loc(p_1) = \ell_1) \wedge (loc(p_2) = \ell_2)$, stating that process p_1 is in location ℓ_1 and p_2

is ℓ_2. The simplest method we tried is to use an initial abstraction consisting of the predicates $loc(p_1) = \ell_1$, $loc(p_2) = \ell_2$, so that the abstraction could always distinguish an error state from a non-error state. Alternatively, we generated two abstractions based on initial predicates $loc(p_1) = \ell_1$ and $loc(p_2) = \ell_2$ kept separate. Somewhat surprisingly, the latter turned out to be better, and can compete with other heuristics we use for comparison. To overcome the focus of the abstraction refinement on one particular path, we then tried to base each refinement step on *several paths*, rather than just a single path. The most surprising result for us was that this had no positive impact. We learnt that the way predicates are added based on an abstract path sometimes implies that the refinement loop terminates with an extremely coarse abstraction, and yet there is no "incentive" to refine this abstraction. Moreover, we learnt that this is not a question of choosing the right paths. To tackle this problem, we used initial abstractions based on "random" predicates as additional "seed" for the abstraction refinement. The results are generally unstable; sometimes they are extremely good.

Our overall experiments suggest that abstraction refinement is useful for *selecting* predicates from a certain repertoire so that the set obtained is informative enough yet small enough not to pose a prohibitive computational overhead; in this respect the present approach is an advance over [16]. We had also hoped that abstraction refinement can, compared to other ways of generating predicates, significantly enhance the repertoire itself; but, at least in our benchmarks, this does not seem to be the case. It remains to be seen if and how the situation changes in the context of DMC in other formalisms (or even just other benchmarks), and in the context of AI search problems. As such, our work provides only a first exploration of a much larger research topic.

This paper is organised as follows. In the next section, we introduce timed automata. In Sec. 3, we explain predicate abstraction for heuristic generation. In Sec. 4, we explain how predicates are generated by refinement based on an error path in ARMC, and how this approach is adapted for generating heuristic functions. In Sec. 5 we report on experiments, in Sec. 6 we discuss related work, and in Sec. 7 we conclude.

2 Timed Automata in Uppaal

We introduce timed automata [1] here following the terminology of Uppaal [4], which is an integrated tool environment for modelling, simulation and verification of real-time systems. We restrict to the features that we actually consider in our benchmarks and implementation.

A **timed automaton** is a finite (ω-)automaton enhanced with real-valued variables called **clocks** and integer variables. Instead of the usual word *state* we speak of a **location** of a timed automaton, since a **state** in our parlance consists of a location together with a value for each variable of the automaton. One location is marked as **initial**.

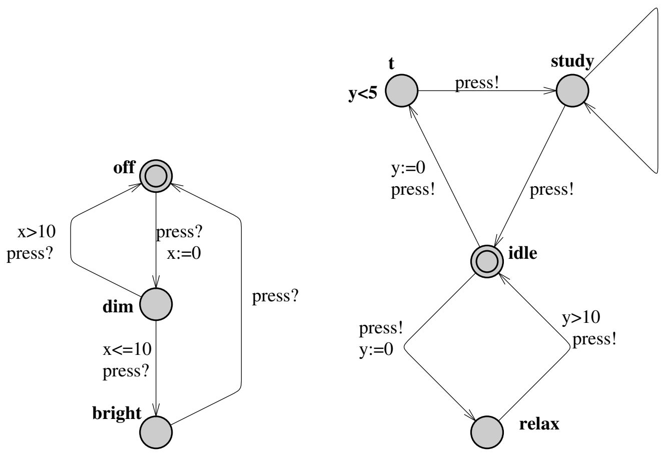


Fig. 1. A system composed of two processes

Locations can be connected by directed **edges**. An edge can be labelled with a **clock** (resp. **integer**) **guard**, which is a conjunction of conditions of the form $x \bowtie c$ or $x - y \bowtie c$ (resp. $c \bowtie 0$), where x, y are clocks and c is an expression using natural constants and integer variables, and $\bowtie \in \{<, \leq, =, \geq, >\}$. An edge can also be labelled with one or several **effects**, which are assignments to an integer variable or resets of a clock variable to 0. A location can be labelled with an **invariant**, which is of the same form as a guard and states a condition that must hold while the automaton remains in this location. One usually requires that clock invariants are downwards closed, i.e., $\bowtie \in \{<, \leq\}$.

An automaton as described so far is called a **process**; several processes can be composed to a network of automata, called **system**, as follows: a **state** is characterised by a location for each of its processes and a value for each of its variables. An edge may be labelled by a synchronisation label $ch?$ or $ch!$ where ch is a symbol called **channel**. If one process has an edge labelled $ch?$ and another process has an edge labelled $ch!$, then the two must be taken simultaneously to obtain a *transition*. An edge of a process that has no synchronisation label can be taken alone; this is called a τ-*transition.*

Figure 1 shows an example. The process on the left-hand side models a lamp switch and the process on the right-hand side models the lamp user. We are dealing here with a widespread design of lamps where pressing the button twice "quickly" causes the light to be bright, whereas pressing it once causes the light to be dimmed if it was off before, and switches the light off otherwise. We illustrate some of the above concepts: x and y are clocks, **off** is a location (marked as initial by the double circle), x > 10 is a guard, x := 0 is an effect, $\mathbf{y < 5}$ is an invariant of location **t**, and 'press' is a channel.

A **run** or **path** is defined in an intuitive way (see [1, 4, 5] for a formal definition): the system starts in its initial state, some time passes (during which all clocks increase at the same speed, and all relevant location invariants must hold), then a transition allowed by the guards is made (which takes no time) and its effects are applied, reaching a new state, and so forth. Here is a run for the lamp example (**L** and **U** stand for the "lamp" and "user" process, respectively):

$$(\mathbf{L}.\mathbf{off}, \mathbf{U}.\mathbf{idle}, x = 0, y = 0) \rightarrow (\mathbf{L}.\mathbf{off}, \mathbf{U}.\mathbf{idle}, x = 31, y = 31) \xrightarrow{\text{press}}$$
$$(\mathbf{L}.\mathbf{dim}, \mathbf{U}.\mathbf{t}, x = 0, y = 0) \rightarrow (\mathbf{L}.\mathbf{dim}, \mathbf{U}.\mathbf{t}, x = 2, y = 2) \xrightarrow{\text{press}}$$
$$(\mathbf{L}.\mathbf{bright}, \mathbf{U}.\mathbf{study}, x = 2, y = 2)$$

The **state space** $\mathcal{S}$ is the directed graph where the nodes are all states, and the edges are the transitions. The set of **error** or **target** states which are undesirable are given by a formula ϕ, specifying a condition on the locations and variables.

