# Bounding and heuristics in forward reachability algorithms

A. Fehnker

Computing Science Institute/

**CSI-R0002 February**

Computing Science Institute Nijmegen
Faculty of Mathematics and Informatics
Catholic University of Nijmegen
Toernooiveld 1
6525 ED  Nijmegen
The Netherlands

# Bounding and heuristics in forward reachability algorithms

Ansgar Fehnker*
Computing Science Institute Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
ansgar@cs.kun.nl

**Abstract**

Recently timed automata models have been used to solve realistic scheduling problems. In this paper we want to establish the relation between timed automata and job shop scheduling problems. The timed automata models of the scheduling problems can serve as input for a forward reachability checker. In contrast to job shop algorithms the forward reachability algorithms will usually not yield an optimal solution. There are also only few ways to direct the exploration of the state space. Starting from job shop problem we will describe how forward reachability can be equipped with two concepts from branch and bound methods: *heuristics* and *bounding*. This extended algorithm is then applicable to all kinds of timed automata models.

KEYWORDS AND PHRASES: Timed automata, Static Scheduling, Reachability, Model Checking, UPPAAL, Branch and Bound Algorithms, Job Shop, Heuristics
AMS SUBJECT CLASSIFICATION: 68M14, 68W20, 90B35, 90B90
CR SUBJECT CLASSIFICATION:D.2.2., D.2.4, F.1.1, F.3.1, I.2.2, I.2.8

## 1 Introduction

Reachability analysis of timed automata models has recently been used to solve scheduling problems. A. Fehnker used this approach [8] to compute a feasible schedule for the SIDMAR steel plant. Th. Hune, K.G. Larsen and P. Pettersson [10] showed for a physical model of the plant how to restrict the state space effectively and how to translate the generated schedules to control programs automatically. P. Niebert and S. Yovine used a similar approach in [13] to determine schedules for the Dortmund University experimental batch plant.

*Research supported by Netherlands Organization for Scientific Research (NWO) under contract SION 612-14-004, and by the European Community Esprit-LTR Project 26270 VHS (Verification of Hybrid systems)

They were able to compute optimal schedules, which improve over currently used ad hoc schedules.

During the discussion of these results, the question arose how the model checking approach compares to branch and bound methods for job shop scheduling. Branch and bound methods have been applied to numerous optimization problems like the traveling salesman problem, integer programming and of course scheduling problems. Branch and bound methods are in essence enumeration schemes for solving optimization problems. The set of possible solutions is divided in subsets. By certain rules subsets that do not contain an optimal solution are identified and eliminated. The applicability of these methods derive from the fact that in many cases only a small fraction of possible solutions has to be enumerated.

Reachability algorithms are methods to search for an error state by enumerating all reachable (symbolic) states. Starting from an initial configuration reachable states are computed till either an error state has been found or a fixpoint has been reached. When checking for an error state a diagnostic trace can be generated that shows how to reach the error state.

The purpose of this paper is two-fold. First, we want to define job shop problems as networks of timed automata. Next, we want to extend the reachability algorithm in a way that allows us to direct the search and find optimal solutions. The job shop problem will be defined in section 2, timed automata in section 3. In section 4 we will introduce the timed automata model and show that it is sound and complete. Some of the problems that arise when we use model checkers to derive schedules are illustrated in section 5 for a small example. We then will review basic concepts of branch and bound algorithms, and derive a branch and bound algorithm for timed automata. In section 6 we give some remarks on a future implementation.

## 2 The Job Shop Problem

The *job shop scheduling* problem is to optimally schedule sets of jobs on a set of machines. Each job is a chain of operations, and machines can only process a limited number of operations at a time. The purpose is to allocate starting times to the operation, such that the maximal completion time is minimal. The job shop problem is known to be (NP-)hard [9, p. 242]. Many solutions methods such as local search algorithms like simulated annealing or tabu search [1], and shifting bottleneck algorithms [3] have been proposed.

**Definition 2.1** A job shop $\mathcal{P}$ is a tuple $(\mathcal{J}, \mathcal{O}, \mathcal{M}, j, d, m, c, \prec)$ where

$\mathcal{J}$ is a finite set of jobs,

$\mathcal{O}$ is a finite set of operations,

$\mathcal{M}$ is a finite set of machines,

$j : \mathcal{O} \to \mathcal{J}$ gives for each operation the job it belongs to,

$d : \mathcal{O} \to \mathbb{N}\backslash 0$ defines the duration of each operation,

$m : \mathcal{O} \to \mathcal{M}$ gives the machine on which the operation has to be performed,

$c : \mathcal{M} \to \mathbb{N}$ defines the capacity of the machines, and finally

$\prec$ is a partial order on the set of operations that satisfies

$$\forall o, p \in \mathcal{O}, o \neq p.\ (o \prec p \vee p \prec o) \Leftrightarrow j(o) = j(p) \tag{1}$$

Equivalence (1) states that all operations of the same job are totally ordered, and that there is no precedence between operations of different jobs. In contrast with the definitions of job shops in [3, 6, 17] we do not require that the maximal capacity of machines is one. We also allow several operations of the same job to be processed on the same machine, as long as the other constraints hold. The definition of the general job shop in [16] and [7] do not require (1) to hold, and cover a larger class of problems.

