Adapting an AI Planning Heuristic for Directed Model Checking

Sebastian Kupferschmid[1], Jörg Hoffmann[2], Henning Dierks[3], and Gerd Behrmann[4]

[1] University Freiburg, Germany, kupfersc@informatik.uni-freiburg.de
[2] Max Planck Institute for CS, Saarbrücken, Germany, hoffmann@mpi-sb.mpg.de
[3] Oldenburg University, Germany, Dierks@informatik.uni-oldenburg.de
[4] Aalborg University, Denmark, behrmann@cs.aau.dk

Abstract. There is a growing body of work on directed model checking, which improves the falsification of safety properties by providing heuristic functions that can guide the search *quickly* towards *short* error paths. Techniques of this kind have also been made very successful in the area of AI Planning. Our main technical contribution is the adaptation of the most successful heuristic function from AI Planning to the model checking context, yielding a new heuristic for directed model checking. The heuristic is based on solving an abstracted problem in every search state. We adapt the abstraction and its solution to networks of communicating automata annotated with (constraints and effects on) integer variables. Since our ultimate goal in this research is to also take into account clock variables, as used in timed automata, our techniques are implemented inside UPPAAL. We run experiments in some toy benchmarks for timed automata, and in two timed automata case studies originating from an industrial project. Compared to both blind search and some previously proposed heuristic functions, we consistently obtain significant, sometimes dramatic, search space reductions, resulting in likewise strong reductions of runtime and memory requirements.

1 Introduction

When model checking safety properties, the ultimate goal is to prove the absence of error states. However, to do so one has to explore the entire state space of the application under consideration. It is therefore essential to use an efficient representation and implementation of that state space. Prominent examples of such implementations are the SPIN (e.g. [15]) and UPPAAL (e.g. [1]) tools. SPIN handles the Promela language, describing systems of communicating processes. UPPAAL handles networks of extended timed automata, which is a formalism with less complex communication than Promela, but where the processes can be annotated with real-valued clock variables. Both languages also feature integer variables.

Enumerating the entire state space is often not feasible in practise. A potentially much easier task is to only try to *detect* error states, i.e., to *falsify* the safety property. An error may be found by exploring only a small fraction of the search space. Algorithms that are good at detecting errors can be used for debugging purposes. They can even be good for proving an application error-free, because they can be used to handle the intermediate iterations in the abstraction refinement life cycle, i.e. those iterations in which spurious error states exist.

There are two main issues to be addressed: first, the *search space size*, i.e. the number of search states that need to be considered before the error state is found; and second, the length of the detected *path* to the error state. The search space size determines the scalability of the search. Short error paths are easier to understand for debugging; in abstraction refinement, they provide better information about what aspects of the abstraction should be refined. Ideally, one wants an *optimal*, i.e. a shortest possible, path to an error.

Both search space size and error path length can be addressed by the *order* in that the search states are explored. One defines a *heuristic function* h, a function that maps states to integers, estimating the state's distance to the nearest error state. The search then gives a preference to states with lower h value. There are many different ways of doing the latter, of which we consider the wide-spread methods A^* search and *greedy search*. In the former, search nodes s are explored by increasing value of $c(s) + h(s)$ where $c(s)$ is the length of the search path on that s was reached. If h is *admissible*, i.e., if it never overestimates the real distance to the nearest error state, then A^* is guaranteed to return an optimal error path. In greedy search, search nodes are explored by increasing value of $h(s)$. This gives no guarantee on the length of the detected error path, but tends to explore less search states in practise.

The application of heuristic search to model checking was pioneered a few years ago by Edelkamp et al [9, 10], christening this research direction *directed model checking*, and inspiring various other approaches of this sort, e.g. [12, 18, 7]. 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?* Different definitions make all the difference because no heuristic can work well in *all* examples, and the best one can hope to do is to define a range of heuristics that cover (work well in) an as large as possible range of examples.

Edelkamp et al [9, 10] work in the context of SPIN. They propose to base the distance estimation on the graph-distances within each single process. For process i, let $d(i)$ be the distance of i's start location to its target location, when ignoring all edge guards (if there is no target location, set $d(i) := 0$). Then an admissible heuristic function, called d^L, is defined as $max_i d(i)$, and a non-admissible heuristic function, called d^U, is defined as $\sum_i d(i)$. We implemented these heuristic functions in UPPAAL, taking the $d(i)$ to be the graph distances in the individual automata.

Note that d^L and d^U are rather crude approximations of the system semantics. They completely ignore communication and integer variables. Our main contribution in this paper is an approximation technique that does *not* do that. The approximation is more costly – i.e., computing the heuristic function takes more runtime than what is needed for d^L and d^U – but, as we will see, this often pays off in terms of much smaller search spaces. We obtain our approximation by adapting the most successful heuristic method [5, 14] from the area of AI Planning, where heuristic search has been overwhelmingly successful in the past decade, in particular winning all the planning competitions (e.g. [14, 11, 19]).

The heuristic method is based on what AI people call a *relaxation*, which is the same as the model checking term *abstraction*: an over-approximation. The abstraction technique used is, however, quite different from what one usually uses in model checking, due to the very different way of *using* the abstracted task. Namely, the heuristic values are generated by solving the abstract problem in every search state, and taking the length of the abstract solution as the distance estimate. To be able to solve the abstract problem in every search state, of course the abstraction has to be very coarse. In our particular case, the abstraction assumes that *every state variable, once it has obtained a value, keeps that value forever.* Which means, in the abstraction the "value" of any variable at any time point is not a member but a subset of the variable's domain. The subsets grow monotonically as abstract transitions are taken. We prove that, like in the planning context, solving the abstract problem optimally, i.e., finding an optimal abstract error path, and thereby computing an admissible heuristic function, is still **NP**-hard, even if the addressed formalism allows only parallel automata with communication. For parallel automata with communication and integer variables, we define two polynomial-time methods for approximating the length of an optimal abstract error path. We call the resulting heuristic functions h^L and h^U. The former is a lower bound on the length of an optimal abstract error path, the latter is an upper bound on that length; h^L is admissible, h^U is not.

Our heuristics are implemented inside the UPPAAL system, since our goal in this research is to speed up model checking of (networks of extended) timed automata. Ultimately, of course, we want to develop heuristics that also take into account the clock variables. We are currently investigating that direction; it is highly non-trivial in our context due to the nature of our abstraction. Since timed transitions are continuous, the value subset of a clock x will be $[0, \infty)$ as soon as one reaches a location without an invariant limiting x; we discuss this in more detail below. As said, so far we can offer heuristics that take into account communication and integer variables. To the best of our knowledge, no similar heuristics were developed in any other area of model checking (the differences to the existing other heuristics are outlined in the related work section).

In the standard versions of UPPAAL, the search order can be fixed to either depth-first (DF) or breadth-first (BF).[1] We test our implementation in networks of extended timed automata. We consider a few toy examples, and two realistic case studies coming from an industrial project. We evaluate the performance of different UPPAAL configurations finding optimal error paths, and of UPPAAL configurations finding (possibly) sub-optimal error paths. The former are BF,

[1] There is also a version doing heuristic search [3], but for that the user has to provide the heuristic function manually, in difference to our fully-automatic technology. Note that a successful manual heuristic specification requires inside knowledge on the side of the user, and careful tuning.

and A^* with h^L or d^L; the latter are randomised DF, and greedy search with h^L, h^U, d^L, and d^U (remember that d^L and d^U were defined by Edelkamp et al [10]). Of the optimal configurations, BF and A^* with d^L perform roughly similarly except in the toy examples; A^* with h^L brings a moderate runtime advantage, but much smaller search spaces, enabling success in one more example due to the lower memory usage. For the (potentially) sub-optimal configurations, our results are much stronger. While the d^L and d^U search orders bring hardly any advantage over DF in our industrial case studies, both h^L and h^U yield dramatic search space reductions, and with that better runtimes and the ability to solve more examples. At the same time, the error paths found with h^L and h^U are orders of magnitude shorter than those found with DF, d^L, and d^U.

