Schedulability Analysis with Variable Computation Time of Tasks*

Martin Stigge
Uppsala University[†], Sweden
Humboldt University Berlin[‡], Germany
Email: mstigge@informatik.hu-berlin.de

June 11, 2007

Abstract

The goal of this project is to study the borderline of decidability for schedulability analysis problems. Task automata (i.e., timed automata extended with tasks) have been introduced to model complex non-deterministic patterns of task releases and executions in real-time systems. For task automata it has been shown that the schedulability analysis problem is undecidable for almost all scheduling policies when the following three conditions hold: (1) tasks may preempt other already running tasks, (2) the execution times of tasks are variable within intervals, and (3) the precise finishing time of a task may influence new task releases. It has also been shown that the problem becomes decidable if conditions (1) or (2) are dropped. The first technical result of this report is to show that the problem is still undecidable if just (3) is dropped, but in turn replaced by extensions of the model like adding memory to the scheduler or applying the model to a multi-processor environment. Without these extensions, the decidability is unknown. The second technical result of this report is to develop an approximation algorithm for the schedulability problem in this case. We conjecture that the algorithm is exact, meaning that it solves the schedulability problem.

1 Introduction

Real-time systems are usually abstracted as a set of real-time tasks and a scheduling policy which schedules the tasks to the available computation hardware. The real-time tasks are often described using a set of parameters such as release patterns, execution time behaviour, priorities, deadlines etc. One important question in the analysis of these models is, whether the real-time tasks

*Parts of this work were submitted to FORMATS'07, see [KSY07].
[†]Supervisors: Pavel Krcal and Prof. Wang Yi, {pavelk,yi}@it.uu.se
[‡]Examinor: Prof. Johannes Köbler, koebler@informatik.hu-berlin.de

can be scheduled on the processor(s) in a way that they finish their executions within their deadlines. This is called *schedulability analysis*.

Classic approaches to schedulability analysis are often based on static and deterministic task parameters like release rates, while non-deterministic behaviour (like jitters) are only allowed in a very narrow and controlled way. But these solutions, e.g., Rate-Monotonic Analysis ([LL73]), often lack flexibility in expressing non-determinism of a greater scale. In recent years, timed automata ([AD94]) have established as a standard model to describe non-deterministic timing behaviour of systems. Analysis methods and tools for timed automata were developed and made it also a good candidate for modelling task release patterns and analysing schedulability problems.

One way of doing this is to extend the model by asynchronous tasks ([FKPY07]). The locations of such a *task automaton* are associated with task releases. When the automaton visits a location during a run, the associated tasks are released and inserted into a queue by a scheduler, waiting for serial processing. Since the model includes an explicit notion of time, one can formulate the *schedulability problem* for task automata as the property that during all runs of an automaton, all tasks will finish their execution within their given deadlines.

Models of real-time systems may have the following three important properties, and it turned out that the decidability of the schedulability problem depends on them:

(1) The scheduler runs a preemptive scheduling policy,

(2) Tasks may finish their execution at a non-deterministic time within a given interval between best- and worst-case computation time, and

(3) The precise finishing times of tasks may influence the releases of other tasks.

In [FKPY07] it is shown that for the general class of models where all three conditions may hold, the schedulability problem is undecidable, but it is decidable if (1) or (2) are dropped. It is still an open question if the problem is decidable for the single-processor setting if just (3) is dropped. The goal of this project is to study this borderline. This report presents two technical results:

(a) a proof that dropping (3) still retains undecidability if the scheduler may use memory or we move to a multi-processor setting, and

(b) an approximation algorithm for the single-processor case with a stateless scheduler. We conjecture that this algorithm is exact, s.t. it solves the schedulability problem if property (3) is dropped.

The report is structured in the following way: Section 2 introduces the underlaying models, problems and known results. Section 3 shows how the feedback property can be dropped in the former undecidability results and be replaced by either memory in the scheduler or the use of multiple queues. In Section 4 we develop an approximation algorithm for the original setting using Clock Difference Relations (CDRs). Section 5 concludes the report.