**Definition 2.2** Let $\mathcal{P}$ be a job shop. A schedule of $\mathcal{P}$ is a function $S : \mathcal{O} \to \mathbb{N}$ that defines the starting time of each operation. A schedule $S$ is feasible if it satisfies, for all $o, p \in \mathcal{O}$

$$o \prec p \Rightarrow S(o) + d(o) \leq S(p) \tag{2}$$

$$\#\{p \in \mathcal{O} | m(p) = m(o) \wedge S(p) \leq S(o) < S(p) + d(p)\} \leq c(m(o)) \tag{3}$$

We write $\mathcal{F}(\mathcal{P})$ for the set of all feasible schedules.

Thus, a schedule is feasible if operations of the same job are not processed at the same time (2), and the capacity of the machines is not exceeded (3). The latter is guaranteed by (3), because the number of operations that is processed whenever a operation starts does not exceed the capacity of the machine.

**Definition 2.3** Let $\mathcal{P} = (\mathcal{J}, \mathcal{O}, \mathcal{M}, j, d, m, c, \prec)$ be a job shop. The job shop problem is to find a feasible schedule $S \in \mathcal{F}(\mathcal{P})$ such that

$$\max_{o \in \mathcal{O}}(S(o) + d(o)) = \min_{S' \in \mathcal{F}(\mathcal{P})} \max_{o \in \mathcal{O}}(S'(o) + d(o)) \tag{4}$$

i.e. $S$ is a schedule with minimal makespan.

## 3 Timed Automata

Timed Automata have proven to be a useful formalism to model and verify real-time systems. Timed Automata, due to Alur and Dill [2], are finite state automata with clock variables. The formalism can be used to model real-time requirements of systems in a natural way. In recent years several tools for automatic model checking based on timed automata have become available, such as UPPAAL [11] and KRONOS [18]. In this paper we will work with networks of timed automata as defined in [5, 11] to model the job shop problem.

## Network of Timed Automata

A timed automaton is a finite automaton over a set of labels $Act$, equipped with a finite set of clocks $C$, whose value increase uniformly with time. Labels are either local or synchronizing. If label $a$ is synchronizing it has a complement $\bar{a}$, which in turn has as complement $\bar{\bar{a}} = a$. A clock constraint $\phi$ is generated by the grammar

$$\phi := x < c | x - y < c | x = c | x - y = c | x > c | x - y > c | \neg\phi | \phi_1 \wedge \phi_2$$

with $x, y \in C$ and $c \in \mathbb{R}_{\geq 0}$. We denote the language generated by this grammar with $\mathcal{B}(C)$.

A *network of timed automata* is the parallel composition $A_1 | \ldots | A_n$ of the timed automata $A_1, \ldots, A_n$ over clocks $C$ and labels $Act$. Each automaton $A_i$ is defined by a set $L_i$ of control locations, an initial location $l_i^0$, a set $E_i$ of edges of the form $e = (l_i, \phi, a, \rho, l_i')$ with $l_i, l_i' \in L_i$, guard $\phi \in \mathcal{B}(C)$, $a \in Act$, a reset set $\rho \subseteq C$, and finally an invariant $I_i : L_i \to \mathcal{B}(C)$. Edge $(l_i, \phi, a, \rho, l_i') \in E_i$ means that automaton $A_i$ has a transition from control location $l_i$ to $l_i'$, which will be denoted as $l_i \xrightarrow{\phi, a, \rho} l_i'$.

The state of a network of timed automata is a pair $(l, v)$, where $l$ is the vector of control locations and $v : C \to \mathbb{R}_{\geq 0}$ a valuation of the clocks. We use $v \models \phi$ to denote that the clock valuation $v$ satisfies the clock constraint $\phi \in \mathcal{B}(C)$. We will denote the $i$-th element of the control vector with $l_i$, and the control vector where the $i$-th element $l_i$ is replaced by $l_i'$ with $l[l_i'/l_i]$. The initial state is defined by control vector $l^0 = (l_0^0, \ldots, l_n^0)$ and $v^0(x) := 0, \forall x \in C$. The invariant $I(l)$ of control location $l$ is the conjunction of the invariants $I_i(l_i)$.