3 Predicate Abstraction for Heuristic Generation

In this section we recall our previous work [16]; we explain technically how heuristics for Uppaal are computed based on predicate abstraction.

A **predicate** is a logic formula that "talks" about the system. We consider three kinds of predicates: **location** predicates $loc(proc) = \ell$ stating that process *proc* is in location ℓ; and **integer**, resp. **clock** predicates, defined like guards, see Sec. 2.

A vector $\mathcal{P} = (p_1, \ldots, p_m)$ of predicates defines a **predicate abstraction**; e.g. $\mathcal{P} = (loc(1) = 3,\ x > 2,\ y = 0)$. We sometimes regard $\mathcal{P}$ as a *set.* When $|\mathcal{P}| = m$, we call a vector in $\{\mathsf{T}, \mathsf{U}, \mathsf{F}\}^m$ an **abstract state** or **TUF-vector** for $\mathcal{P}$. Here, $\mathsf{T}, \mathsf{U}, \mathsf{F}$ stand for *true*, *unknown*, and *false.* We usually assume that $\mathcal{P}$ is clear from the context.

Viewing a state s as a valuation of the system variables, we use the notation $s \models \psi$ to say that the formula ψ is true under s, and $\phi \models \psi$ means that ψ is true under every valuation under which ϕ is true.

Given a concrete state s, i.e., a vector consisting of a location for each process and a value for each variable, the **abstraction** of s is a vector $\mathbf{b} = (b_1, \ldots, b_m) \in \{\mathsf{T}, \mathsf{F}\}^m$ where $b_i = \mathsf{T}$ if $s \models p_i$ and $b_i = \mathsf{F}$ if $s \models \neg p_i$. E.g. if $s = (loc(1) = 2, loc(2) = 4, x = 2.3, y = 4.7)$ and $\mathcal{P}$ is as above, then the abstraction of s is FTF.

For an abstract state $\mathbf{b}$, we denote by $[\mathbf{b}]$ the **concretisation** of $\mathbf{b}$, i.e., $[\mathbf{b}] = \{s \mid \text{for all } i \in [1..m], s \models p_i \text{ if } b_i = \mathsf{T}, s \models \neg p_i \text{ if } b_i = \mathsf{F}\}$. For $\mathbf{b}, \mathbf{b}' \in \{\mathsf{T}, \mathsf{U}, \mathsf{F}\}^m$, we say that $\mathbf{b}'$ **subsumes** $\mathbf{b}$ if $[\mathbf{b}'] \supseteq [\mathbf{b}]$, which is the case iff $\mathbf{b}'$ is obtained from $\mathbf{b}$ by replacing zero or more occurrences of T or F by U. The **abstract state space** for $\mathcal{P}$, denoted $[\mathcal{S}]^{\mathcal{P}}$, is the directed graph where the nodes are all TUF-vectors for $\mathcal{P}$, and there is an edge from $\mathbf{b}_1$ to $\mathbf{b}_2$ iff there exist $s_1 \in [\mathbf{b}_1]$ and $s_2 \in [\mathbf{b}_2]$ so that there is an edge from s_1 to s_2 in $\mathcal{S}$ (see Sec. 2). Note that $[\mathcal{S}]^{\mathcal{P}}$ is an *over*-approximation of $\mathcal{S}$, i.e., every concrete path corresponds to an abstract path but not necessarily vice versa. This can be seen as follows: suppose we have four concrete states s_1, s_2, s_2', s_3 such that $s_1 \rightarrow s_2$ and $s_2' \rightarrow s_3$ hold in $\mathcal{S}$ but $s_1 \rightarrow s_2'$ and $s_2 \rightarrow s_3$ do not hold in $\mathcal{S}$, and $[s_2] = [s_2']$; then we have $[s_1] \rightarrow [s_2] \rightarrow [s_3]$ in $[\mathcal{S}]^{\mathcal{P}}$ although s_3 might not be reachable from s_1 is $\mathcal{S}$.

Given an error condition ϕ as in Sec. 2, the **abstract error state** is defined as the TUF-vector $\mathbf{b} = (b_1, \ldots, b_m)$ where $b_i = \mathsf{T}$ if $\phi \models p_i$, and $b_i = \mathsf{F}$ if $\phi \models \neg p_i$, and $b_i = \mathsf{U}$ otherwise. Note that ϕ might not uniquely determine the value of each predicate p_i. In addition, we obtain a precision loss due to the fact that we consider the predicates in isolation. E.g., it might be the case that $\phi \models (p_1 \wedge p_2 \wedge p_3) \vee (p_1 \wedge \neg p_2 \wedge \neg p_3)$, so that we should have *two* abstract error states TTT and TFF, but instead, we just take *the* abstract error state TUU. This merging is referred to as *Cartesian abstraction* [3].

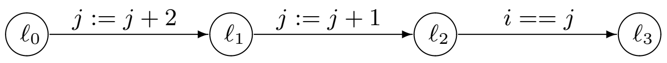
The abstract state space is computed starting from the abstract error state and computing its abstract predecessors to obtain the first *layer*; then all predecessors of those states give the second layer, and so forth. Whenever a state is computed that is subsumed by a state computed earlier, then the new state will be ignored. This computation is called *regression*.

Computing the predecessor of an abstract state $\mathbf{b}$ is a logical deduction task. To compute the predecessor of $\mathbf{b}$ obtained by applying (backward) some edge (pair) t, we construct a formula using the variables occurring in $\mathcal{P}$ and in the guards, effects, and invariants of (the sources and targets of) t, where we prime ($'$) the variables to denote their value in the target location; the unprimed variants correspond to the source location. First, $\mathbf{b}$ translates into $\psi_{\mathbf{b}} = \bigwedge_{b_i=\mathsf{T}} p_i' \wedge \bigwedge_{b_i=\mathsf{F}} \neg p_i'$, where p_i' denotes the result of priming each variable in p_i. Secondly, t translates into a formula ψ_t that expresses the effect of applying t, by relating the values of the variables in the sources and targets. E.g. if t has an effect $i := i + 1$ then one of the conjuncts of ψ_t will be $i' = i + 1$.

The exact method of determining all the abstract predecessor states of $\mathbf{b}$ would be as follows: we enumerate all abstract states $\mathbf{a} \in \{\mathsf{T}, \mathsf{F}\}^m$, build a formula $\psi_{\mathbf{a}} = \bigwedge_{a_i=\mathsf{T}} p_i \wedge \bigwedge_{a_i=\mathsf{F}} \neg p_i$ in analogy to $\psi_{\mathbf{b}}$ but with unprimed variables, and check if $\psi_{\mathbf{b}} \wedge \psi_t \wedge \psi_{\mathbf{a}}$ is satisfiable. If yes, then there is an assignment to the variables, corresponding to concrete states in $[\mathbf{a}]$ and $[\mathbf{b}]$, respectively, with a transition between these concrete states, and hence $\mathbf{a}$ is an abstract predecessor state of $\mathbf{b}$.