2 Preliminaries: Models and results

In this section, we introduce the well-known concept of Timed Automata and the concept of Task Automata developed in [FKPY07], together with the known decidability results for these models. A reader familiar with these concepts might as well skip to the next section and refer back just to consult the definitions. We also use some new visualisation methods (region diagrams) for a more intuitive way of expressing clock regions for timed automata.

2.1 Timed Automata

Timed Automata are a well-studied model for the behaviour of real-time systems, first introduced in [AD94]. We just give basic descriptions and definitions here.

A timed automaton is a finite state automaton extended with a finite set of clocks. These clocks take values from $\mathbb{R}_{\geq 0}$, and to model time progress, the clock values are increased during a run of the automaton. Transitions may reset clock values to 0 and the values can be used to restrict location-changing transitions.

We give now the basic definitions. Let $\mathcal{C}$ be the set of clocks, then a function $\nu : \mathcal{C} \to \mathbb{R}_{\geq 0}$ is called a *clock valuation* while the set of all clock valuations over the clocks $\mathcal{C}$ is denoted by $\mathcal{V}(\mathcal{C})$. With $\nu[r]$ we denote the clock valuation which is equal to ν, except that all clocks in $r \subseteq \mathcal{C}$ are being reset. $\mathcal{B}(\mathcal{C})$ is the set of *clock guards* g, which are conjunctions of expressions $x_1 \bowtie N$ and $x_1 - x_2 \bowtie N$ with $x_1, x_2 \in \mathcal{C}$, $N \in \mathbb{N}_{\geq 0}$ and $\bowtie \in \{<, =, >\}$. If g contains only $x_1 \bowtie N$ expressions, we say it is *diagonal-free*. A valuation ν is said to be *consistent* with a guard g (written $\nu \models g$) if for all expressions $x_1 \bowtie N$ and $x_1 - x_2 \bowtie N$ in g it holds that $\nu(x_1) \bowtie N$ and $\nu(x_1) - \nu(x_2) \bowtie N$, respectively.

Definition 1. *A* timed automaton *over actions $\mathcal{A}ct$ and clocks $\mathcal{C}$ is a tuple $\langle N, l_0, E, I\rangle$ where*

- *N is a finite set of locations,*
- *$l_0 \in N$ is the initial location,*
- *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \mathcal{A}ct \times 2^{\mathcal{C}} \times N$ is the set of edges and*
- *$I : N \to \mathcal{B}(\mathcal{C})$ is a function assigning a clock constraint to each location which is called* location invariant.

We write $l \xrightarrow{g\,a\,r} l'$ for $\langle l, g, a, r, l'\rangle \in E$.

Semantics: Such a timed automaton has a state consisting of a location and a clock valuation, thus the state space is $S := N \times \mathcal{V}(\mathcal{C})$. It may perform two types of transitions. An *event transition* changes the location according to an edge in the automaton, if the condition in the guard is fulfilled. Additionally, the clocks stated in the edge are reset. A *time pass transition* in turn leaves the

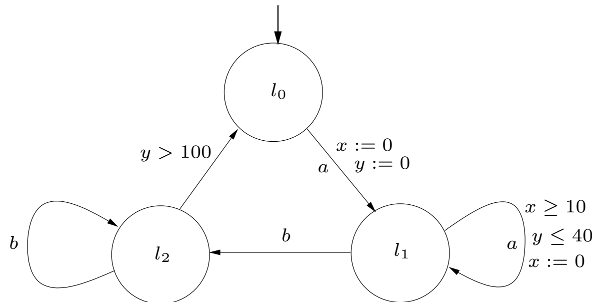

Figure 1: Example for a (diagonal-free) timed automaton.

location constant but adds a certain real value to all clock values in the clock valuation of the state.

This leads usually to a definition like the following, where ν_0 is such that $\nu_0(x) = 0$ for all clocks $x \in \mathcal{C}$ and the set of labels $\Sigma := \mathcal{A}ct \cup \mathbb{R}_{\geq 0}$ is a union of all possible events and time pass values.