The network of timed automata then has two types of transitions:

- a local transition $l \xrightarrow{\phi, a, \rho} l'$ if there exist transition $l_i \xrightarrow{\phi, a, \rho} l_i'$ with local label $a$ of automaton $A_i$ such that $l' = l[l_i'/l_i]$
- a synchronizing transition $l \xrightarrow{\phi, a, \rho} l'$ if there exists a transition $l_i \xrightarrow{\phi_i, a, \rho_i} l_i'$ with synchronizing label $a$ of automaton $A_i$ and a transition $l_j \xrightarrow{\phi_j, \bar{a}, \rho_j} l_j'$ of automaton $A_j$ such that $\phi = \phi_i \wedge \phi_j$, $\rho = \rho_i \cup \rho_j$ and $l' = l[l_i'/l_i][l_j'/l_j]$.

A transition $l \xrightarrow{\phi, a, \rho} l'$ from state $(l, v)$ to $(l', v')$ can be taken if $v \models I(l)$, $v' \models I(l')$, $v \models \phi$, and $v'(x) = 0$ if $x \in \rho$, and $v'(x) = v(x)$ otherwise. We will denote transitions from $(l, v)$ to $(l', v')$ with label $a$ by $(l, v) \xrightarrow{a} (l', v')$.

In state $(l, v)$ of the network a delay of $d \in \mathbb{R}_{\geq 0}$ may take place if $\forall 0 \leq t \leq d.\ v + t \models I(l)$. Here, we use $v + t$ to denote the valuation that maps clocks $x \in C$ to $v(x) + t$. Thus, a delay may occur as long as the invariant in that location holds. We will denote delays by $(l, v) \xrightarrow{d} (l, v + d)$.

## Executions

A sequence of delays and transitions $(l^0, v^0) \xrightarrow{d^0} (l^0, v^0 + d^0) \xrightarrow{a^1} (l^1, v^1) \ldots$ with initial state $(l^0, v^0)$ is called *execution*. State $(l, v)$ is called *reachable* if there exists a finite execution with final state $(l, v)$. We denote the final state of a execution $\alpha$ with $\alpha.(l, v)$. For $\alpha, \beta \in exec(\mathbf{TA})$ we use the notation $\alpha \sqsubseteq \beta$ if $\alpha$ is a prefix of $\beta$. We write $exec(\mathbf{TA})$ for the set of (finite) executions of a timed automaton $\mathbf{TA}$. The *span* of a finite execution is defined as the finite sum $\sum_{i \geq 0} d_i$ of the delays.

### Symbolic States

The prior definition of networks of timed automata gives rise to an infinite transition system. To be able to use verification algorithms a finite *symbolic* semantics based on *symbolic* states $(l, \Delta)$ with $\Delta \in \mathcal{B}(C)$ is used. It can be shown that for symbolic state $(l, \Delta)$ and transition $l \xrightarrow{\phi, a, \rho} l'$ there exists a symbolic state $(l', \Delta')$ such that

$$v' \models \Delta' \Leftrightarrow \exists v \models \Delta, d \in \mathbb{R}_{\geq 0}\,.\, (l, v) \xrightarrow{a} (l', v' - d) \xrightarrow{d} (l', v') \tag{5}$$

We then call symbolic state $(l', \Delta')$ the successor of $(l, \Delta)$ and denote this relation by $(l, \Delta) \overset{a}{\Rightarrow} (l', \Delta')$. It can be shown that $(l, v)$ is reachable if and only if $(l^0, \Delta^0) \overset{a^1}{\Rightarrow} \ldots \overset{a^n}{\Rightarrow} (l, \Delta)$ for some $\Delta$ such that $v \models \Delta$; with initial state $(l^0, \Delta^0)$ such that for all $v \models \Delta^0$ there exists a delay $(l^0, v^0) \xrightarrow{d} (l^0, v)$. For a proof see e.g. [5].

## 4 Timed Automata Model

### The Model

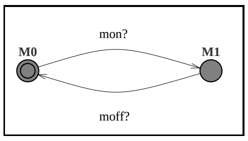
Let $\mathcal{P} = (\mathcal{J}, \mathcal{O}, \mathcal{M}, j, d, m, c, \prec)$ be a job shop. We model $\mathcal{P}$ as a network of timed automata. For each job $J \in \mathcal{J}$ we include a timed automaton $\mathbf{J}$, which will be constructed as follows. Let $\mathbf{j}$ be the number of operations $o \in \mathcal{O}$ with $j(o) = J$. Let $o_1, \ldots, o_{\mathbf{j}}$ be the operations of $J$ ordered with respect to to $\prec$. Then we will construct an automaton with control locations $s_0, \ldots, s_{\mathbf{j}}$ and $t_1, \ldots, t_{\mathbf{j}}$. The automaton will be in location $s_i$ after operation $o_i$ and before $o_{i+1}$, and in location $t_i$ while operation $o_i$ is processed. To time the duration of the operations we include clock $clock_J$. We have for each machine two labels in set $L_J$. One to model the beginning of an operations on that machine and one to model the end of an operation. For convenience we define two injective mappings $on : \mathcal{M} \to L_J$ and $\mathit{off} : \mathcal{M} \to L_J$ with $on(\mathcal{M}) \cap \mathit{off}(\mathcal{M}) = \emptyset$.