The next section briefly gives our notations. Sections 3 and 4 formally define the abstraction used, and the algorithms computing the heuristic functions, respectively. Section 5 describes our empirical results, Section 6 discusses related work. Section 7 closes the paper. Most proofs are replaced in the text by short proof sketches; the full proofs are in an appendix.

2 Notations

We assume the reader is roughly familiar with timed automata and their commonly used extensions. We give a brief description of the particular formalism treated in our current implementation. We use (a slight variation of) the terminology and notation given by Behrmann et al [2].

We treat networks of timed automata with binary synchronisation and integer variables. For the sake of presentation herein, we restrict atomic expressions over integer variables to variables, variable increments/decrements, or constants. That is, we allow only comparisons like $v \leq v'$ or $v = c$, and assignments like $v := v'$, $v := c$ or $v := v \pm 1$. Our implementation in fact deals with arbitrary linear expressions over the variables; for the sake of readability, we omit these and only explain the extensions in the text. As mentioned earlier, the heuristic function so far completely ignores the clock variables (the reasons for this are explained in Section 3.2). We therefore don't give formal notations for these variables. Our notations are as follows. The timed automata share a set A of actions, and a set V of integer variables. Each $v \in V$ has a domain $dom(v)$. Each automaton i has a location set $L(i)$, a start location $l^0(i)$, and a set of edges $E(i)$. Each edge is annotated with an action $a \in A$, with a guard g, and with an *effect* f. The guard is a conjunction of conditions of the form $x \bowtie y$ where $x, y \in \mathbb{Z} \cup V$ and $\bowtie \in \{<, \leq, =, \geq, >, \neq\}$. The effect is a list of assignments of the form $v := v'$, $v := c$ or $v := v \pm 1$, where $v, v' \in V$ and $c \in \mathbb{Z}$. Each variable v occurs on the left hand side of at most one such assignment. The semantics are defined as obvious. Transitions are asynchronous and triggered by an edge annotated with a special void action, or synchronous and triggered by two edges with inverse actions.

The safety properties we can verify take the form of (negated) edge guards plus location vectors, i.e., our implementation can check whether there exists a reachable state in that the automata are in specified locations, and that satisfies a conjunction of conditions $x \bowtie y$. We call the former the *target locations*, and the latter the *target formula*. A path of transitions is called a *solution* if it leads from the start state to a state complying with target locations and target formula.

3 Abstraction

We introduce the abstraction method, called *monotonicity abstraction*, underlying our implemented heuristic function. We first give a high-level description of the abstraction in a generic way, then we define it as currently used in the context of networks of automata.

Before we start, let us remark that the monotonicity abstraction was first invented in AI Planning for a formalism called STRIPS, under the name "ignoring delete lists" [5]. In STRIPS, the "delete lists" are effect instructions that make a boolean variable FALSE. This simplifies the problem because, in STRIPS, variables are only ever required to be TRUE. The monotonicity abstraction we describe below is a generalisation of this abstraction approach. We remark that the generalisation is *not* published in the AI Planning literature; it is, in spirit, somewhat similar to the framework presented in [8].

3.1 The Monotonicity Abstraction

The abstraction is based on the simplifying assumption that *every state variable, once it obtained a value, keeps that value forever.* The value of a variable is no longer an element, but a *subset* of its domain. That subset grows monotonically over transition applications – hence the name of the abstraction.

In a little more detail, in general a transition system (a planning task, a system of timed automata, a piece of program code, etc.) can be viewed as given by a set of state variables, a set of transition rules, a start state, and a target formula. The transition rules have a guard – a formula out of some class of valid (non-temporal) formulas – and an effect – an instruction how the variable values change when the rule is applied. States are value assignments to the variables, the target formula is a valid formula. A solution is a path of transitions that, when applied to the start state, ends in a state that satisfies the target formula.

Under the monotonicity abstraction, the semantics of a transition system as above are changed as follows. States now map each variable to a subset of its domain. The start assignment contains the single value assigned by the start state. A formula evaluates to TRUE in a state if there *exists* a variable value vector in the state so that the formula evaluates to TRUE when inserting these values. Executing an effect instruction becomes a *set union* operation, where the new value of each variable x is its old value (a domain subset) plus the new value assigned by the effect. If the effect outcome depends on variables, then all possible value vectors for these variables are used, each yielding a value for x.

E.g., say we have one integer variable v, and one transition with guard $v = 0$ and effect $v := v + 1$. The start state is $v = 0$, and the target formula is $v = 2$.

Obviously, there is no solution. There is, however, a solution in the abstraction. The start assignment is $\{0\}$. After one transition, this becomes $\{0, 1\}$. Since the transition guard is abstracted to $\exists c \in s(v) : c = 0$, the transition can be applied a second time, and we get the state $\{0, 1, 2\}$: the new values obtained for v are 1 (inserting 0 into the effect right hand side) and 2 (inserting 1). In this state the abstract target formula, taking the form $\exists c \in s(v) : c = 2$, evaluates to TRUE.

It is not difficult to see that the monotonicity abstraction induces an over-approximation of the real transition system: every solution path in the real system corresponds to a solution path in the abstract system. We will state this formally below, for our abstraction of timed automata. In many cases, deciding solution existence is a polynomial-time problem under the abstraction, making it feasible to solve the abstract problem in every search state.[2]

3.2 The Monotonicity Abstraction in Timed Automata

Before we give our definitions, consider at a higher level of abstraction what happens if we apply the above abstraction to a system of timed automata. Under the abstraction, each automaton will (potentially) be in several locations in a state. The integer variables will have several possible values in a state. The clock variables will only accumulate new values. Transitions will be applicable as soon as one of the possible value vectors satisfies the guard.

Thinking a little more about the clocks, one sees that they are likely to trivialise very quickly under the abstraction. The reason for that are the *timed* transitions: as time passes, the clocks accumulate all the passing time points. After waiting from time point u to time point $u + d$, the new clock value subsets contain the entire interval $[u, u + d]$. So in a location with invariant I, the clock value subsets immediately gather all values up to the upper bound specified by I. Now, all clock values are 0 initially. Since time passes continually, therefore the clock value subsets will always have the form $[0, max]$ (where max is the latest time point yet reached), containing no information other than max. As soon as a location with empty invariant is reached, max will become infinite, i.e., the clock value subsets will be the entire time line.

For the above, reasoning about clock values under the abstraction is not likely to contribute useful information, unless additional techniques are used. We outline an idea for such additional techniques in Section 7. For now, we ignore the clocks altogether (inside the heuristic function). While this is undesirable, as said our empirical results demonstrate that taking (abstract) account of automaton locations, synchronisation, and integer variables can yield useful search guidance.