Definition 2. *The* semantics of a timed automaton $A = \langle N, l_0, E, I \rangle$ *is a labeled transition system* $[\![A]\!] = \langle S, s_0, \Sigma, T \rangle$ *with states* S*, initial state* $s_0 = (l_0, \nu_0)$*, labels* Σ *and transitions* T *defined by the following rules:*

- $(l, \nu) \xrightarrow{a} (l', \nu[r])$ *if* $l \xrightarrow{g\,a\,r} l'$, $\nu \models g$, *and* $\nu[r] \models I(l')$,
- $(l, \nu) \xrightarrow{t} (l, \nu + t)$ *if* $t \in \mathbb{R}_{\geq 0}$ *and* $(\nu + t) \models I(l)$.

Note that since we are only interested in reachability problems in this work, we do not define an accepted language or other objects usually linked to such types of automata.

2.2 Decidability for Timed Automata

To decide questions like the reachability of states, the infinite state space of the timed automaton has to be reduced to a finite one. The usual and well-known approach to this is the construction of the *region graph*. This finite graph is defined using an equivalence relation with finite index on the state space, and this relation is a bisimulation. The key observation for this is, that since all guards (on locations and edges) are just direct comparisons of clock values to each other or to natural numbers, certain sets of clock valuations will "behave the same" in the semantics. This is meant in the sense that they enable exactly the same transitions – and that their successor states will also retain this property. In particular, two clock valuations are considered to be *region-equivalent*, when the integral parts and the order of the fractional parts of all values are the same below a certain threshold, and both are consistent with a given set of diagonal constraints.

For a $t \in \mathbb{R}$, we will use $\langle t \rangle$ to denote the fractional part of t, and $\lfloor t \rfloor$ for its integral part (thus $t = \langle t \rangle + \lfloor t \rfloor$).

Definition 3. *(Region equivalence) For a clock $x \in \mathcal{C}$ let C_x be a natural number and let $u, v \in \mathcal{V}$ and $\mathcal{G}$ be a finite set of diagonal constraints of the form $x - y \bowtie N$ where $N \in \mathbb{N}_{\geq 0}$. Then u and v are* region-equivalent *$(u \sim v)$ iff:*

1. *For each clock x, either $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or $u(x) > C_x$ and $v(x) > C_x$,*
2. *For all clocks x, y it holds that if $u(x) \leq C_x$ and $u(y) \leq C_y$ then*
   (a) *$\langle u(x) \rangle = 0 \iff \langle v(x) \rangle = 0$ and*
   (b) *$\langle u(x) \rangle \leq \langle u(y) \rangle \iff \langle v(x) \rangle \leq \langle v(y) \rangle$, and*
3. *$u \models g \iff v \models g$ for all $g \in \mathcal{G}$.*

Note that the last condition is an extension from [BY04] of the widely used basic region equivalence from [AD94] and is needed to support diagonal constraints in guards. (The general reachability results from [AD94] also hold for this extension.)

This relation is obviously an equivalence relation, and the equivalence classes called *clock regions* can be described as:

1. a map, assigning an integer to each clock and each clock difference (with a special symbol for clock differences or clocks x having a value greater than the constant C_x), and
2. a set of relations between the fractional parts of the clock values.

We call this relation the *fractional part* of the region information, in contrast to the map which we call the *discrete part*. Both views – set of equivalent valuations and its description as map/relations – will be used equivalently because they describe the same object.

The description of a clock region by the order of the fractional parts plus the integral part for each clock can be visualised by a *region diagram* as shown in Figure 2. Note that the numbers can be omitted when we only deal with the fractional part of the region.

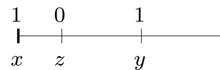

Figure 2: A region diagram for a region D where $\lfloor D(x) \rfloor = \lfloor D(y) \rfloor = 1$, $\lfloor D(z) \rfloor = 0$ and $0 = \langle D(x) \rangle < \langle D(z) \rangle < \langle D(y) \rangle$. Note that the left end (marked with a bold line) has a special meaning, since clocks there have integral values, like x in this example.