However, the method described in the previous paragraph would be too inefficient because we would have to enumerate 2^m abstract states. Instead, we again apply Cartesian abstraction: we compute just *one* abstract predecessor state $\mathbf{a}$ of $\mathbf{b}$ w.r.t. t by taking each predicate p_i *in isolation*: if p_i is implied by $\psi_{\mathbf{b}} \wedge \psi_t$, then $a_i = \mathsf{T}$; if $\neg p_i$ is implied by $\psi_{\mathbf{b}} \wedge \psi_t$, then $a_i = \mathsf{F}$; otherwise $a_i = \mathsf{U}$. For checking the implications we use the solver ICS [13].

Since we compute the abstract state space $[\mathcal{S}]^{\mathcal{P}}$ backwards starting from the error state, we obtain the distance of each abstract state from the abstract error state for free. This information is stored in a so-called *pattern database*, which is implemented as a certain tree data structure [15] supporting efficient subsumption tests. The pattern database is the basis for defining the heuristic value of a concrete UPPAAL search state s. Essentially, the lookup of a heuristic



$$\begin{tikzcd} \ell_0 \arrow[r, "j := j+2"] & \ell_1 \arrow[r, "j := j+1"] & \ell_2 \arrow[r, "i == j"] & \ell_3 \end{tikzcd}$$

Fig. 2. Predicate Generation in ARMC

value for a concrete UPPAAL search state s works as follows: the abstraction of s is computed; call it $\mathbf{b}$; now consider all $\mathbf{c}$ in the pattern database that subsume $\mathbf{b}$; among those, take the one whose (precomputed) distance to the abstract error state is minimal; this distance is the heuristic value of s (see [16] for more details).

We (and others before us [12, 17]) have experienced that combining several simple abstractions is often better than having one complicated abstraction. Thus we usually have several sets of abstraction predicates and thus abstract spaces and "raw" heuristic functions. The latter can be combined by taking, for each concrete search state, the maximum or the sum.[3] We refer to this as *MAX option*, *SUM option*, resp.

4 Abstraction Refinement

When predicate abstraction refinement is used for verification, one successively refines an abstraction by adding predicates in order to exclude spurious error paths. But which predicates? We explain here how the *abstraction refinement model checker* (*ARMC*) [23] answers this question, since we generate predicates in the same way. We immediately jump to a timed automata setting. So the aim is to verify a timed automaton, i.e. to show that a certain error location ℓ_{err} is unreachable.

A *predicate* is of the form $e_1 \bowtie e_2$ where e_1, e_2 are expressions involving integer variables and $\bowtie \in \{=, \leq, <, \geq, >\}$. Each location ℓ is associated with a set of predicates $\mathcal{P}_\ell$, and an abstract state is a pair $(\ell, \mathbf{a})$ where $\mathbf{a} \in \{\mathsf{T}, \mathsf{U}\}^{|\mathcal{P}_\ell|}$. Note that in the verification context [23], one only distinguishes *true* and *unknown*.

The abstraction refinement loop starts with an initial abstraction where $\mathcal{P}_\ell = \emptyset$ for all ℓ. We now explain how one iteration of the loop works. Assume that we have a current abstraction, i.e., a $\mathcal{P}_\ell$ for each ℓ. We compute an abstract error path $\ell_0 \to \ldots \to \ell_n = \ell_{\mathsf{err}}$, i.e. a path from the initial location to ℓ_{err}. By treating $\ell_0 \to \ldots \to \ell_n$ backward, we generate new predicate sets $\mathcal{P}'_{\ell_0}, \ldots, \mathcal{P}'_{\ell_n}$ as follows: $\mathcal{P}'_{\ell_n} := \mathcal{P}_{\ell_n}$; if $\mathcal{P}'_{\ell_{i+1}}$ is already computed, and $\ell_i \to \ell_{i+1}$ has guard $g_1 \wedge \ldots \wedge g_r$ and assignments $x_1 := e_1, \ldots, x_k := e_k$, then $\mathcal{P}_{\ell_i} = (\{g_1, \ldots, g_r\} \cup \mathcal{P}'_{\ell_{i+1}})[x_k \mapsto e_k, \ldots, x_1 \mapsto e_1] \cup \mathcal{P}_{\ell_i}$ (the notation $[x \mapsto e]$ denotes the replacement of x by e).

In Fig. 2 we illustrate a fragment of an error path. Here, $i == j$ is a guard. Starting with $\mathcal{P}_{\ell_i} = \emptyset$ for $i \in [0..3]$ we obtain $\mathcal{P}'_{\ell_3} = \emptyset$, $\mathcal{P}'_{\ell_2} = \{i = j\}$, $\mathcal{P}'_{\ell_1} = \{i = j+1\}$ and $\mathcal{P}'_{\ell_0} = \{i = j+3\}$. Each $\mathcal{P}'_\ell$ corresponds to the *weakest precondition* of $i = j$ w.r.t. the subsequent updates, i.e. it expresses exactly the condition under

[3] The issue of *admissibility* is discussed in [16].

which $\ell_2 \to \ell_3$ can be taken. If ℓ_0 is the initial location of the automaton and we assume that variables are initialised to 0, then $\mathcal{P}'_{\ell_0}, \mathcal{P}'_{\ell_1}, \mathcal{P}'_{\ell_2}, \mathcal{P}'_{\ell_3}$ are sufficient to exclude this spurious error path (fragment), i.e., this error path will not be computed again.

We adopt the method of generating predicates just described. Yet there are some differences between ARMC and our setting. Unlike ARMC, our abstract states represent location information only to the extent that there are location predicates. Therefore, our abstract paths can be spurious for location reasons, i.e., we could have a transition $\ell_0 \to \ell_1$ followed by a transition $\ell'_1 \to \ell_2$ where $\ell_1 \neq \ell'_1$, because the abstraction cannot distinguish ℓ_1 from ℓ'_1. One could of course change this, but our experiments suggest that abstractions preserving all location information lead to abstract state spaces that are much too expensive to compute. Therefore, we do not have a different predicate set *per location*, but an abstraction is given by one global predicate set.

More importantly, as explained in the introduction, our aim is not to exclude all spurious error paths (verification), but to characterise a sufficiently large environment of the error state with sufficient precision to provide a good heuristic.[4] Therefore, instead of basing one refinement step on one path from the initial to the error state, we can base one refinement step on *arbitrarily many* paths starting from an *arbitrary state* and leading to the error state.