We then have for each operation $o_i \in \mathcal{O}$ the following transitions:

$$\begin{array}{rcl} s_{i-1} & \xrightarrow{on(m(o_i)), \{clock_J\}} & t_i \\ t_i & \xrightarrow{clock_J \geq d(o_i), \mathit{off}(m(o_i))} & s_i \end{array} \tag{6}$$

We further assume that in location $t_i$ the invariant $clock_J \leq d(o_i)$ holds.

For each machine $M$ we include a timed automaton $\mathbf{M}$. Let $\mathbf{m} = c(M)$. $\mathbf{M}$ has locations $s_0, \ldots, s_{\mathbf{m}}$ to count the number of jobs being processed on



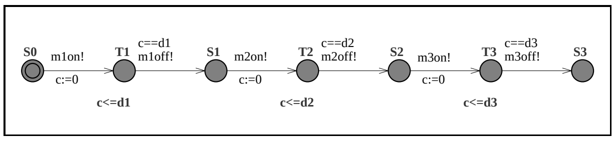


Figure 1: The left figure shows a timed automaton modeling a machine with capacity 1. The right figure shows how to model a job with three operations, and durations `d1`, `d2`, `d3`.

that machine. The timed automaton $\mathbf{M}$ then has the following transitions, for $i \in 1, \ldots, \mathbf{m}$:

$$\begin{aligned} u_{i-1} &\xrightarrow{\overline{on(M)}} u_i \\ u_i &\xrightarrow{\overline{off(M)}} u_{i-1} \end{aligned} \tag{7}$$

We write $\mathbf{P}$ for the composition automaton $\mathbf{J}_1| \ldots |\mathbf{J}_n| \, \mathbf{M}_1| \ldots |\mathbf{M}_m$ that models job shop $\mathcal{P}$. Table 1 shows how to model the Fisher and Thomson problem *mt06*, a job shop problem with 6 machines and 6 jobs, as network of timed automata in Uppaal.

The final states of automaton $\mathbf{P}$ are states $(l, \Delta)$ that satisfy $l_{\mathbf{J}_i} = \mathbf{J}_i.s_{\mathbf{j}_i}$, $i \in 1, \ldots, n$. In such states the automaton is deadlocked, and only time can elapse without bound. We will denote the set of executions that end in a final state with $cexec(\mathbf{P})$. Since the job automata are linear we can conclude that the composition automaton can only reach a final state if and only if the execution contains a $on(m(o))$ and $off(m(o))$ transition for each operation $o \in \mathcal{O}$.

## Soundness and Completeness

**Definition 4.1** Let $\mathcal{P}$ be a job shop and $\mathbf{P}$ the corresponding composition automaton. Let $S : \mathcal{O} \to \mathbb{N}$ be a schedule and $\alpha \in cexec(\mathbf{P})$. We will say that $\alpha$ *corresponds* with $S$, denoted as $\alpha \sim S$, if for all $o \in \mathcal{O}$ there exists a $\alpha' \xrightarrow{on(m(o))} (l'', v'') \sqsubseteq \alpha$ such that for some locations $s_i, t_i, u_k$.

$$l'' = \alpha'.l[j(o).t_i/j(o).s_{i-1}][m(o).u_k/m(o).u_{k-1}] \;\wedge\; S(o) = span(\alpha') \tag{8}$$

**Lemma 4.2** *Let $\mathcal{P}$ be a job shop and $\mathbf{P}$ the corresponding timed automaton. Let $\alpha \in cexec(\mathbf{P})$ and $S : \mathcal{O} \to \mathbb{N}$ such that $\alpha \sim S$. Then $S \in \mathcal{F}(\mathcal{P})$.*