The number of these clock regions is obviously finite, and thus they are used to construct the quotient system called *region graph*. The states from

$S := N \times \mathcal{R}(\mathcal{C})$ are tuples of a location and a clock region, and the set of labels $\Sigma := \mathcal{A}ct \cup \{t\}$ is a union of all possible events and a special time pass symbol. For a region D let $\mathsf{TimeSuccessors}(D)$ be the set of all regions D' such that for all $\nu \in D$ there exists $t \in \mathbb{R}_{\geq 0}$ such that $\nu + t \in D'$.

Definition 4. *The* region graph *of a timed automaton $A = \langle N, l_0, E, I \rangle$ is a labeled transition system $[\![A]\!]_{\mathsf{reg}} = \langle S, s_0, \Sigma, T \rangle$ with the states S, initial state s_0, labels Σ and transitions T. The initial state is $s_0 = (l_0, D_0)$ with a start location l_0 and the initial clock region $D_0 = \{\nu_0\}$. The transitions T are defined by the following rules:*

- $(l, D) \xrightarrow{a} (l', D[r])$ *if* $l \xrightarrow{g\,a\,r} l'$, $D \models g$, *and* $D[r] \models I(l')$,
- $(l, D) \xrightarrow{t} (l, D')$ *if* $D' \in \mathsf{TimeSuccessors}(D)$ *and* $D' \models I(l)$.

As shown in [AD94], this region graph is a *time-abstract bisimulation* of the LTS corresponding to the semantics of the associated timed automaton. And since it has a finite state space, it can be easily searched for reachable states. This makes the region graph a key tool for deciding the reachability problem for timed automata and will be used as a basis in Section 4 where we refine this construction.

2.3 Task Automata

The Timed Automata model can now be used to model task releases. This is useful for schedulability analysis, where the release of non-periodic tasks with non-deterministic behavior can be modeled this way and schedulability properties can be checked.

In [FKPY07] the Task Automata model is introduced as an extension of timed automata with task releases, and we summarize the most important aspects here. Each location of the timed automaton can be associated with the release of a task (which is an abstraction of a set of executable processes). The intuition is that as soon as the automaton visits this location, an instance of the associated task is released. This means that a scheduler (modeled by a function) puts the task into a queue for serial processing on a single-processor system. Tasks can have different types and different instances of the same type can be in the queue at the same time. Task instances have properties like remaining best and worst case execution times and a relative deadline. These can be used by the scheduling function to decide where to place the released instances in the queue. In semantics, the time pass transition is extended to update the time properties of the tasks in the queue, and an additional transition for task finishing is introduced, removing the just finished task from the queue. The underlaying timed automaton may use a dedicated clock x_{done} to synchronize on this type of transition, which introduces the possibility of direct feedback from task finishing to new task releases.

This leads to the following definitions from [FKPY07]. Note that we use $\mathcal{P}$ to denote a finite set modelling the tasks, which is described in greater detail below.

Definition 5. *A* task automaton *over actions $\mathcal{A}ct$, clocks $\mathcal{C}$, and task types $\mathcal{P}$ is a tuple $\langle N, l_0, E, I, M, x_{done}\rangle$ where*

- *$\langle N, l_0, E, I\rangle$ is a timed automaton,*
- *$M : N \hookrightarrow \mathcal{P}$ is a partial function assigning locations with task types,* [1] *and*
- *$x_{done} \in \mathcal{C}$ is the clock which is reset whenever a task finishes.*

Like for timed automata, we write $l \xrightarrow{g\,a\,r} l'$ for $\langle l, g, a, r, l'\rangle \in E$.