We present the core of our abstraction refinement algorithm: the generation of new predicates based on a single abstract path. Given a set of predicates $\mathcal{P}$ and an abstract path $t_1, \ldots, t_n$ leading to the abstract error state, we refine $\mathcal{P}$ as shown in Fig. 3. To simplify the presentation, we only show the case of a synchronised transition. We denote by $\mathit{edge}!(t)$, $\mathit{edge}?(t)$ the two edges of t, and by $\mathit{targ}(d), \mathit{src}(d)$ the target and source locations, and by $\mathit{proc}(d)$ the process of an edge d.

The algorithm processes the edge(pair)s of the abstract path backward. The most relevant difference to ARMC is that the abstract path may be spurious for location reasons. We maintain information about the current location of each process in *curloc*. Whenever $\mathit{targ}(\mathit{edge}!(t_i))$ is different from $\mathit{curloc}(\mathit{proc}(\mathit{edge}!(t_i)))$ (or analogously for *edge*?), we have detected that $t_1, \ldots, t_n$ is spurious for location reasons. We then add a location predicate that will exclude the abstract path in the next iteration, and stop the processing of the edge(pair)s (see lines marked "[").

The algorithm must be embedded in one or several refinement loops, to generate one or several abstractions (see end of Sec. 3). Now there are three questions:

- What should the initial abstraction(s) be?
- How many and which abstract paths should be chosen for the refinement?
- When should the refinement loop stop?

We have tried many possible approaches to answering these questions, of which we present some in Sections 5.2 to 5.4, after explaining the setup for our experiments.

[4] Whether or not a heuristic is good is measured here, as usual, by considering the size of the explored state space (see Sec. 5).

```
procedure $refine\_path$(Predicates $\mathcal{P}$, Path $(t_1, \ldots, t_n)$)
  foreach process $p$ do
    $curloc(p) := unknown$ od
  for $i := n$ to 1 do
  ⎡ if $curloc(proc(edge!(t_i))) \neq unknown \wedge targ(edge!(t_i)) \neq curloc(proc(edge!(t_i)))$
  ⎢   $\mathcal{P} := \mathcal{P} \cup \{loc(proc(edge!(t_i))) = curloc(proc(edge!(t_i)))\}$ break fi
  ⎣ $curloc(proc(edge!(t_i))) := src(edge!(t_i))$
  ⎡ if $curloc(proc(edge?(t_i))) \neq unknown \wedge targ(edge?(t_i)) \neq curloc(proc(edge?(t_i)))$
  ⎢   $\mathcal{P} := \mathcal{P} \cup \{loc(proc(edge?(t_i))) = curloc(proc(edge?(t_i)))\}$ break fi
  ⎣ $curloc(proc(edge?(t_i))) := src(edge?(t_i))$
    foreach invariant $g$ of $targ(edge!(t_i)), targ(edge?(t_i))$ do
      $\mathcal{P} := \mathcal{P} \cup \{g\}$ od
    foreach assignment $i := e$ of $edge!(t_i), edge?(t_i)$ in reverse order do
      $\mathcal{P} := \mathcal{P}[i \mapsto e] \cup \mathcal{P}$ od
    foreach guard $g$ of $edge!(t_i), edge?(t_i)$ and
    foreach invariant $g$ of $src(edge!(t_i)), src(edge?(t_i))$ do
      $\mathcal{P} := \mathcal{P} \cup \{g\}$ od
  od
  return $\mathcal{P}$
```

Fig. 3. Refining an abstraction based on a path $t_1, \ldots, t_n$

5 Experiments

Our benchmarks come from two industrial case studies [9, 18]. Since we are dealing with error detection, i.e., falsification, our examples had an error injected.

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 [9]. An error was injected by increasing an upper time bound. Examples "Ci", $i = 2, \ldots, 9$, are a case study called "Single-tracked Line Segment" coming from an industrial project partner of the UniForM-project [18]. The problem is to design a distributed real-time controller for a segment of tracks where trams share a piece of track. The controller was modelled in terms of PLC-automata (*PLC* stands for *programmable logic controllers*) [9, 18] 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 eight models with decreasing size by abstracting more and more of these inputs. The numbering of the benchmarks is ad hoc, but the benchmarks become bigger with increasing i.

Concerning "Ci", we observed that the model checking could be dramatically simplified by a slight modification of the benchmarks, namely to reduce the number of times a certain loop edge can be taken. This issue is completely orthogonal to the topic of this paper, but we mention the fact that we used the modified benchmarks for the sake of comparison with other works [10, 16, 19, 20].

We performed DMC using UPPAAL. We used *greedy best-first search* [20], i.e., the search queue is a priority queue over the value of the heuristic function. The