The schedule $S$ obtained from the execution $\alpha$ by assigning to each operation the span of the corresponding prefix has to satisfy (2). This is guaranteed by the construction of the automata modeling the jobs (6). The constraints (3) hold, since the timed automata that model the machines (7) do not allow that they exceed their capacity.

```
chan on0,on1,on2,on3,on4,on5,off0,off1,off2,off3,off4,off5;
process machine(chan mon; chan moff){
state s0, s1;
init s0;
trans u0 -> u1 {sync mon?; },
u1 -> u0 {sync moff?; };
}
process job(chan m1on,m1off;const d1;chan m2on,m2off;const d2;
chan m3on,m3off;const d3;chan m4on,m4off;const d4;
chan m5on,m5off;const d5;chan m6on,m6off;const d6){
clock c;
state s0,s1,s2,s3,s4,s5,s6,
t1{c<=d1},t2{c<=d2},t3{c<=d3},t4{c<=d4},t5{c<=d5},t6{c<=d6};
init s0;
trans s0 -> t1 {sync m1on!; assign c:=0; },
s1 -> t2 {sync m2on!; assign c:=0; },
s2 -> t3 {sync m3on!; assign c:=0; },
s3 -> t4 {sync m4on!; assign c:=0; },
s4 -> t5 {sync m5on!; assign c:=0; },
s5 -> t6 {sync m6on!; assign c:=0; },
t1 -> s1 {guard c==d1; sync m1off!; },
t2 -> s2 {guard c==d2; sync m2off!; },
t3 -> s3 {guard c==d3; sync m3off!; },
t4 -> s4 {guard c==d4; sync m4off!; },
t5 -> s5 {guard c==d5; sync m5off!; },
t6 -> s6 {guard c==d6; sync m6off!; };
}
M0:=machine(on0,off0);
M1:=machine(on1,off1);
M2:=machine(on2,off2);
M3:=machine(on3,off3);
M4:=machine(on4,off4);
M5:=machine(on5,off5);
//
// Fisher and Thompson 6x6 instance, alternate name (mt06)
// 6 6
// 2  1  0  3  1  6  3  7  5  3  4  6
// 1  8  2  5  4 10  5 10  0 10  3  4
// 2  5  3  4  5  8  0  9  1  1  4  7
// 1  5  0  5  2  5  3  3  4  8  5  9
// 2  9  1  3  4  5  5  4  0  3  3  1
// 1  3  3  3  5  9  0 10  4  4  2  1
//
J0:=job(on2,off2,1,on0,off0,3,on1,off1, 6,on3,off3, 7,on5,off5, 3,on4,off4,6);
J1:=job(on1,off1,8,on2,off2,5,on4,off4,10,on5,off5,10,on0,off0,10,on3,off3,4);
J2:=job(on2,off2,5,on3,off3,4,on5,off5, 8,on0,off0, 9,on1,off1, 1,on4,off4,7);
J3:=job(on1,off1,5,on0,off0,5,on2,off2, 5,on3,off3, 3,on4,off4, 8,on5,off5,9);
J4:=job(on2,off2,9,on1,off1,3,on4,off4, 5,on5,off5, 4,on0,off0, 3,on3,off3,1);
J5:=job(on1,off1,3,on3,off3,3,on5,off5, 9,on0,off0,10,on4,off4, 4,on2,off2,1);
system J0,J1,J2,J3,J4,J5,M0,M1,M2,M3,M4,M5;
```

Table 1: Uppaal model of the Fisher and Thompson problem *mt06*

**Lemma 4.3** *Let $\mathcal{P}$ be a job shop and $\mathbf{P}$ the corresponding timed automaton. Each feasible schedule $S \in \mathcal{F}(\mathcal{P})$ corresponds to an execution $\alpha \in cexec(\mathbf{P})$ with $\alpha \sim S$.*

*Proof.* Let $S$ be a schedule. We construct a execution as follows. We start with two queues $P = p_1, \dots, p_n$ and $Q = q_1, \dots, q_n$, each containing all operations ordered with respect to $S(p_i)$ and $S(p_i) + d(p_i)$, respectively.

Starting from the initial state $(l^0, v^0)$ we take transitions depending on the heads of the queues. If $S(p_1) < S(q_1) + d(q_1)$, we remove $p_1$ from queue $P$ and can take, according to (6) and (7), a transition $on(m(p_1))$ from $(l, v)$ to $(l[j(p_1).t_i/j(p_1).s_{i-1}][m(p_1).u_{k+1}/m(p_1).u_k], v')$. Constraint (3) guarantees that we can take the *on*-transitions. If $S(p_1) \geq S(q_1) + d(q_1)$, then $q_1$ will be removed from $Q$ and transition $\mathit{off}(m(p_1))$ to a corresponding state will be taken. We iterate this step until both queues are empty.

We end up with an execution that contains for each operation a *on* and *off*-transition. We can therefore conclude that $l_i = \mathbf{J}_i.s_{\mathbf{j}_i}$ holds for $1 \leq i \leq n$.

## 5 Algorithms

### Scheduling Problems and Model Checking

Model checkers for timed automata have recently been used to find feasible schedules for a steel plant [8] and an experimental batch plant [13]. Forward reachability algorithms were applied in these cases to decide whether a final state in which the jobs were completed was reachable. In case such a state was found the model checking tool produced a diagnostic trace, which was used to build a schedule.

The generation of schedules for the steel plant revealed numerous problems. We illustrate the main problems for a small job shop problem. We use Uppaal to find a feasible schedule for the Fisher and Thompson problem *mt06* (Table 1), and ask the verifier to search for a state that satisfies $\Phi = \bigwedge_{i=0,\dots,5} \mathtt{J}i\mathtt{.s6}$.