Our definitions are straightforward and read as follows. We denote abstract constructs with a superscribed $^+$ to indicate the additivity of the abstraction. An abstract state s^+ assigns each automaton i a location subset $s^+(i) \subseteq L(i)$. Each

[2] Under certain conditions, checking satisfaction of a formula becomes **NP**-hard in the abstraction, due to the additional existential quantification. In particular, this is the case in our context of timed automata. We make an additional simplification to get around this, see the explanation below.

integer variable v is assigned a value set $s^+(v) \subseteq dom(v)$. Formulas (conjunctions of conditions) are abstract by, "locally", existentially quantifying the variables *in each condition separately.* E.g. a formula $v \bowtie_1 v' \wedge v \bowtie_2 c$ is abstracted to $\exists c_1 \in s^+(v), c_1' \in s^+(v') : c_1 \bowtie_1 c_1' \wedge \exists c_2 \in s^+(v) : c_2 \bowtie_2 c$. That is, we allow achievement of each condition in separate. When, "globally", quantifying the variables over the entire formula, one gets an **NP**-complete constraint problem, so there is no way around making further abstractions. We chose to do local quantification mainly because it is very simple and can be implemented efficiently. Also, it comes in handy also for linear arithmetic. When allowing linear arithmetic between integer variables, checking even a single condition $\exists \bar{x} : f(\bar{x}) = c$ is **NP**-hard. This isn't usually a problem since the number of variables in the expressions ($f(\bar{x})$) is typically small, up to four maybe.[3] However, the total number of variables in a *conjunction* of expressions can become quite big. So it is convenient to address the single expressions in separate.

An assignment $v := c$ results in $s^+(v) := s^+(v) \cup \{c\}$. An assignment $v := v'$ results in $s^+(v) := s^+(v) \cup s^+(v')$. An assignment $v := v + 1$ results in $s^+(v) := s^+(v) \cup \{c+1 \mid c \in s^+(v)\}$, $v := v-1$ results in $s^+(v) := s^+(v) \cup \{c-1 \mid c \in s^+(v)\}$. Values not contained in $dom(v)$ are removed from the result. An asynchronous transition of automaton i from location l to l' is enabled if $l \in s^+(i)$, and the respective abstract edge guard holds in s^+. The effect assignments are executed as above, and $s^+(i) := s^+(i) \cup \{l'\}$ is set. A synchronous transition of automaton i from location $l(i)$ to $l'(i)$, and of automaton j from location $l(j)$ to $l'(j)$, is enabled if $l(i) \in s^+(i)$, $l(j) \in s^+(j)$, and both respective abstract edge guards hold in s^+. The effect assignments are executed as above, and $s^+(i) := s^+(i) \cup \{l'(i)\}$as well as $s^+(j) := s^+(j) \cup \{l'(j)\}$ are set.

When the start state is s_0, s_0^+ is given by $s_0^+(i) = \{s_0(i)\}$, and $s_0^+(v) = \{s_0(v)\}$. A path of successively enabled transitions from s_0 is a *abstract solution* if it ends in a state s^+ in which the abstract target formula holds.

Proposition 1. *Given a network of timed automata with binary synchronisation and integer variables, a start state, target locations, and a target formula. If $t_1, \ldots, t_n$ is a solution then $t_1, \ldots, t_n$ is also an abstract solution.*

Proof Sketch: The variable values achieved by $t_1, \ldots, t_n$ in the abstraction subsume the values achieved in reality. ■

By Proposition 1, every solution in the real search space is also contained in the abstract search space. So the length of an optimal abstract solution is an admissible heuristic function. We will come back to this below.

Consider Figure 1 as an example. The top automaton needs to go through repeated circles. More precisely, if the bottom automaton has n locations, then the real solution takes $2(n-1)$ steps, half of which are synchronized between both automata. However, an abstract solution can be obtained in only n steps: the top automaton goes to the right once, and can then go to the left ten times in sequence since its right location remains in the reached location subset.

[3] Also, one can handle the expressions in an incremental way, see Section 4.

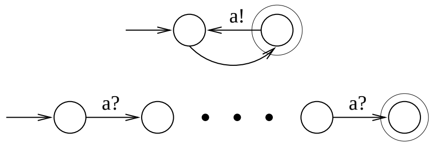

Fig. 1. A simple example where h^L and h^U deliver bad heuristic values.

We can decide in polynomial time if there exists an abstract solution or not.

Theorem 1. *Let TASolEx$^+$ denote the following problem. Given a network of timed automata with binary synchronisation and integer variables, a start state, target locations, and a target formula. Is there a abstract solution?*

TASolEx$^+$ is in **P**.

Proof: A polynomial solution algorithm is described in Section 4. ■

The polynomial solution algorithm forms the basis of our heuristic functions: for a heuristic function, what we want to know is not primarily if there is an abstract solution, but what the *length* of an abstract solution is (if there is one). Abstract solutions may contain arbitrarily many useless transitions, and we want to know what an *optimal* abstract solution is. We call the length of such a solution, for a state s, $h^+(s)$. Unfortunately, computing h^+ is still hard.

Proposition 2. *Let TASolMin$^+$ denote the following problem. Given a network of timed automata with binary synchronisation, a start state, a target formula, and an integer b. Is there an abstract solution of length at most b?*

TASolMin$^+$ is ***NP****-hard.*

Proof Sketch: By a straightforward reduction of 3SAT, using one automaton per clause and variable. ■

Note that *one does not even need integer variables* in the proof to Proposition 2. The desired admissible heuristic function h^+, based on our abstraction, can not be computed efficiently. So, in practise, we will have to *approximate* h^+. We introduce two approximation techniques in the next section, one computing a lower bound, and one computing an upper bound. Both are implemented as heuristic functions inside UPPAAL.

4 Approximating h^+

Our heuristic functions map search states to integers. For each state s during search, we are facing the following situation. We are given a network of timed automata, target locations, and a target formula. The start state is s. We want to approximate the length of an optimal abstract solution.

Both approximations are based on a forward-chaining algorithm that generalises algorithms proposed in the context of numeric planning [13]. The algorithm is a forward fixpoint computation. It determines in polynomial time if

$k := 0$, $L_0(i) := \{s(i)\}$ for all i, $V_0(v) := \{s(v)\}$ for all v
while target locations are not in L_k, or V_k does not model abstract target formula **do**
  $L_{k+1}(i) := L_k(i)$ for all i, $V_{k+1}(v) := V_k(v)$ for all v
  for all transitions t enabled by L_k and V_k **do**
    $L_{k+1}(i) := L_{k+1}(i) \cup \{l(i)'\}$ where t goes to $l(i)'$ in automaton i
    if t synchronously also goes to $l(j)'$ in automaton j **then**
      $L_{k+1}(j) := L_{k+1}(j) \cup \{l(j)'\}$
    endif
    if $v := c$ is an effect of t **then** $V_{k+1}(v) := V_{k+1}(v) \cup \{c\}$ **endif**
    if $v := v'$ is an effect of t **then** $V_{k+1}(v) := V_{k+1}(v) \cup V_k(v')$ **endif**
    if $v := v + 1$ is an effect of t **then** $V_{k+1}(v) := [min(V_k(v)), \infty]$ **endif**
    if $v := v - 1$ is an effect of t **then** $V_{k+1}(v) := [-\infty, max(V_k(v))]$ **endif**
  endfor
  if $L_{k+1}(i) = L_k(i)$ for all i, and $V_{k+1}(v) = V_k(v)$ for all v **then**
    $minlayer := \infty$, stop
  endif
  $k := k + 1$
endwhile
$minlayer := k$

Fig. 2. Building an abstract transition graph (ATG).

there is a abstract solution, by building a data structure called *abstract transition graph*, short *ATG*. The ATG is a layered graph encoding reachability information. Pseudo-code is given in Figure 2.