Table 1. M1-M4, N1-N4: search space size | heuristic computing time | user time

	M1			M2			M3			M4		
h^L	5656	n.a.	**0.1**	30743	n.a.	**0.3**	**18431**	n.a.	**0.2**	122973	n.a.	1.3
h^U	*14679*	n.a.	**0.2**	*68407*	n.a.	0.9	75976	n.a.	0.9	*233378*	n.a.	2.8
syn 5000/10/SUM	*23257*	0.3	0.6	*84475*	0.4	1.5	92548	0.4	1.6	*311049*	0.6	4.7
syn 5000/10/MAX	*23744*	0.3	0.6	*102042*	0.4	1.7	98715	0.4	1.7	*399114*	0.6	5.6
syn 20000/20/SM	*12780*	1.7	*1.7*	34947	10.7	*10.7*	55098	10.8	10.8	139875	15.1	*15.1*
1abs 5000/10	*20188*	0.7	0.8	39369	0.3	0.7	64522	0.3	1.0	110240	0.3	1.6
2abs 5000/10/SUM	*14955*	0.7	0.7	17753	0.5	**0.6**	86316	0.4	1.4	110240	0.3	1.5
2abs 5000/10/MAX	3769	0.6	0.6	**7637**	0.5	**0.5**	119108	0.4	1.7	**32034**	0.3	**0.5**
seed 100/10/SUM	*18566*	0.6	0.7	29253	0.5	0.8	64671	0.5	1.3	110240	0.5	1.8
seed 100/10/MAX	**1050**	0.5	0.5	**3921**	0.6	**0.6**	*5514784*	0.5	*97.6*	**32034**	0.5	**0.8**
seed 5000/10/SUM	*14955*	1.7	*1.7*	17753	1.3	1.3	86316	1.0	2.0	110240	0.7	2.0
seed 5000/10/MAX	3769	1.7	*1.7*	**7637**	1.3	1.3	119096	1.0	2.4	**32034**	0.7	**0.9**
	N1			N2			N3			N4		
h^L	16335	n.a.	**0.5**	*88537*	n.a.	2.5	**28889**	n.a.	**0.6**	226698	n.a.	**5.0**
h^U	*25577*	n.a.	**0.8**	*132711*	n.a.	3.9	*143969*	n.a.	4.2	*747210*	n.a.	20.0
syn 5000/10/SUM	*36030*	0.3	*1.8*	*178333*	0.5	*6.9*	*196535*	0.4	7.7	*983344*	0.6	*37.4*
syn 5000/10/MAX	*31589*	0.3	*1.3*	*163001*	0.4	5.6	*207665*	0.5	7.5	*1255213*	0.6	*47.1*
syn 20000/20/SM	17357	1.7	*2.1*	63596	10.8	*12.0*	96202	10.6	*12.4*	445359	15.0	*29.0*
1abs 5000/10	*31042*	0.7	*1.8*	*91367*	0.3	3.2	*135906*	0.3	4.9	353609	0.3	12.5
2abs 5000/10/SUM	17584	0.7	*1.2*	38216	0.5	**1.6**	*157491*	0.4	5.5	353609	0.3	12.4
2abs 5000/10/MAX	4031	0.7	**0.7**	**18914**	0.5	**1.0**	*242012*	0.4	*8.7*	**119802**	0.3	**3.9**
seed 100/10/SUM	*27364*	0.6	*1.4*	71411	0.6	2.9	*132538*	0.5	5.0	353609	0.5	13.0
seed 100/10/MAX	**1209**	0.5	**0.5**	**15920**	0.6	**1.0**	117276	0.6	4.5	**119802**	0.5	**4.3**
seed 5000/10/SUM	17584	1.7	*2.0*	38216	1.3	2.3	*157491*	1.0	*6.2*	353609	0.7	13.0
seed 5000/10/MAX	4031	1.6	*1.6*	**18914**	1.3	**1.7**	*241968*	1.0	*9.3*	**119802**	0.7	**4.4**

interface to UPPAAL is the one introduced in [19].[5] For the experiments, we used a machine with two Intel Xeon processors running at 3.02 GHz with 6 GB RAM.

We present results for six kinds of heuristics here. First, we have heuristics h^L and h^U, which are based on a heuristic method from AI planning [20]. Second, we have heuristic "syn" based on extracting abstraction predicates from the guards of a system [16]. Third, we have a heuristic based on refinement of a single error path with initial predicate set $\{loc(p_1) = \ell_1, loc(p_2) = \ell_2\}$ ("1abs"), and fourth, a heuristic based on two abstractions obtained by having two initial predicate sets $\{loc(p_1) = \ell_1\}$, $\{loc(p_2) = \ell_2\}$ ("2abs"). Fifth, we have a method where refinement is based on several abstract paths (Table 3), and finally, a method where we use initial predicate sets that contain "random" predicates as additional "seed" for the abstraction refinement.

The entries in Tables 1 and 2 are of the form $a|b|c$. Here, a is the size of the search space explored by UPPAAL, i.e., the number of explored states (not to be confused with the size of the abstract state space!). The second figure b is the total time in seconds taken for precomputing the heuristic, and c is the total

[5] The UPPAAL source code was provided to our group by Gerd Behrmann.

Table 2. C2-C9: search space size | heuristic computing time | user time

	C2			C3			C4			C5		
h^L	**4059**	n.a.	**0.2**	3253	n.a.	**0.2**	**2683**	n.a.	**0.2**	87342	n.a.	*6.5*
h^U	5629	n.a.	**0.4**	4756	n.a.	**0.3**	3471	n.a.	**0.2**	19598	n.a.	1.2
syn 5000/10/SUM	**2866**	3.3	3.3	**2691**	3.8	3.8	3941	5.3	5.3	*68077*	5.9	*6.0*
syn 5000/10/MAX	**2421**	3.7	3.7	**2360**	3.8	3.8	**1621**	5.3	5.3	518	5.9	*5.9*
syn 20000/20/SUM	9938	8.5	*8.5*	5446	9.2	*9.2*	*11061*	13.4	*13.4*	timeout		
syn 20000/20/MAX	*18085*	8.2	*8.2*	9506	8.8	*8.8*	*21712*	13.0	*13.0*	timeout		
1abs 5000/10	*28303*	1.3	1.3	*13458*	1.4	1.4	*11836*	1.6	1.6	**378**	2.0	2.0
2abs 5000/10/SUM	7382	1.4	1.4	**2866**	1.5	1.5	**2679**	1.8	1.8	**258**	2.0	2.0
2abs 5000/10/MAX	5616	1.4	1.4	**3612**	1.4	1.4	3376	1.7	1.7	**299**	2.0	2.0
seed 100/10/SUM	5154	0.5	0.5	4793	0.6	0.6	4002	0.8	0.8	657	0.5	**0.5**
seed 100/10/MAX	7011	0.6	0.6	**4196**	0.6	0.6	3512	0.8	0.8	797	0.5	**0.5**
seed 5000/10/SUM	6849	5.7	*5.7*	**3496**	5.9	*5.9*	**3140**	6.6	6.6	583	7.8	*7.8*
seed 5000/10/MAX	5612	6.0	*6.0*	**3608**	6.0	*6.0*	3374	6.7	6.7	**304**	7.8	*7.8*

	C6			C7			C8			C9		
h^L	*16284*	n.a.	1.3	*79769*	n.a.	6.2	37202	n.a.	2.6	134489	n.a.	**9.1**
h^U	9327	n.a.	**0.5**	46193	n.a.	**2.6**	**6569**	n.a.	**0.5**	127924	n.a.	11.0
syn 5000/10/SUM	*13518*	6.7	6.7	*71916*	7.1	*7.1*	61871	10.4	*10.4*	156445	14.5	15.0
syn 5000/10/MAX	**1514**	6.3	6.3	17534	7.6	7.6	39158	11.2	*11.2*	146810	15.8	16.7
syn 20000/20/SUM	6176	14.1	*14.1*	32290	15.4	*15.4*	30394	18.3	*18.3*	**31280**	25.8	*25.8*
syn 20000/20/MAX	**863**	13.8	*13.8*	**9603**	16.4	*16.4*	19682	19.9	*19.9*	**22176**	27.9	*27.9*
1abs 5000/10	*16346*	2.0	2.0	*102611*	2.1	**2.7**	*329298*	2.2	5.2	*1637759*	1.9	*19.1*
2abs 5000/10/SUM	3127	2.2	2.2	31584	2.3	**2.4**	106065	2.8	3.9	451610	3.2	8.8
2abs 5000/10/MAX	3549	2.1	2.1	31676	2.3	**2.3**	103416	2.8	3.6	359530	3.2	**7.4**
seed 100/10/SUM	*18711*	0.6	**0.8**	*110878*	0.7	**1.9**	*333488*	0.6	4.5	*1623070*	0.7	*20.0*
seed 100/10/MAX	*20583*	0.6	**0.8**	*119415*	0.7	**2.0**	*340927*	0.7	4.6	*1637147*	0.7	*20.3*
seed 5000/10/SUM	*20250*	8.0	*8.0*	*118830*	8.1	*8.1*	*290987*	10.0	*11.8*	*1270802*	12.0	*30.0*
seed 5000/10/MAX	3539	7.6	7.6	31610	7.9	7.9	103140	9.3	9.3	358171	11.3	**14.7**