Tasks and task queues: We define a task type as a tuple (P, B, W, D) written $P(B, W, D)$ where P is the task name (unique for each task type), $B, W \in \mathbb{N}_{\geq 0}$ the best and worst case calculation times (with $B \leq W$ and $W \geq 1$) and $D \in \mathbb{N}_{\geq 1}$ the relative deadline. A task instance $P_i(b_i, w_i, d_i)$ of type $P_i(B_i, W_i, D_i)$ is a "released copy" of this task type with $b_i, w_i, d_i \in \mathbb{R}$ being the remaining values. A task queue q is a list $[P_1(b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)]$ of task instances (of possibly the same type). By a discrete part of a queue $[P_1(b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)]$ we mean the list of the corresponding task names $[P_1, \ldots, P_n]$ (the information about the remaining computation times and deadline is projected out.) Let $\mathcal{P}$ be the set of task types, then $Q_{\mathcal{P}}$ is the set of queues. We use a function $\mathsf{Run} : Q_{\mathcal{P}} \times \mathbb{R}_{\geq 0} \to Q_{\mathcal{P}}$ which given a non-negative real number t and a task queue q returns the task queue after t time units of execution on a processor. The result of $\mathsf{Run}(q, t)$ for $t \leq w_1$ and $q = [P(b_1, w_1, d_1), Q(b_2, w_2, d_2), \ldots,\ R(b_n, w_n, d_n)]$ is defined as $q' = [P(b_1 - t, w_1 - t, d_1 - t), Q(b_2, w_2, d_2 - t), \ldots,\ R(b_n, w_n, d_n - t)]$. For an empty queue denoted by $[]$ we have $\mathsf{Run}([], t) = []$.

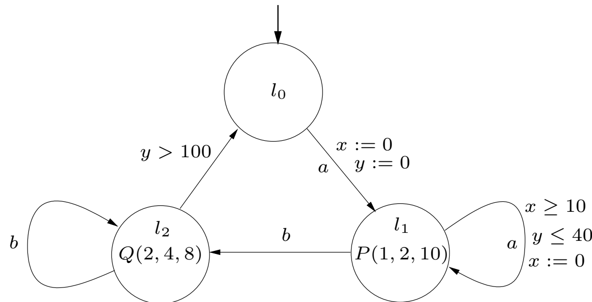

Figure 3: Example for a task automaton for releasing tasks of types Q and P.

An important part of the operational semantics will be the scheduling function $\mathsf{Sch} : \mathcal{P} \times Q_{\mathcal{P}} \to Q_{\mathcal{P}}$. Given a task instance and a task queue, it returns the task queue with the task instance inserted and the order of the other task

[1] Note that M is a partial function meaning that some of the locations may have no tasks.

instances preserved. Depending on whether the scheduler is preemptive or non-preemptive, the function may insert new tasks as the first element or not. Since we want this function to be encodable in timed automata, the definition has the following important condition.

Definition 6. $\mathsf{Sch} : \mathcal{P} \times Q_{\mathcal{P}} \to Q_{\mathcal{P}}$ *is a* scheduling function*, if for each task type* $P(B, W, D)$ *and* discrete part $[P_1, \ldots, P_n]$ *of a queue there can be effectively constructed a diagonal-free timed automaton with*

1. *Clocks* $y_1^b, y_1^w, y_1^d, \ldots, y_n^b, y_n^w, y_n^d$,
2. $n + 2$ *locations* $l_0, l_1, \ldots, l_{n+1}$ *and*
3. $n + 1$ *edges from* l_0 *to* l_i *for* $1 \leq i \leq n + 1$,

such that $\mathsf{Sch}(P(B, W, D), [P_1(b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)])$ *inserts* $P(B, W, D)$ *into the queue at the* k*-th position if and only if* l_k *is the only location reachable from* (l_0, u) *where* $u(y_i^b) = b_i, u(y_i^w) = w_i, u(y_i^d) = d_i$ *for all* $1 \leq i \leq n$.