The Uppaal verifier (version 3.0.16) allows, like most other on-the-fly checkers, two strategies to search the state space. Depending on whether the list of encountered but unexplored states is a stack or a queue we are searching depth-first or breadth-first, respectively. The breadth-first strategy is not able to find a trace for the *mt06*-problem, due to limited memory. The depth-first strategy yields a schedule with completion time 193. This is at the same time the worst possible schedule, since the sum of all durations of the *mt06*-problem is 193. This solution is obtained by scheduling first all operations of one job, then all operations of another job, etcetera.

The Uppaal simulator allows us to apply random search. In this case the simulator selects at random a state from the last generated set of successor states. We apply this strategy to the *mt06*-problem which has an optimal solution with makespan 55. The model has 12 synchronizing transitions for each job automaton. Since these automata have no cycles it is guaranteed that the

72nd successor of the initial state corresponds to a feasible schedule, no matter how we search for a feasible solution. The generated schedules during 10 experiments with random search had an average completion time of 94.5. The maximal makespan was 120, the minimal makespan was 87. The schedule with the makespan 120 was the only one that took longer than 100 time units.

A different heuristic to find better feasible schedule uses Uppaals concept of urgent transitions. We declare all *on* transitions urgent, i.e. they are taken as soon as possible. This heuristic will not always lead to an optimal solution, since not only unnecessary delays but also useful delays might be omitted. The verifier found with a depth-first search for this greedy approach a trial schedule with makespan 70.

We combined this approach with some other heuristics that estimated the makespan and pruned branches with an estimated makespan bigger than a given upper bound by adding guards. We then used the verifier to produce iteratively schedules with a smaller makespan by decreasing the upper bound. We were able to find within 3.5 seconds cpu-time on a Sun Ultra-5 a schedule with makespan 57. Uppaals verifier proved (within 18 seconds) that there was no schedule with a makespan smaller than 57. In our attempts to bound the search space by introducing urgency we obviously pruned also the branch that contains the optimal solution with makespan 55.

In the remainder of this paper we want to modify the model checking algorithm such that it allows us to use other strategies than depth-first and breadth-first. The search space has to be bounded, without eliminating optimal solutions. Finally we want the algorithm to search for the optimum automatically.

## Basic Rules of Branch and Bound Algorithms

Branch and bound algorithms search in a tree structure for feasible solutions with minimal costs. What solutions are considered feasible and how to compute the costs depends on the application area. Branch and bound algorithms can be characterized by four basic rules [12, 15].

1. The branching rule defines how to calculate the successors of a node in the search tree. The successors will be added to the waiting list.

2. The selection rule defines which state in the waiting list to branch from next. Heuristic functions can be used to cover all possible search strategies. The heuristic function assigns to each element of the waiting list a heuristic value. The algorithm then selects and removes the element with the smallest heuristic value. The heuristic function can be a probabilistic function that assigns a heuristic value with a certain distribution.

3. The bounding rule defines how to compute a lower bound on the costs of the complete solutions that can be obtained from a partial solution. This lower bound allows to prune branches with a lower bound larger than the cost of the best solution found so far. A tight lower bound in combination with a good trial solution can bound the search space effectively.

4. The elimination rule defines how to recognize branches that cannot contain an optimal solution. A partial solution can be eliminated if is dominated by another partial solution, this means it can be proven that the costs of the best completion of the second is less or equal to the costs of the best completion of the first. The dominance relation should be transitive and consistent with the lower bound. The lower bound on the costs of partial solution has to be equal or greater than the lower bound of the dominating problem.

## Branch and Bound for Timed Automata

To exploit the ideas from branch and bound algorithms for timed automata we introduce lower bounds and heuristic values. We extend the symbolic states with a lower bound and a heuristic value to $(l, \Delta, lb, heu) \in L \times \mathcal{B}(C) \times \mathbb{Z} \times \mathbb{Z}$. The cost function is defined as a function $f : L \times \mathcal{B}(C) \times \mathbb{Z} \to \mathbb{Z}$ on location, clock constraints and lower bound. We define the lower bound function $g : \mathbb{Z} \times E \times \mathcal{B}(C) \to \mathbb{Z}$ recursively as function of the lower bound in the source location, a transition, and the clock constraints in the target location.

The heuristic function $h : \mathbb{Z} \times E \times \mathcal{B}(C) \to \mathbb{Z}$ is defined similarly as function of heuristic value, a transition, and clock constraints. The initial values $lb^0$ and $heu^0$ are commonly zero, but may be set arbitrarily.