user time for the UPPAAL model checking as measured by the Linux command `time`, including the precomputation time b. Note that in some cases b is equal to c because the time for the actual Uppaal model checking is negligible. We have highlighted the best and the worst figures. For each example, those figures within factor 2 of the best figure are in **boldface**, and those within factor 2 of the worst are in *italics*.

5.1 Other Heuristics Used for Comparison

On the one hand, we used two heuristics introduced in [20], which are among the best heuristics for these benchmarks [19]. There, a heuristic method from AI planning is adapted, based on a notion of "monotonicity" where it is assumed that a state variable accumulates, rather than changes, its values. In Tables 1 and 2 and in [20], there are two heuristics referred to as h^L (which states in which layer of the *relaxed transition graph* a state can be found, and is a strongly underestimating heuristic) and h^U (which gives an actual path in the abstract system, and may be overestimating). Since h^L and h^U are computed on-the-fly, there is no heuristic precomputation.

Concerning the runtime, h^L and h^U are very often among the best by far. Concerning the size of the explored state space, they are more often among the worst than among the best. The heuristics are relatively expensive to compute *per state* which shows for N3, C5, C9, but the fact that the runtime is so small whenever the size of the explored search space is small exhibits the advantage of on-the-fly computation: in contrast, for heuristics based on predicate abstraction and pattern databases, it could happen that a heuristic is very good but expensive to compute because the abstract state space is big.

On the other hand, we compared with a kind of abstraction we developed previously [16], where the repertoire of predicates consists of all guards and invariants that textually occur in the system definition, together with all possible location predicates. Since taking one abstraction based on all these predicates is much too inefficient to compute, we only take the predicates of one single process, generate an abstract state space, and see if the number of states or the number of layers exceeds certain thresholds. If yes, we take the current predicate set for defining one pattern database and start generating another one based on another process. Otherwise, we continue to add the predicates coming from another process to the current abstraction. Typically we end up with two or three abstractions this way. The results are referred to as "syn" for "syntax-based".

A notation such as "5000/10/SUM", used not only for "syn" but also for other heuristics, means that the threshold on the number of states (resp., layers) is 5000 (resp., 10), and that we use the SUM option (see Sec. 3). The figures for "syn 20000/20/MAX" and "syn 20000/20/SUM" are identical in Table 1 because only one (or only one non-trivial) abstraction was generated, which is why we write "SM".

For M and N, the results are never among the best and often among the worst. For C, the explored state space size is sometimes among the best (note in particular C9), but often among the worst as far as the runtimes are concerned, as computing the heuristic is so expensive. The problem with "syn" is that composing predicate sets along process boundaries as described above is a very coarse approach: it can happen that we have a predicate set that is very small and yields a heuristic that is fast to compute but not very good, and then adding the predicates from another process immediately results in an abstraction that might be good but is extremely expensive to compute.[6] In contrast, with our abstraction refinement the differences between subsequent abstractions will typically be much smaller so that the balance between heuristic quality and heuristic computation time can be better tuned.

[6] This problem also occurs when we use off-the-shelf abstraction refinement [16].

5.2 Using the (Joint or Separate) Target Locations as Initial Predicates

In our benchmarks the error condition is given by a formula of the form $(loc(p_1) = \ell_1) \wedge (loc(p_2) = \ell_2)$, i.e., the error state consists of process p_1 being in ℓ_1 and p_2 being in ℓ_2 simultaneously. The simplest setup we tried is the following: the initial abstraction consists of the two location predicates for the two target locations.[7] The abstraction can thus definitely distinguish a target state from a non-target state, which in our intuition is a good basis for a heuristic function. In each step, the refinement is based on a single abstract path of maximal length in the current abstract space. The results for this strategy are marked in Tables 1 and 2 with "1abs".

As a termination criterion for the refinement, we used thresholds on the size of the abstract state space and on the number of layers as in Sec. 5.1. Whenever one of them is exceeded, then the current abstraction will be the last one generated in this refinement loop. The thresholds were chosen to pose a sensible limit on the computational resources spent in building the abstractions.

Although it seems intuitively reasonable to have an initial abstraction that contains location predicates for both target locations, on the other hand one may also argue that it is good to combine several abstractions, and a simple way of doing this is to have two abstractions, each based on the initial predicate set $\{loc(p_i) = \ell_i\}$ for $i = 1, 2$. Everything else is as before. The results are marked in the tables with "2abs".

Concerning the size of the explored state space, "2abs" is better than "1abs" on most examples. This suggests that, *ceteris paribus*, the argument that having several different heuristics is better than having just one outweighs the argument that a good heuristic should at least capture the difference between a target state and a non-target state. Therefore, in the investigation of other aspects of the design space presented in the sequel, we decided to base the initial abstractions on the two targets kept separate.

Concerning the total runtime, "1abs" is sometimes negligibly better than "2abs" because we only have to compute one abstract space instead of two.

Now compare "2abs" to the other heuristics. The results are never extraordinarily bad: in the few cases where the results are among the worst, they are "in good company". The results are often among the best, but in none of these cases are they dramatically better than the corresponding result for "seed", see Sec. 5.4. Compared to "syn", the balance between heuristic quality and heuristic computation time can be better tuned because the differences between subsequent abstractions are typically small.

[7] Note that having two processes involved in the target condition does not mean that the whole system consists of just these two processes.