Note that such a restrictive definition (in particular diagonal-freeness, which is needed for the decidability proof in Section 2.4.2) still allows encoding of strategies like, e.g., Earliest Deadline First (EDF), Fixed Priority Scheduling (FPS) or FIFO, but already prevents strategies where the decisions are based on comparing the time-information of the tasks in the queue to each other (like Least Slack First, LSF). However, LSF only makes sense for non-preemptive scheduling, where the decidability result also holds for weaker scheduler restrictions, as shown in Section 2.4.2.

Semantics: A task automaton may – just as a timed automaton – perform event and delay transitions, and additionally *task finishing* transitions. An event transition corresponds to the arrival of a new task, and a delay transition corresponds to an active task being executed while the others are waiting, or just processor idling in case the queue is empty (therefore, we have two types of delay transitions). The third type of transitions deals with the finishing of a task. We give now the formal definition as a labeled transition system (LTS), where $S := N \times \mathcal{V}(\mathcal{C}) \times Q_{\mathcal{P}}$ is the state space, thus incorporating the queue into the state information. ν_0 is a clock valuation assigning all clocks the value 0, and $\Sigma := \mathcal{A}ct \cup \mathbb{R}_{\geq 0} \cup \{\mathsf{fin}\}$ is a set of labels (for events, time pass values and task finishing).

Definition 7. *Given a scheduling strategy* Sch*, the* semantics of a task automaton automaton $A = \langle N, l_0, E, I, M, x_{done} \rangle$ *is a labeled transition system* $[\![A_{\mathsf{Sch}}]\!] = \langle S, s_0, \Sigma, T \rangle$ *with the states* S*, initial state* s_0*, labels* Σ *and transitions* T*. The initial state is* $s_0 = (l_0, \nu_0, [])$ *and the set* T *of transitions is defined by the following rules:*

- $(l, \nu, q) \xrightarrow{a}_{\mathsf{Sch}} (l', \nu[r], \mathsf{Sch}(M(l'), q))$ *if* $l \xrightarrow{g\,a\,r} l'$, $\nu \models g$, *and* $\nu[r] \models I(l')$,

- $(l, \nu, q) \xrightarrow{t}_{\mathsf{Sch}} (l, \nu + t, \mathsf{Run}(q, t))$ *if* $t \in \mathbb{R}_{\geq 0}$, $(\nu + t) \models I(l)$ *and in case* $q = P(b, w, d) :: q'$ *it holds that* $t \leq w$, *and*

- $(l, \nu, P(b, w, d) :: q) \xrightarrow{\mathsf{fin}}_{\mathsf{Sch}} (l, \nu[x_{done}], q)$ *if* $b \leq 0 \leq w$ *and* $\nu[x_{done}] \models I(l)$,

where $P(b, w, d) :: q$ *denotes the queue with the task instance* $P(b, w, d)$ *on the first position and* q *being the (possibly empty) tail.*

Having defined the semantics of task automata, we can now finally also define the main question we deal with in this work, namely whether they model schedulable releases of tasks. As all deadlines in task automata are hard, we say that a task automaton is schedulable for a given scheduling strategy if no matter how the (non-deterministic) computations evolve, all deadlines are met. We use q_{err} to denote queues containing a task instance $P(b, w, d)$ with $d < 0$.

Definition 8. *(Schedulability) A task automaton A with initial state* $(l_0, u_0, [])$ *is* non-schedulable *with* Sch *if* $(l_0, u_0, []) \longrightarrow^{*}_{\mathsf{Sch}} (l, u, q_{err})$ *for some l and u. Otherwise, we say that A is* schedulable *with* Sch.

We call a queue *non-schedulable* if it will inevitably lead to a deadline miss, provided that all tasks take their worst case computation times. Otherwise a queue is said to be *schedulable*. The important observation for schedulable queues is that their length is bounded ([FKPY07]):

Lemma 1. *Given a task type set $\mathcal{P}$, one can effectively construct a natural number $B_{\mathcal{P}}$ such that $|q| \leq B_{\mathcal{P}}$ for all schedulable queues q.*

As a consequence, there are only finitely many discrete parts of *schedulable* queues for each finite set $\mathcal{P}$ of task types.