The ATG is a sequence of location sets $L_k(i)$ and of variable value sets $V_k(v)$: the graph *layers*. The algorithm builds these in an incremental way, so that their contents increase monotonically over k. Satisfaction of a formula, and enabled transitions, are defined in the obvious manner analogous to abstract states. In each iteration of the algorithm, for every enabled transition the respective new values are put into the sets. For the example from Figure 1, if the top automaton has locations t_1 (left) and t_2 (right), and the bottom automaton has locations $b_1, \dots, b_n$ (from left to right), then $L_0(top) = \{t_1\}$, $L_0(bottom) = \{b_1\}$, $L_1(top) = \{t_1, t_2\}$, and $L_1(bottom) = \{b_1\}$; for $2 \leq k \leq n$, we get $L_k(top) = \{t_1, t_2\}$ and $L_k(bottom) = \{b_1, \dots, b_k\}$. In $L_n(bottom)$ we have the target location and the algorithm stops.

The treatment of $v := v + 1$ and $v := v - 1$ effects is slightly more complicated, using a sort of "shortcut" to avoid the repeated incremental increasing (decreasing) of a variable up to (down to) a needed value n (which could take a number of iterations exponential in the representation of n). Setting a border of a $V_k(v)$ interval to ∞ is interpreted as telling us that arbitrarily high/low values can now be reached for v, by applying the respective effect.

It is important to note that the $V_k(v)$ sets can always be represented using only a number of values polynomial in the size of the input task, i.e. one does not need to explicitly enumerate all values in the reachable interval. If one of the bounds is infinite, one just records that plus the value at which the continuous region ends. In more detail, one can represent $V_k(v)$ by an ordered list of possible values, plus a marker at the lowest and highest value, indicating if or if not below/above the bound there is an infinite region inside $V_k(v)$. The values in the explicitly stored list can originate from $v := c$ assignments only, so their number is bounded by the number of such assignments in the input. It should be self-explanatory how this representation corresponds to the pseudo-code given in Figure 2. The representation of each $L_k(i)$ and $V_k(v)$ is polynomial. Satisfaction of an abstract formula in L_k and V_k can be tested in polynomial time processing the – at most binary – single conditions in the formula in turn; a condition on variables v and v' can be tested by, at most, processing the product of $V_k(v)$ and $V_k(v')$. Finally, after a polynomial number of iterations, L_k and V_k will not change anymore, or reach the respective full sets of locations/values. So altogether the algorithm terminates in polynomial time. It encodes admissible reachability information.

Lemma 1. *Given a network of timed automata with binary synchronisation and integer variables, a start state, target locations, and a target formula. If there is an abstract solution of length n, then the algorithm in Figure 2 stops successfully in an iteration $minlayer \leq n$.*

Proof Sketch: When building the ATG without stopping criteria, the abstract solution $t_1, \ldots, t_n$ is a sub-sequence of the ATG, i.e., t_k is enabled by L_{k-1} and V_{k-1}. The effects of t_k are over-approximated and contained in L_k and V_k. ∎

In particular, if the ATG terminates unsuccessfully, then there is no abstract solution. It is easy to see that, if the targets are reached in layer $minlayer$, then an abstract solution can be constructed as the sequence, for $k = 0, \ldots, minlayer - 1$, of all transitions enabled by L_k and V_k. So altogether the ATG is a polynomial procedure deciding existence of an abstract solution, and Theorem 1 follows.

Extending the ATG to deal with linear arithmetic over the integer variables does not require a lot of deep thought, but results in rather unreadable algorithm specifications. As said, testing $\exists \bar{x} : f(\bar{x}) = c$ is **NP**-hard for linear expressions $f(\bar{x})$, but the number of variables in $\bar{x}$ is typically small. Our main algorithmic trick to deal with the expressions efficiently is an *incremental* computation. If, at some point during building the ATG, we want to know whether $\exists \bar{x} : f(\bar{x}) = c$ is true based on the current value subsets (V_k), then we can refer back to the last time we asked that same question, and just take account of how the value subsets have changed since then. In fact, we just keep a flag at each expression occuring in the input, saying if or if not the expression can be satisfied yet. Every time the value subset of a variable occuring in the expression changes (grows), we see whether that change serves to satisfy the expression; if so, we set the flag. Checking guard satisfaction in the ATG then simply means to refer to the flags. Similarly, one can deal with linear expression effect right hand sides, $v := f(\bar{x})$. We just enumerate the set of value tuples for $\bar{x}$, referring back to the previous version of that set. Typically, just one or two variables in $f(\bar{x})$ have gathered new values since the last evaluation of $f(\bar{x})$. It suffices to enumerate these changes and extend the old tuple set correspondingly. Note that this incremental approach can, in fact, be implemented for (almost) arbitrarily complicated expressions, not only linear ones.

Let us focus again on how to approximate h^+. As said, we compute a lower bound as well as an upper bound. We call the lower bound h^L, and the upper bound h^U. By Proposition 1, a lower bound on h^+ is the *minlayer* value determined by the ATG algorithm. We set $h^L(s)$ to that value as computed by the ATG for s. Regarding an upper bound, note that, with the above, the number of all transitions enabled at layers $k = 0, \dots, minlayer - 1$ provides such a bound. However, this bound is likely to be far too generous, counting transitions that are reachable but not needed to achieve the targets. We therefore use a more involved method to determine our upper bound h^U. The method basically selects, at each layer $k = 0, \dots, minlayer - 1$, a *subset* of the enabled transitions, so that the sequence of the selected transitions is still an abstract solution. This is done by a backward-chaining procedure on the ATG. Since the details are not overly important here, the procedure is described in the appendix. The selected abstract solution is not necessarily optimal, and we set h^U to its length. Both h^U and h^L have the value ∞ in case there is no abstract solution (implying with Proposition 1 that there is no real solution either).

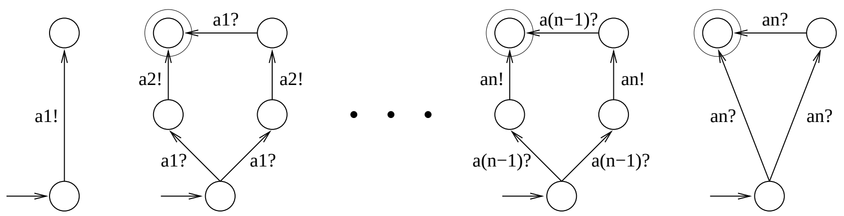

Fig. 3. A simple example where h^L and h^U deliver the precise error state distance.

Figure 3 gives another example. In the start state, all automata are in the bottom location. The error state is to reach the top left locations. In each automaton except the first one, one has two choices, one of which leads into a dead end (a state from which the error can not be reached), since the required communication signal won't be available anymore. Built for the start state, each layer k of the ATG corresponds exactly to the locations that can be reached within k steps – in particular, the top left location in the kth automaton from the left. So $minlayer = n$, and $h^L = h^U = n$ is the precise error state distance. If, during search, a wrong decision was made in automaton i, then the top left location in i does not appear in the ATG, and the heuristic value is ∞. So all dead ends are excluded from the search space. In contrast, $d^L = 2$ and $d^U = 2n - 1$ for the start state, and no dead ends are detected. Another example where h^L and h^U are precise is, e.g., a situation that requires (only) to repeatedly increment an integer variable. Intuitively, h^L and h^U are good at detecting long sequences of transitions that build upon each other to achieve some target, and at finding out that such a sequence is not available. What they are *not* good at is to see that *the same thing has to be done multiple times*[4] – under the monotonicity abstraction, everything needs to be done at most once. A bad situation was given earlier in Figure 1, where the top automaton needs to go through repeated circles, while h^L and h^U act as if a single circle is sufficient.