Table 3. Refinement based on q paths of length l, setting 5000/10/SUM

$l \rightarrow$	$\frac{w}{1}$	$\frac{w}{2},\frac{w}{3},\frac{w}{4}$	$\frac{w}{1}$	$\frac{w}{2},\frac{w}{3},\frac{w}{4}$	$\frac{w}{1}$	$\frac{w}{2},\frac{w}{3},\frac{w}{4}$	$\frac{w}{1}$	$\frac{w}{2},\frac{w}{3},\frac{w}{4}$
$\downarrow q$	M1		M2		M3		M4	
1,2,4,8	14955	22162	17753	38234	86316	61252	110240	110240
	N1		N2		N3		N4	
1,2,4,8	17584	30269	38216	69276	157491	145018	353609	353609

$l \rightarrow$	$\frac{w}{1}$	$\frac{w}{2}$	$\frac{w}{3}$	$\frac{w}{4}$	$\frac{w}{1}$	$\frac{w}{2}$	$\frac{w}{3}$	$\frac{w}{4}$	$\frac{w}{1}$	$\frac{w}{2}$	$\frac{w}{3}$	$\frac{w}{4}$
$\downarrow q$	C2				C5				C8			
1	7382	13653	23389	23389	258	258	522	522	106065	293694	308413	308413
2,4,8	7382	12500	12500	12500	258	258	258	258	106065	288112	288112	288112
	C3				C6				C9			
1	2866	5208	9634	9634	3127	14707	15925	15925	451610	1412455	1560656	1560656
2,4,8	2866	4819	4819	4819	3127	12922	12922	12922	451610	1413494	1413494	1413494
	C4				C7							
1	2679	4989	8587	8587	31584	92322	98028	98028				
2,4,8	2679	4600	4600	4600	31584	86465	86465	86465				

5.3 Abstractions Based on Several Paths

In this setup, each iteration of the refinement loop is based on refining *several* paths, each as shown in Fig. 3. We generated two abstractions, each based on the initial predicate set $\{loc(p_i) = \ell_i\}$ for $i = 1, 2$.

For each target, the refinement loop works as follows: given the current abstract space, we select several paths for doing refinement — selecting *all* paths would be too expensive, selecting just one path might result in too few predicates being added in the refinement step. Moreover, the longer the abstract path the more predicates will be added due to this path. Therefore our intuition was that the number and quality of the predicates added are strongly correlated to the length of the abstract paths one chooses, and so we decided that all abstract paths should all have the same length l, which is thus a parameter of the method. Technically, in an ad hoc random way one selects up to q states in the abstract state space such that there is an abstract path of length l from such a state to the error state. Then one refines the current abstraction based on these paths.

In Table 3, we show the explored state space size depending on q and l. Here w is the number of layers of the current abstract state space, e.g., $l = \frac{w}{2}$ means that the refinement is based on abstract paths whose length is half the maximal length of any path in the current abstract state space. We set l to a minimum of 3 to avoid some extremely short paths. Note that the results for $l = \frac{w}{1}$, $q = 1$ are necessarily identical to those of “2abs 5000/10/SUM”.

The parameter q has little influence on the number of explored states. I.e., it does not seem to matter which or how many of the paths are chosen for the refinement. This is a bit surprising, and contrary to our intuition [16] that off-the-shelf abstraction refinement is too much focused on a single path, which should be bad because it leads to abstractions that are fine in some regions and coarse in others. In contrast, the parameter l is important. If l is too small, then the refinement will not "take off", i.e., it will converge very quickly generating very few predicates (with M3 and N3 as exceptions).

We analysed some of the predicate sets generated for our benchmarks more closely. Our observations suggest the following thesis: it is not the case that the above strategy could lead to many very different predicate sets, depending on how the parameters are chosen. Rather, for reasonably chosen parameters, an abstraction refinement based on one of the target locations will converge to a certain predicate set. This convergence can happen very early and the predicate set can be very small. This motivates the method described next, adding additional "seeds" for the abstraction. We go back to taking $q = 1$ and l maximal, as in Sec. 5.2.

5.4 Generating Several Abstractions Based on "Random Seeds"

Using the setups above, we observed that our refinement sometimes reaches a fixpoint extremely early, way before any "artificial" termination criterion (see Sec. 5.1) applies. If the final transition of the abstract path is $\ell \rightarrow \ell'$ and the initial abstraction is just the predicate $loc = \ell'$, then the abstract space will have two states T and F, and the refinement will not add any predicates. Generally, if the abstract path consists of location-wise consistent edges and either there are no integer guards/invariants or they have already been added, then the refinement stops, and this can happen very early, so that the abstraction is extremely coarse.

Our idea to overcome this phenomenon is to insert a "random" location predicate as an additional "seed" for the abstraction refinement, i.e., to use a slightly finer initial abstraction. As a first naïve realisation of this, we generated abstractions by refinement based on each pair $(loc = \ell_0, loc = \ell)$ where ℓ_0 is a target location and ℓ any other location, for any process. This will give dozens of abstract spaces for our examples, so we had to find ways of generating fewer abstractions yet sufficiently many interesting ones. After observing that many generated spaces are extremely similar, we pruned spaces that are likely to be similar to previously generated spaces, as follows: firstly, if for a refinement based on some seed predicate, the generated abstract space in the first refinement iteration consists of abstract states where the seed predicate is always U, then we deem the seed predicate to be useless and abort the refinement based on it. Secondly, we take at most one location predicate per system process as seed, that is to say, we consider the location predicates of each process in turn, but once we have successfully generated an abstraction for the current location predicate, we disregard all remaining location predicates of the current process. With this approach, we typically obtained between four and six abstractions.

The results are marked in Tables 1 and 2 with the word "seed". The results are very unstable, sometimes extremely good, sometimes extremely bad. An improvement of the approach could be to allow for random restarts, that is to say, whenever the error search is taking too long, one might abort the search and compute a new heuristic function, obtained by modifying some of the parameters. This is in loose analogy to random restarts in propositional satisfiability solving [21]. Alternatively, one could have several searches running in parallel, based on several abstractions obtained by choosing the parameters in different ways.

For the M and N examples, the search space sizes are sometimes the same for "100/10" and "5000/10". Also, some search space sizes are identical to the respective values for "2abs". Of course, we do not believe that this is a pure coincidence, but rather, that it is due to the fact that the underlying abstractions are similar. While inspection confirms this to some extent, it is not obvious that the abstractions should be so *extremely* similar, but we do not consider this to be a particularly important issue.

6 Related Work

Using pattern database heuristics for DMC has been proposed in [25]. The systems considered are finite-state transition systems, where each state consists of an assignment of values from a finite domain to the state variables. In a next step, it is assumed that a state can be encoded as a bitvector. Unlike in our work, the bits are not interpreted as logical formulas that "talk about" system states, and predicate abstraction is not even mentioned. The encoding of states as bitvectors is no abstraction, in the sense of information loss; it is only in the next step that a pattern database abstraction is defined by ignoring some of the bits. Note that there is no abstraction *refinement* in that work.

Another work by the same authors [24] does consider refinement, however it is not the abstractions that are refined, but the abstract error paths themselves are refined, removing those that are spurious.