2.4 Decidability for Task Automata

A system, modeled by a task automaton and a scheduling function, can have the following three properties:

Preemption: The scheduler may insert a newly released task to the head of the queue, thus preempting all tasks already in the queue (a non-preemptive scheduler may insert only at other positions).

Variable task execution times: It may have task types $P_i(B_i, W_i, D_i)$ such that $B_i < W_i$, meaning that the task may non-deterministically finish execution at a time within the interval $[B_i, W_i]$.

Feedback: The precise finishing time of a task may influence the new task releases (by means of using x_{done} in a guard).

Note that the more of these properties are dropped for a task automaton, the "easier" the problem of schedulability becomes.

In [FKPY07] it was shown that schedulability becomes undecidable if a task automaton has all three properties. In turn, it was also shown that if there is

no preemption, the problem becomes decidable. The same holds if there is no variable execution time. The open question remained, whether schedulability is decidable if x_{done} is not used in the guards for creating feedback. This last variant has also been proven decidable for certain types of schedulers.

We will give now the main ideas behind the known results developed in [FKPY07].

2.4.1 Undecidability

The general problem, whether a given task automaton with a given scheduling strategy is schedulable, is undecidable – provided that all three properties from above may be used in the automaton. The proof of this (given in [FKPY07]) is based on reducing the halting problem for two-counter machines to schedulability of task automata, by constructing a task automaton which simulates a two-counter machine.

A *two-counter machine* consists of a finite state control unit and two unbounded non-negative integer counters which are both initially set to zero. The three possible instructions are counter-increment, counter-decrement and conditional branching (conditioned on checking one of the counters to be zero). After each step, the machine state is changed deterministically. One of the states is a dedicated *halt state*. It is known, that the problem whether this halt state is reachable (the *halting problem for two-counter machines*) is undecidable.

The idea is, given a two-counter machine M, to construct a task automaton A_M such that a dedicated halt location in A_M is reachable if and only if the halt state of M is reachable. It will be ensured that no task can miss its deadline as long as halt is not visited. In turn, the queue can "overflow" in the halt location, making tasks miss their deadlines. The result of these two properties is that A_M will be non-schedulable if and only if M can reach its halt state. This gives the wanted reduction of the schedulability problem to the halting problem for two-counter machines.

In the construction of A_M for a given M, there exists one location l_i in A_M corresponding to each state of M's control unit. These locations are connected depending on the operation which M would execute at the corresponding state (increment, decrement, branch), through auxiliary locations "executing" this instruction.

To encode the counters into clock values, an N*-wrapping* construction from [HKPV98] is used. All clocks x stay within the interval $[0, N]$ for a constant N by resetting each clock x as soon as $x = N$ (*wrapping reset*). For a dedicated system clock x_{sys} these are the only resets (which makes x_{sys} periodic). This way, *wrapping values* for all other clocks can be defined as their values at the (periodic) times where $x_{sys} = 0$. The wrapping value is thus a constant value associated to the clock, until a *non-wrapping reset* of the clock (i.e., when $x < N$) which changes its wrapping value.

Using this construction, the values v of a counter C can be kept as the wrapping-value 2^{1-v} of a clock x_C. Therefore, this wrapping-value of x_C is bounded ≤ 2. The conditional branching is done by directly comparing the value of x_C to 2 which can be implemented as a guard, and the increment (and

decrement) operations are done by dividing (multiplying) the wrapping value of x_C by (with) 2, respectively.

For the implementation of the decrement, i.e., the doubling of the value of x_C, a task Q with fixed computation time is released at a nondeterministically chosen time at which also a clock x_{Cnew} is reset. This task is then preempted by a task P of higher priority with variable execution time. P is released when x_C is reset to zero by a wrapping edge. A guard with $x_{done} = 0$ is used to check if its execution time is equal to the wrapping value of x_C, i.e., if it finishes when x_{sys} is reset. This preemption is repeated, and by using the $x_{done} = 0$ guard again afterwards to check that Q finishes when x_{sys} is reset, the wrapping value of x_{Cnew} is finally forced to be the double of the wrapping value of x_C. The response time of Q is a constant time (its computation time) plus two times the wrapping value x_C – because of the two preemptions from P. If any step in the construction fails (some non-deterministic choice was wrong), the automaton enters a sink location where no task is released. Figure 4 illustrates the whole decrement procedure.