[4] When repeatedly incrementing a variable, every increment has a *different* effect.

5 Results

We ran experiments on an Intel Xeon 3.06 Ghz system with 4 GByte of RAM. As said, our configurations finding optimal error paths are UPPAAL's standard BF, and A^* search with h^L or d^L. Our sub-optimal configurations are UPPAAL's standard randomised DF, short *rDF* (which is by far the most efficient standard method across many examples, including ours), and greedy search with any of h^L, h^U, d^L, and d^U.

				t			S			M			l		
Exp	a	c	v	rDF	h^L	h^U	rDF	h^L	h^U	rDF	h^L	h^U	rDF	h^L	h^U
F_5^A	5	5	1	0.0	0.0	0.0	526	27	34	3	1	1	161	9	9
F_{10}^A	10	10	1	0.4	0.0	0.0	6371	42	54	7	1	1	1096	9	9
F_{15}^A	15	15	1	1.3	0.0	0.0	20010	57	74	10	1	1	2356	9	9
F_5^B	5	5	1	0.0	0.0	0.0	356	612	74	2	1	1	114	13	18
F_{10}^B	10	10	1	0.5	0.8	0.0	7885	55866	274	7	11	1	1363	29	33
F_{15}^B	15	15	1	3.8	40.3	0.0	58793	1.5e+6	599	18	75	1	6956	367	48
F_5^C	5	5	2	0.0	0.0	0.0	63	22	23	1	1	1	23	7	7
F_{10}^C	10	10	2	0.0	0.0	0.0	205	37	38	1	1	1	37	7	7
F_{15}^C	15	15	2	0.0	0.0	0.0	692	52	53	1	1	1	83	7	7
M_1	3	4	11	0.8	0.1	0.2	29607	5656	14679	7	1	9	1072	169	120
M_2	4	4	13	3.1	0.3	0.8	118341	30742	67398	10	11	11	3875	431	142
M_3	4	4	13	2.8	0.2	0.8	102883	18431	75976	9	10	11	3727	231	158
M_4	5	4	15	12.7	0.8	2.5	543238	76785	230466	22	13	16	15K	731	185
N_1	3	7	11	1.9	0.5	0.8	41218	16335	25577	7	10	10	1116	396	157
N_2	4	7	13	9.3	2.4	3.8	199631	88537	134444	13	13	13	4775	990	241
N_3	4	7	13	8.4	0.6	4.0	195886	28889	143969	12	11	13	3938	324	228
N_4	5	7	15	40.9	5.1	19.2	878706	240366	758167	39	20	31	18K	1671	282
C_1	5	3	12	0.8	0.2	0.2	25219	2339	3021	7	9	10	1056	95	87
C_2	6	3	14	1.0	0.3	0.5	65388	5090	7484	8	10	10	875	86	100
C_3	6	3	15	1.1	0.5	0.6	85940	6681	8259	10	10	10	760	109	101
C_4	7	3	17	8.4	2.5	3.8	892327	40147	65781	43	11	13	1644	125	140
C_5	8	3	19	72.4	13.2	16.7	8.0e+6	237600	333692	295	21	23	2425	393	218
C_6	9	3	21	–	10.1	94.7	–	207845	8.7e+6	–	20	223	–	309	1000
C_7	10	3	23	–	169	836	–	2.7e+7	9.2e+7	–	595	2.1G	–	1506	4630
C_8	10	3	24	–	14.5	932	–	331733	9.8e+7	–	23	2.3G	–	686	16K
C_9	10	3	25	–	1198	–	–	1.3e+8	–	–	2.5G	–	–	18K	–

Table 1. Experimental results for the sub-optimal configurations rDF, greedy search with h^L, and greedy search with h^U. Abbreviations: a number of automata, c number of clocks, v number of variables, t runtime in seconds, S search space size (number of visited states, "e+x" means $*10^x$), M peak memory used in MByte ("G" GByte), l length of detected error path ("K" thousand). Dashes indicate out of memory.

In the sub-optimal configurations, we use a *bitstate hashing* technique. This is a table with N entries, containing heuristic values, indexed by hash values of search states. Initially all table entries are empty. If the table entry for a new search state already contains a value, then that value is returned. Otherwise, the heuristic value is computed and stored in the table. This is a greedy method to bound the number of calls of the heuristic computation. After some limited experimentation, we set N to 256000 in the reported experiments.[5]

The tool executable and our benchmark examples are available for download from `http://www.informatik.uni-freiburg.de/~kupfersc/spin`. The data for the sub-optimal configurations are in Table 1 (rDF, h^L, and h^U) and Table 2 (d^L and d^U). The data for the optimal configurations are in Table 3. Below, we first explain the examples used, then we discuss the results.

				t		S		M		l	
Exp	a	c	v	d^L	d^U	d^L	d^U	d^L	d^U	d^L	d^U
F_5^A	5	5	1	0.0	0.0	80	80	1	1	21	21
F_{10}^A	10	10	1	0.0	0.0	130	130	1	1	21	21
F_{15}^A	15	15	1	0.0	0.0	180	180	1	1	21	21
F_5^B	5	5	1	0.0	0.0	1300	23	1	1	58	7
F_{10}^B	10	10	1	24.7	0.0	1.5e+6	38	81	1	42K	7
F_{15}^B	15	15	1	37.2	0.0	1.5e+6	53	277	1	112K	7
M_1	3	4	11	0.4	0.5	31927	39288	10	10	1349	1695
M_2	4	4	13	2.8	40.0	203051	3.4e+6	17	150	7695	183K
M_3	4	4	13	2.2	1.5	174655	130580	14	14	5690	5412
M_4	5	4	15	6.8	65.7	579494	6.0e+6	33	445	25K	668K
N_1	3	7	11	1.6	1.3	42931	36858	10	10	1803	1601
N_2	4	7	13	9.1	124	264930	5.1e+6	20	289	9279	366K
N_3	4	7	13	4.8	77.4	134798	2.6e+6	19	218	11K	127K
N_4	5	7	15	49.4	181	1.5e+6	6.7e+6	74	234	41K	127K
C_1	5	3	12	0.2	0.2	19263	19628	10	10	977	987
C_2	6	3	14	0.5	0.4	68070	60618	12	12	1501	830
C_3	6	3	15	0.7	0.6	97733	86474	14	14	1238	856
C_4	7	3	17	6.3	5.6	979581	854090	47	45	4510	1906
C_5	8	3	19	61.7	58.6	8.8e+6	8.3e+6	306	306	12K	8943
C_6	9	3	21	–	–	–	–	–	–	–	–

Table 2. Experimental results for greedy search with d^L and d^U. Abbreviations as in Table 1.

We use three variants of the Fischer protocol for mutual exclusion. The examples are "F_i^X" in the tables, where X is A, B, or C, and i is the number of parallel automata. The error condition is that at least two of the automata are 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. Variant C introduces a variable specifying the number of automata in the error location.