The state $(l', \Delta', lb', heu')$ is a successor of state $(l, \Delta, lb, heu)$ if there exists a transition $l \xrightarrow{\phi,a,\rho} l'$ such that $(l', \Delta') \overset{a}{\Rightarrow} (l', \Delta')$, $lb' = g(lb, l \xrightarrow{\phi,a,\rho} l', \Delta')$ and $heu' = h(heu, l \xrightarrow{\phi,a,\rho} l', \Delta')$. We denote this relation with $(l, \Delta, lb, heu) \overset{a}{\Rightarrow} (l', \Delta', lb', heu')$. Additionally, we assume

$$lb \leq g(lb, l \xrightarrow{\phi,a,\rho} l', \Delta') \tag{9}$$

$$f(l', \Delta', lb') = g(lb, l \xrightarrow{\phi,a,\rho} l', \Delta') \qquad \text{if } (l', \Delta') \models \Phi \tag{10}$$

The lower bound on the cost should not decrease as we take a transition, and it should coincide with the actual cost for a complete solution.

We do not want to restrict the choice of the heuristic function. Heuristic functions can postpone exploration of a branch but do not prune solutions. This can influence the performance negatively as well as positively but does not harm the correctness of the algorithm. The breadth-first strategy can be realized by a heuristic function that assigns the number of transitions that has been taken so far to each node. Depth-first, in contrast, takes the negative number of transitions. Also random searches can be useful, as shown for *mt06* example. Another heuristic that has proved to be useful for error detection (in an untimed system) is the distance to a distinguished error state [14].

Finally we require for a transition $l \xrightarrow{\phi,a,\rho} l'$, lower bounds $lb_1 \leq lb_2$ and clock constraints $\Delta_1 \Leftarrow \Delta_2$:

$$g(lb_1, l \xrightarrow{\phi,a,\rho} l', \Delta_1) \leq g(lb_2, l \xrightarrow{\phi,a,\rho} l', \Delta_2) \tag{11}$$

```
passed := {}, trial :=$\perp$, $f^* := \infty$
if $(l^0, \Delta^0) \models \Phi$ then trial := $(l^0, \Delta^0)$, $f^* := f(l^0, \Delta^0, lb))$, waiting := []
else waiting := $[(l^0, \Delta^0, lb^0, heu^0)]$ fi
while waiting $\neq$ [] do
    get $(l, \Delta, lb, heu)$ from waiting
    if $\neg(\exists(l', \Delta', lb') \in$ passed. $l' = l \wedge \Delta' \Leftarrow \Delta \wedge lb' \leq lb)$
    then add $(l, \Delta, lb)$ to passed
          succ:=$\{(l', \Delta', lb', heu')|(l, \Delta, lb, heu) \overset{a}{\Rightarrow} (l', \Delta', lb', heu')\}$
          forall $(l, \Delta, lb, heu) \in$ succ do
              if $lb < f^*$
              then if $(l, \Delta) \models \Phi$
                      then trial := $(l, \Delta)$
                              $f^* := f(l, \Delta, lb)$
                      else put $(l, \Delta, lb, heu)$ in waiting fi
              fi
          od
    fi
od
return trial,$f^*$
```

Table 2: Branch and Bound Algorithm based on extended states. The function get selects and removes the extended state from the waiting list with the smallest heuristic value.

This ensures that tighter clock constraint do not decrease the lower bound. The lower bound based on a conjunction of clock constraints should be at least as small as the lowest lower bound of each of the parts.

We use (11) to define an elimination rule. Suppose we have a two states $(l, \Delta, lb, heu)$ and $(l, \Delta', lb', heu')$ with $\Delta \Leftarrow \Delta'$ and $lb \leq lb'$. Then there is no need to explore $(l, \Delta', lb', heu')$, since all its successor yield, due to (10) and (11), a smaller lower bound and consequently lower costs. With these definitions for $f, g, h$ and the extended states and the elimination rule we get the branch and bound algorithm depicted in Table 2.

## 6 Conclusion and Future Work

In this section we want to describe briefly how a implementation could be realized. It is not the intention to get an implementation that supports all kinds of cost functions and heuristics. Our aim is to restrict the class of costs and heuristics such that a working implementation can easily be obtained from an existing model checker. Particular choices are motivated by experiences with case study 5 of the VHS project and experiments with small job shop problems

The implementation should allow to define costs by decorating transitions with assignments on distinctive integer variables. Besides to an integer variable for the lower bound, auxiliary variables to compute the lower bound should be allowed. We will treat the auxiliary costs like a lower bound, to ensure that (11) holds.

The heuristic function should be defined in a way similar to the cost function by decorating transitions. We also allow auxiliary heuristic functions as scratch pad to compute the heuristic value. Since the heuristic values should only be used to sort the waiting list, there are almost no restrictions on the assignments to (auxiliary) heuristic variables, except that computing the successor should not be too expensive. To support randomized search it should be possible to use a function that returns random integers.