In [12], pattern database heuristics are also used for DMC. Refinement is not considered. It is observed that combining several pattern databases is useful.

In [10], DMC of timed automata is considered. The heuristic function is based on an abstraction that merges locations until there are at most N locations left, where N is a parameter. The heuristic function is read off the overall merged automaton. This approach does not involve predicate abstraction or refinement.

We have already mentioned another approach to DMC, which we used for comparison because it is among the best for our benchmarks [20]. Several heuristic functions including [10, 20] have been joined in the tool UPPAAL/DMC [19].

In our own previous work [16], we have presented a first implementation of predicate abstraction for DMC of timed automata, including one approach where the abstraction was generated using refinement. Unlike in the work presented here, in the implementation we actually used the tool ARMC [23]. The setup of the refinement process differs substantially from the one of this paper — most importantly, in [16] we have a preprocessing of the original system splitting it into several subsets of its processes, similarly as described for "syn". The motivation for implementing the refinement process ourselves instead of relying on ARMC is that we wanted to cater for the aspects that are different in our scenario, as discussed in Sec. 4.

7 Discussion

In this paper, we have presented various methods of setting up a predicate abstraction refinement loop for the purpose of generating heuristic functions for DMC.

We found that generating predicates by collecting guards along abstract paths is useful for generating reasonably small and good abstractions (Sec. 5.2), in contrast to the syntax-based abstractions [16] that tend to be too big ("syn"). So the main benefit is that abstraction refinement selects good predicates from a repertoire.

Moreover, we hoped that abstraction refinement enlarges the repertoire of predicates, as in Fig. 2, where the predicate $i = j + 3$ cannot be read off the system syntax. We have looked into this, and it seems to be of hardly any significance. For our benchmarks it happens very rarely (though not never!) that a predicate is generated that is not contained as a guard or invariant in the system. Thus for these benchmarks, generating predicates by updating guards and invariants is overly subtle and complicated.

Basing the refinement loop on several, not necessarily error paths rather than just one path did not have the positive impact of adding more predicates that we hoped (Sec. 5.3). As we explained in Sec. 5.4, this is mostly due to the fact that the refinement can reach a fixpoint at a very early point: the abstract paths, even though extremely short, do not give rise to the addition of any further predicates.

The idea of adding seeds (Sec. 5.4) provides a remedy to this problem. The idea can be varied in many ways, of which we discussed some. It can yield very good results, but the results are quite unpredictable, and we still hope to enhance abstraction refinement using some technique which systematically improves on the method of Sec. 5.2.

In fact, suppose there is a more stable solution to the problem of early convergence, consistently obtaining reasonably good results. Then this solution might provide the key to boosting the performance of our initial idea, basing the refinement on several paths rather than just one path. This is a promising issue to be explored in further work.

In summary, while the application of abstraction refinement for the intelligent creation of heuristic functions is an exciting idea, making this idea work in practice is non-trivial and involves several subtle issues. A variety of these issues has been identified and investigated in our work. It remains to be seen in future work how the prevailing difficulties can be overcome. In particular, it may be that some of the issues disappear or are less critical in other search problems. Hence investigating the method in other contexts – other DMC formalisms or AI problems – is important.

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 TR14/AVACS). We would like to thank Henning Dierks, Sebastian Kupferschmid, Bernhard

Nebel, Andreas Podelski, and Martin Wehrle for useful discussions and help with the implementation. We could also like to thank the anonymous reviewers of this paper for their useful comments.

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Completeness of abstraction refinement for software model checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*. Springer-Verlag, 2002.
3. Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *International Journal on Software Tools for Technology Transfer*, 5(1):49–58, 2003.
4. Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Revised Lectures on Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 200–236, 2004.
5. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets 2003*, volume 3098 of *LNCS*, pages 87–124. Springer-Verlag, 2004.
6. Edmund Clarke, Anubhav Gupta, and Ofer Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Transactions on Computer Aided Design*, 23(7):1113–1123, 2004.
7. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
8. Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.
9. Henning Dierks. Comparing model checking and logical reasoning for real-time systems. *Formal Aspects of Computing*, 16(2):104–120, 2004.
10. Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking Software*, volume 3925 of *LNCS*, pages 19–34. Springer-Verlag, 2006.
11. Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology*, 5(2-3):247–267, 2004.
12. Stefan Edelkamp and Alberto Lluch-Lafuente. Abstraction in directed model checking. In *Proceedings of the ICAPS Workshop on Connecting Planning Theory with Practice*, pages 7–13, 2004.
13. Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. ICS: Integrated canonizer and solver. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification*, volume 2102 of *LNCS*, pages 246–249. Springer-Verlag, 2001.

14. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.
15. Jörg Hoffmann and Jana Koehler. A new method to index and query sets. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 462–467. Morgan Kaufmann, 1999.
16. Jörg Hoffmann, Jan-Georg Smaus, Andrey Rybalchenko, Sebastian Kupferschmid, and Andreas Podelski. Using predicate abstraction to generate heuristic functions in UPPAAL. In Stefan Edelkamp and Alessio Lomuscio, editors, *Post-Proceedings of the 4th (2006) Workshop on Model Checking and Artificial Intelligence*, volume 4428 of *LNCS*, pages 51–66. Springer-Verlag, 2007.
17. Richard E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference*, pages 700–705. MIT Press, 1997.
18. Bernd Krieg-Brückner, Jan Peleska, Ernst-Rüdiger Olderog, and Alexander Baer. The UniForM workbench, a universal development environment for formal methods. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *LNCS*, pages 1186–1205. Springer-Verlag, 1999.
19. Sebastian Kupferschmid, Klaus Dräger, Jörg Hoffmann, Bernd Finkbeiner, Henning Dierks, Andreas Podelski, and Gerd Behrmann. UPPAAL/DMC – Abstraction-based heuristics for directed model checking. In Orna Grumberg and Michael Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 679–682. Springer-Verlag, 2007.
20. Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an AI planning heuristic for directed model checking. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking Software*, volume 3925 of *LNCS*, pages 35–52. Springer-Verlag, 2006.
21. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM, 2001.
22. Judea Pearl. *Heuristic*. Addison-Wesley, 1985.
23. Andreas Podelski and Andrey Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In Michael Hanus, editor, *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages*, volume 4354 of *LNCS*, pages 245–259. Springer-Verlag, 2007.
24. Kairong Qian and Albert Nymeyer. Abstraction-based model checking using heuristical refinement. In Farn Wang, editor, *Proceedings of the 2nd International Conference on Automated Technology for Verification and Analysis*, volume 3299 of *LNCS*, pages 165–178. Springer-Verlag, 2004.
25. Kairong Qian and Albert Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 497–511. Springer-Verlag, 2004.