[5] For very small values of N, around 10000, we observed many "outliers", i.e., examples where search took several orders of magnitude shorter or longer when using the bitstate hashing. For larger N values, the behaviour becomes more stable, and most of the time gives a speedup factor of around 2 to 10 in our examples.

					t			S			M		l
Exp	a	c	v	BF	h^L	d^L	BF	h^L	d^L	BF	h^L	d^L	
F_5^A	5	5	1	0.0	0.0	0.0	1467	207	1457	6	1	1	9
F_{10}^A	10	10	1	0.5	0.0	0.6	37942	2022	37922	8	1	8	9
F_{15}^A	15	15	1	7.8	0.3	7.8	348827	9187	348797	31	10	32	9
F_5^B	5	5	1	0.0	0.0	0.0	362	138	242	1	1	1	7
F_{10}^B	10	10	1	0.0	0.0	0.0	5422	1768	2352	1	1	1	7
F_{15}^B	15	15	1	0.6	0.2	0.2	34307	8648	10437	7	11	6	7
F_5^C	5	5	2	0.0	0.0	na	362	130	na	1	1	na	7
F_{10}^C	10	10	2	0.0	0.0	na	5442	755	na	1	1	na	7
F_{15}^C	15	15	2	0.6	0.0	na	34307	2255	na	7	1	na	7
M_1	3	4	11	0.8	0.3	0.8	50001	24035	50147	7	7	7	50
M_2	4	4	13	3.1	1.4	3.4	223662	101253	223034	11	10	10	51
M_3	4	4	13	3.3	1.6	3.4	234587	115008	231357	11	10	10	53
M_4	5	4	15	13.6	6.4	14.5	990513	468127	971736	29	22	25	54
N_1	3	7	11	5.2	3.2	5.6	100183	59573	99840	9	9	8	50
N_2	4	7	13	25.6	15.1	25.5	442556	273235	446465	18	15	15	53
N_3	4	7	13	26.4	16.7	27.2	476622	301963	473117	17	15	15	53
N_4	5	7	15	120	77.4	119	2.0e+6	1.3e+6	2.0e+6	65	39	45	56
C_1	5	3	12	0.3	0.7	0.3	35325	17570	35768	7	9	7	55
C_2	6	3	14	0.9	1.7	1.0	109583	46495	110593	10	12	10	55
C_3	6	3	15	1.2	2.1	1.3	143013	53081	144199	11	13	11	55
C_4	7	3	17	10.8	16.9	12.2	1.4e+6	451755	1.4e+6	78	49	51	56
C_5	8	3	19	114	128	123	1.2e+7	3.4e+6	1.2e+7	574	322	377	57
C_6	9	3	21	–	1328	–	–	3.2e+7	–	–	2.7G	–	57

Table 3. Experimental results for our optimal configurations, i.e., BF, A^* search with h^L, and A^* search with d^L. Abbreviations as in Table 1, na means not applicable.

The other examples in the tables are from two more realistic case studies. Examples "M_i" and "N_i", $i = 1, \ldots, 4$, come from a study called "Mutual Exclusion". This study models a real-time protocol to ensure mutual exclusion of states in a distributed system via asynchronous communication. The protocol is described in full detail in [6]. 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. The resulting models do not have many automata but a non-trivial amount of clocks and variables.

Examples "C_i", $i = 1, \ldots, 9$, come from a case study called "Single-tracked Line Segment". This study stems from an industrial project partner of the UniForM-project [16] 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 [16, 6], an automata-like notation for real-time programs. The PLC-Automata were translated into timed automata with the tool Moby/RT [17]. The property to be checked requires that never both directions are given permission to enter the shared segment simultaneously. This property is ensured by 3 PLC-Automata of the whole controller. We injected an error by manipulating a delay such that the asynchronous communication between these automata is faulty. In Moby/RT abstractions are offered for the translation into the timed automata. The given set of PLC-Automata had eight input variables and we constructed nine models with decreasing size by abstracting more and more of these inputs.

The results in Tables 1 and 2 clearly demonstrate the potential of our heuristic functions. Consider Table 1 first. Except in F_i^B (where h^L behaves very badly), and F_i^C (where no approach needs any time), the heuristic searches consistently find the error paths much faster. Due to the reduced search space size and memory requirements, they can solve more of the large C_i examples. At the same time, they find *much*, by orders of magnitude, shorter error paths in *all* cases. In F_i^B, h^L does worse than h^U because its heuristic value does not improve if only one of the two target automata moves closer to its destination: the ATG becomes shorter only if both get closer. The somewhat odd behaviour of h^L in C_8, where search is a lot faster than in C_9, is an outlier caused by the bitstate hashing (outliers suggest a direction for future work discussed in Section 7).

Considering Table 2, we observe that, using d^L and d^U in greedy search, except in the Fischer variants the search space sizes and runtimes one gets are similar to that of rDF, in most cases somewhat worse. The error paths are longer (up to two orders of magnitude) than those found by rDF, except in Fischer variant A. The heuristics can't handle Fischer variant C – the target condition is not expressed in terms of target *locations* – which is, for that reason, left out of the table. In variant B, similarly to h^L, d^L fails quickly. In variant A, due to the construction both d^L and d^U are constantly 1, and the search spaces are identical to those of a non-randomised DF.

The results for the optimal configurations, Table 3, demonstrate that h^L also has some potential to improve the finding of optimal error paths, if to a lesser extent than in the sub-optimal setting. A^* with h^L has the smallest search spaces in all cases, and the best runtimes in all cases except the large C_i examples, *of which it can solve more than the other configurations due to the lower memory requirements.* The d^L heuristic, on the other hand, most of the time yields performance very similar to that of BF. None of the configurations could solve C_7, C_8, or C_9.

6 Related Work

The published approaches to directed model-checking all differ from ours either in that the heuristic has to be provided by the user, or in that the heuristic is based on a very different kind of reasoning.

Bloem et al [4] describe a mechanism how to model check ECTL and ACTL formulas. The method computes least and greatest fixpoints by under and over approximations based on *hints* provided by the user. Apart from relying on the user, this method differs from ours in that it can treat more general formulas, and does not do a heuristic search. Behrmann et al [3] have studied *priced* timed automata. Transitions are labelled with prices, and a heuristic estimates the remaining costs. Behrmann et al achieved good results in an application for which they hand-coded the heuristic; they don't provide an automatic computation.

Yang and Dill [20] use Hamming distance to drive a heuristic search. This is generally a much cruder approximation than our ATG-based heuristics (with the advantage of taking much less time to compute). We implemented the Hamming distance heuristic in UPPAAL, and found it to not work well in our examples: roughly similar to d^L and d^U in the Fischer examples, by far the worst heuristic (much worse runtime results) in the M_i, N_i, and C_i examples. Groce and Visser [12] introduce two heuristics, inspired by the area of testing, for model checking Java programs. The heuristics do not try to target an error formula but instead drive the search to cover yet unexplored branches in the program. Edelkamp et al [10] introduced heuristics to improve error detection with SPIN. As discussed earlier, we implemented these heuristics (d^L and d^U) in UPPAAL and found them to not work very well in our context. Qian and Nymeyer [18] introduced the use of "pattern database" heuristics based on abstractions generated by ignoring some of the state variables. This is a very different abstraction technique than ours, which keeps all variables, and, instead, simplifies their semantics.