The heuristic should be defined by a tuple of heuristic values. The function `get` in Table 2 selects the symbolic state that is the smallest in the first heuristic value. If a second heuristic value is defined, it will be used to solve ties. In case that there are still ties a third heuristic value can be used, etcetera. If no heuristic value is defined it might be set to zero by default. To ease modeling it should be possible to define standard choices like FIFO, LIFO or random selection by using keywords.

The depth-first strategy can be realized by a LIFO rule that decreases the heuristic value with every transition. The algorithm searches breadth-first if it increases this value. A randomized depth-first search can be obtained by decreasing the heuristic value with each transition and select then randomly among states with the same value. A randomized breadth-first strategy in contrast increases the heuristic value with each transition. If we use a pure random selection rule we can expect a search strategy in between depth and breadth-first.

We do not expect that the proposed algorithm can significantly outperform job shop algorithms in solving standard job shop problems. Most job shop algorithms consider for example only *left justified* (or *semi-active*) schedules, which reduces the size of the search space effectively. A schedule is left-justified if no operation can be completed earlier without changing the order of the operations on any machine or delaying other operations. It has been proven, that there always exists a left justified optimal schedule. It is however often not obvious how to apply these techniques to other that job shop problems.

The proposed algorithm is applicable to networks of timed automata. Applying a general purpose algorithm to a special problem area usually involves some overhead. Still, it makes branch and bound methods applicable to a range of problems other than scheduling, like error detection in concurrent systems.

Heuristics allow strategies other than depth or breadth-first, and are more flexible than guiding strategies that bound the search space explicitly by adding guards [10]. Recent work by G. Behrmann, Th. Hune and F. Vaandrager on a parallel model checker for timed automata confirms the need for branch and bound techniques in model checking [4]. The parallel model checker can not ensure a certain search strategy, though one may want to find a error state with the shortest trace. The combination of bounding and branching can then guarantee to find the shortest trace, even if the search strategy is not breadth-first.

**Acknowledgment** Many ideas that are developed in this paper were influenced by discussions with other VHS participants. I would like to mention in particular Paul Petterson, Peter Niebert, Thomas Hune and Gerd Behrmann.

## References

[1] E. Aarts, P. van Laarhoven, J. Lenstra, and N. Ulder. A Computational Study of Local Search Algorithms for Job-Shop Scheduling. *OSRA Journal on Computing*, 6(2):118–125, Spring 1994.

[2] R. Alur and D. Dill. A Theory of Timed Automata. *Theoret. Comput. Sci.*, 126:183–235, 1994.

[3] D. Applegate and W. Cook. A Computational Study of the Job-Shop Scheduling Problem. *OSRA Journal on Computing 3*, pages 149–156, 1991.

[4] G. Behrmann, T. Hune, and F. Vaandrager. Distributing Timed Model Checking – How the search order matters. Technical report, Computing Science Institute, Nijmegen, 2000.

[5] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial order reductions for timed systems. In *CONCUR'98*, LNCS 1427, pages 485–500, Vancouver, Canada, June 1998. Springer.

[6] J. Carlier and E. Pinson. An Algorithm for Solving the Job-Shop Problem. *Management Science*, 35(2):164–176, 1989.

[7] A. Dolzman, T. Sturm, and V. Weispfennig. Real Quatifier Elimination in Practice. In *School on Computational Aspects and Applications of Hybrid Systems*, 1998. Proc KIT Workshop on Verifications of Hybrid systems.

[8] A. Fehnker. Scheduling a Steel Plant with Timed Automata. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, 1999.

[9] M. Garey and D. S. Johnson. *Computers and Intractability, A guide to the Theory of NP-Completeness.* W.H. Freeman and Company, San Francisco, 1979.

[10] T. Hune, K. G. Larsen, and P. Pettersson. Guided synthesis of control programs using UPPAAL for VHS case study 5. VHS deliverable, 1999.

[11] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1+2):134–152, 1997.

[12] L. Mitten. Branch-and-bound Methods: General Formulation and Properties. *Operations Research*, 18:24–34, 1970.

[13] P. Niebert and S. Yovine. Computing Optimal Operation Schemes for Multi Batch Operation of Chemical Plants, May 1999. VHS deliverable.

[14] F. Reffel and S. Edelkamp. Error detection with directed symbolic model checking. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods*, number 1708 in LNCS, pages 195–211, 1999.

[15] H. W. J. M. Trienekens. *Parallel Branch and Bound Algorithms.* PhD thesis, Erasmus Universiteit Rotterdam, 1990.

[16] R. Vaessens. *Generalized Job Shop Scheduling: Complexity and Local Search.* PhD thesis, Eindhoven University of Technology, 1995.

[17] R. Vaessens, E. Aarts, and J. Lenstra. Job shop scheduling by local search. *INFORMS Journal on Computing*, 8:302–317, 1996.

[18] S. Yovine. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1/2,), October 1997.