In parallel to ours, related work is done by Dräger et al [7]. A paper is submitted to this same conference. The two pieces of work are conducted (and submitted) separately because, like in the works listed above, *the techniques used to generate the heuristic functions are fundamentally different.* While we approach from an AI Planning perspective, Dräger et al modify established abstraction methods from Verification. While we developed combined treatments of communication and integer variables, their focus so far is (almost) exclusively on finding good approximations of communication, particularly of cyclic patterns. Treating integer variables in Dräger et al's approach appears non-trivial, and has not yet been done. Their approximation works by, in a pre-process, iteratively "merging" a pair of automata, i.e., by computing their product and then merging locations until there are at most N locations left, where N is an input parameter. The resulting heuristic has, in difference to ours, no trouble with the communication structure depicted in Figure 1 (Section 4) – however, when merging locations one runs the risk to lose the distinction between dead ends and non dead ends in Figure 3. Indeed, in that example, UPPAAL excels with our heuristics but doesn't scale with Dräger et al's; in Towers of Hanoi – an example containing excessively many repetitions in its solution – the picture is exactly inverse. As more realistic examples, we shared the M_i, N_i, and C_i benchmarks. While these have communication structures more like Figure 1, they also rely heavily on integer variables. The results for the two different heuristics are roughly comparable. There are advantages for h^L in the M_i and N_i benchmarks, and advantages for Dräger et al's heuristic in the C_i benchmarks except C_6, C_7, and C_8. Investigating combinations of the two approaches – e.g., using our approach to treat integers in Dräger et al's approach – is future work.

7 Conclusion

We have introduced methods for automatically generating two heuristic guidance functions in UPPAAL. We have shown the functions' potential for yielding more reliable finding of error states, by reducing the number of search states that need to be considered, as well as guiding the search to short error paths.

The most pressing research topic right now is how to take clock variables into account in the heuristic computation. As said, a straightforward treatment is very unlikely to yield any useful information. We think there is hope in, when building the ATG, distinguishing between the clock value subsets that can be reached *at the individual automaton locations*. Due to location invariants restricting the passage of time, the intervals possible at individual locations are more restricted than the "global" reachable interval. Particularly, constraints on how one clock value can change due to a transition often transfer to all other clocks as well since for them time elapses in the same way. (As a simple example, if one steps from l to l' and $x \leq 5$ is an invariant for l', then we know that the maximum reachable value for any clock is at most 5 larger than it was in l.) In a similar fashion, we hope to make the treatment of integer variables more informed by distinguishing between the value subsets that can be reached at individual locations.

In the long term, we want to explore the following two directions. First, the "outliers" – instances solved in extremely short time – observed with very small hash tables in bitstate hashing suggest that *randomised local search with restarts* might be suitable. Such methods do gradient descents on the search space surface, with random perturbations, until either a solution is reached or a termination criterion (e.g. path length bound exceeded) holds, and a restart is made. We take the existence of outliers to indicate that there is a good enough chance for such gradient descents to find shallow solutions. Second, we believe there is hope in generating heuristic functions based on *predicate abstractions*: these could take the clocks into account very naturally.

References

1. Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL implementation secrets. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002.
2. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL, 2005. Department of Computer Science, Aalborg University, Denmark.
3. Gerd Behrmann and Ansgar Fehnker. Efficient guiding towards cost-optimality in UPPAAL. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–188, London, UK, 2001. Springer-Verlag.
4. Roderick Bloem, Kavita Ravi, and Fabio Somenzi. Symbolic guided search for CTL model checking. In *Proceedings of the 37th conference on Design automation*, pages 29–34, New York, NY, USA, 2000. ACM Press.

5. Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.
6. H. Dierks. Comparing model-checking and logical reasoning for real-time systems. *Formal Aspects of Computing*, 16(2):104–120, May 2004.
7. Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In *13th International SPIN Workshop on Model Checking of Software (SPIN'2006)*, 2006.
8. Stefan Edelkamp. Generalizing the relaxed planning heuristic to non-linear tasks. In S. Biundo, T. Frühwirth, and Günther Palm, editors, *KI-04: Advances in Artificial Intelligence*, pages 198–212, Ulm, Germany, 2004. Springer-Verlag.
9. Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit model checking with hsf-spin. In *Proc. of the 8th International SPIN Workshop on Model Checking of Software (SPIN'2001)*, pages 57–79, 2001.
10. Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology*, 2004.
11. Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research*, 20:239–290, 2003.
12. Alex Groce and Willem Visser. Model checking Java programs using structural heuristics. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 12–21, New York, NY, USA, 2002. ACM Press.
13. Jörg Hoffmann. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.
14. Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
15. G. Holzmann. *The Spin Model Checker - Primer and Reference Manual.* Addison-Wesley, 2003.
16. B. Krieg-Brückner, J. Peleska, E.-R. Olderog, and A. Baer. The UniForM Workbench, a universal development environment for formal methods. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods*, volume 1709 of *LNCS*, pages 1186–1205. Springer, 1999.
17. E.-R. Olderog and H. Dierks. Moby/RT: A tool for specification and verification of real-time systems. *Journal of Universal Computer Science*, 9(2):88–105, February 2003.
18. Kairong Qian and Albert 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, Berlin, Heidelberg, 2004. Springer-Verlag.
19. Benjamin Wah and Yixin Chen. Subgoal partitioning and global search for solving temporal planning problems in mixed space. *International Journal of Artificial Intelligence Tools*, 13(4):767–790, 2004.
20. C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Proceedings of the 35th annual conference on Design automation*, pages 599–604, New York, NY, USA, 1998. ACM Press.

A Additional Information

A.1 Proofs

We list the omitted proofs in chronological order, including the respective claims.

Proposition 1 *Given a network of timed automata with binary synchronisation and integer variables, a start state, target locations, and a target formula. If* $t_1, \ldots, t_n$ *is a solution then* $t_1, \ldots, t_n$ *is also an abstract solution.*

Proof: Denote by s_i, $0 \leq i \leq n$ the state after execution of transitions $t_1, \ldots, t_i$. We show by induction over i that $s_i(j) \in s_i^+(j)$ for all automata j, and $s_i(v) \in s_i^+(v)$ for all integer variables v. This suffices to prove the claim. It is obvious for $i = 0$. If it holds for i, then transition t_{i+1} is enabled in s_i^+. The new location and integer variable values resulting from executing t_{i+1} are, by definition, inserted into the respective s_{i+1}^+ value subsets. E.g. for an assignment $v := v + 1$ we have $s_{i+1}(v) = s_i(v) + 1$, $s_i(v) \in s_i^+(v)$ by induction hypothesis, and $s_i(v) + 1 \in \{c + 1 \mid c \in s^+(v)\} \subseteq s_{i+1}^+$. ■

Lemma 1 *Given a network of timed automata with binary synchronisation and integer variables, a start state, target locations, and a target formula. If there is an abstract solution of length n, then the algorithm in Figure 2 stops successfully in an iteration minlayer* $\leq n$*.*

Proof: Say $t_1, \ldots, t_n$ is an abstract solution. We prove that, when running the algorithm without the stopping criteria, in iteration n the target locations and target formula will be reached. This suffices because, obviously, if the negative stopping criterion holds in an earlier iteration $m < n$ then there is a fixpoint: $L_k(i) = L_m(i)$ for all i and $k > m$, and $V_k(v) = V_m(v)$ for all v and $k > m$, yielding a contradiction.

Denote by s_k^+, $0 \leq k \leq n$ the state after abstract execution of transitions $t_1, \ldots, t_k$ in the start state. We show by induction over k that s_k^+ is contained in L_k and V_k, i.e. for all i $s_k^+(i) \subseteq L_k(i)$ and for all v $s_k^+(v) \subseteq L_k(v)$. This suffices to prove the overall claim. The induction base case is obvious. If the induction hypothesis holds for k, then t_{k+1} is enabled by L_k and V_k, so it is processed in the inner loop. The locations added by t_{k+1} into s_{k+1}, and the values added by its $v := c$ effects, are thus inserted into L_{k+1} and V_{k+1} by construction. The values reached by the transition's $v := v'$ effects are inserted into V_{k+1} because $s_k^+(v') \subseteq V_k(v')$ by induction hypothesis. Similarly, for $v := v + 1$ and $v := v - 1$ effects, by induction hypothesis $s_k^+(v) \subseteq V_k(v)$ and so the values inserted into L_{k+1} and V_{k+1} in particular contain the values inserted by t_{k+1} into s_{k+1}^+ with these effects. ■

Proposition 2 *Let TASolMin*$^+$ *denote the following problem. Given a network of timed automata with binary synchronisation and integer variables, a start* *state, a target formula, and an integer b. Is there a abstract solution of length at most b?*

TASolMin$^+$ *is* ***NP**-hard.*

Proof: By a reduction from 3SAT. Assume a 3-CNF formula with n variables v_i and m clauses $c_j = \{c_{j1}, c_{j2}, c_{j3}\}$ where $c_{jl} \in \{v_1, -v_1, \ldots, v_n, -v_n\}$. In our construction there are n components corresponding to the variables, and m components corresponding to the clauses. The components are depicted in Figure 4. From the construction, it is obvious that there is an abstract solution of length at most $2n + 2m$ iff the CNF is satisfiable. All variables have to be assigned a value by choosing a path to the goal location in the respective component. All clause components have to select one satisfied literal (one var assigned the right value) and take the respective path to the goal location. $2n + 2m$ steps are just enough to do so, and there is no time to "cheat" and let one variable component execute both its truth-value actions.

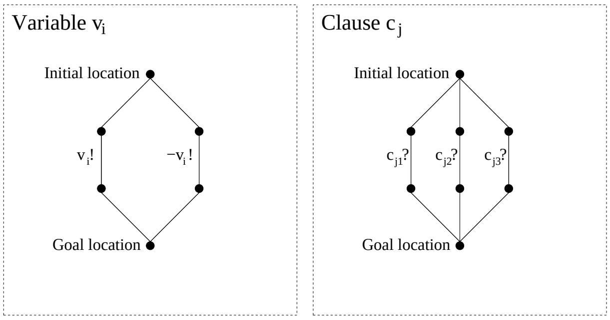

Fig. 4. Reduction of 3SAT to bounded abstract location vector reachability.

■

A.2 Selecting Abstract Solutions from an ATG

From a successfully built ATG, one can easily select a abstract solution. The algorithm is in Figure 5.

The algorithm makes use of location sets $TL_k(i)$ and of variable value sets $TV_k(v)$. These sets, at an ATG layer k, contain the current target locations and variable values at layer k. The sets are initialised as empty, then the target locations and formula of the overall task are inserted by the make-target subroutine. That routine is explained in detail below. Transitions supporting the targets in the $TL_k(i)$ and $TV_k(v)$ sets are selected during a backwards loop over k from the top ATG layer to layer 1. The start locations and guard formulas of the selected transitions are inserted into the TL and TV sets; by construction of the ATG, these new targets will appear at layers below k (see also below). Where necessary, it is ensured that the transition effects achieve the desired target. Selecting a transition n times means to select a respective repetitive sequence.

```
TL_k(i) := ∅ for all i and 0 ≤ k ≤ finallayer
TV_k(v) := ∅ for all v and 0 ≤ k ≤ finallayer
make-target(finallayer, target locations, target formula)
for k := finallayer, ..., 1 do
    for all i, all l'(i) ∈ TL_k(i) do
        select t enabled at k − 1 that ends in l'
        make-target(k − 1, t start locations, t guard formulas)
    endfor
    for all v, c ∈ TV_k(v) do
        if ex. t enabled at k − 1 with effect v := c then
          select one such t
          make-target(k − 1, t start locations, t guard formulas)
        elseif ex. t enabled at k − 1 with effect v := v' and c ∈ V_{k−1}(v') then
              select one such t
              TV_{k−1}(v') := TV_{k−1}(v') ∪ {c}
              make-target(k − 1, t start locations, t guard formulas)
        elseif ex. c' < c ∈ V_{k−1}(v), and t enabled at k − 1 with effect v := v + 1 then
              choose one such c and t, select t (c − c') times
              TV_{k−1}(v) := TV_{k−1}(v) ∪ {c'}
              make-target(k − 1, t start locations, t guard formulas)
        elseif ex. c' > c ∈ V_{k−1}(v), and t enabled at k − 1 with effect v := v − 1 then
              choose one such c and t, select t (c' − c) times
              TV_{k−1}(v) := TV_{k−1}(v) ∪ {c'}
              make-target(k − 1, t start locations, t guard formulas)
        endif
    endfor
endfor
```

Fig. 5. Extracting a abstract solution from a abstract transition graph.

The make-target sub-routine takes as arguments a number m, a set of locations, and a set of formulas (the sets contain 1 or 2 elements each). The routine first determines, for each location l in automaton i, the lowest k such that $l \in L_k(i)$, and sets $TL_k(i) := TL_k(i) \cup \{l\}$. Note that $k \leq m$ will hold by construction, i.e. the new location targets are inserted at layers below in the graph (otherwise the transition would not be enabled at m). Next, the routine iteratively processes all conditions in the formulas. For each condition, a value (a value pair) is chosen that is contained in V_m. For each chosen value c of variable v, the lowest k with $c \in V_k(v)$ is determined, and $TV_k(v) := TV_k(v) \cup \{c\}$ is set. Note again that $k \leq m$ will hold.

Proposition 3. *Given a network of timed automata with binary synchronisation and integer variables, a start state, target locations, and a target formula, so that the algorithm in Figure 2 stops with success. Then the transitions selected by the algorithm in Figure 5 form an abstract solution.*

Proof: First, note that by construction of the abstract transition graph, the algorithm can not fail, i.e., there is always a transition sufficient to support a target location or variable value. We form the abstract solution by arranging the transitions in inverse order of selection, i.e., in particular from bottom $k = 0$ to top $k = finallayer - 1$. We show by induction over k that the start locations, and the guards, of the transitions are satisfied in the respective abstract state of execution. For $k = 0$ this is obvious. For $k > 0$ it follows because the start locations and guards of the transitions were posted as targets at layers below, and thus achieved by the respective selected supporting transitions (which are enabled by induction hypothesis). So all selected transitions will be enabled in the sequence, thus achieving the (global) target locations and formula that were posted at the start of the algorithm. ■

The algorithm depicted in figure 5 does not deal with linear arithmetic, but it can be easily extended to do so. Whenever a transition for the abstract plan is selected that has a linear guard – i.e. a guard of the form $\sum_i c_i \cdot v_i \bowtie c$ where $c, c_i \in \mathbb{Z}$ and v_i are integer variables – we have to make sure that this guard is satisfied. In order to achieve this, we simply add its guard in the corresponding layer (via the make-target sub-routine). The interesting part is the selection of the linear guard. As we know that every guard of any transition in the ATG is satisfied, we have to find a value for each variable that occurs in the guard such that the guard is satisfied. For every such variable value pair we add a transition that assigns this value to the variable to the relaxed plan.