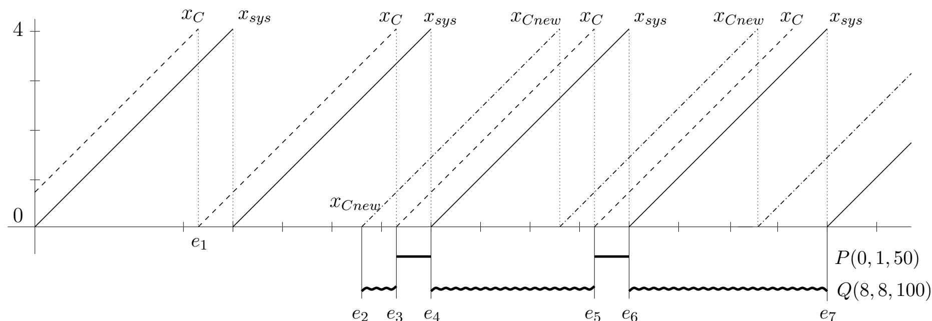

Figure 4: Time chart of the doubling procedure using the N-*wrapping* construction

An increment for C in turn needs to halve x_C. To achieve this, the wanted new value is simply guessed in a clock x_{Cnew} and then checked using the above decrement procedure. Only if the double of x_{Cnew} is x_C, a corresponding guard will be true.

During all three operations, all used tasks meet their deadline (using an FPS scheduler) and therefore, all runs of the automaton which visit only the locations described above do not lead to any unschedulable state. A special location l_{halt} is used for representing the halt state of M: If the task automaton reaches l_{halt}, it will unboundedly release new tasks using an unguarded selfloop. Since they have finite deadlines, this will make the automaton unschedulable.

By this construction it is obvious that the constructed A_M is schedulable if and only if M can not reach its halt state. Note that the construction uses all three properties (preemption, variable execution time, feedback) given in the beginning of this subsection. Section 3 will deal with the role of the feedback

property in the construction.

2.4.2 Decidability

Now we turn to the cases which are known to be decidable, namely task automata with either:

1. A non-preemptive scheduler (but possibly variable execution time and feedback) or

2. Fixed execution time tasks ($B = W$ for all task types, but possibly a preemptive scheduler and feedback).

It is an open question, whether the following third variant is decidable in general, and Sections 3 and 4 will deal with that:

3. No feedback (but possibly a preemptive scheduler and variable execution time).

This last variant has also been proven decidable for certain types of schedulers in [FKPY07], but there is no result for the general case (even though we try to give a good approximation method in Section 4). We will give the ideas of the proofs from [FKPY07] and [EWY99] for these certain scheduler types and also for the above first two types of task automata. They all translate the schedulability problem of task automata to a state rechability problem for timed automata (which is decidable, as shown in Section 2.2).

Non-preemptive scheduler: This is the simplest case. First, the given task automaton A is transformed into a timed automaton $E(A)$ by taking the underlaying timed automaton of A, removing the labels and adding new labels $\mathsf{release}_P$ on transitions where A would release an instance of task type P. Because of Definition 6 and Lemma 1, the whole scheduling strategy on schedulable queues can be encoded as a timed automaton $E(\mathsf{Sch})$ (the *scheduler automaton*) as follows. The continuous part of the queue, namely the best/worst case computation times and remaining relative deadline values, are encoded in clocks. Two clocks x_i^c and x_i^d exist for each task instance p_i in the queue, let's denote its type as P_j. Using these clocks, the values b_i, w_i and d_i from the queue will be expressed as $B_j - x_i^c$, $W_j - x_i^c$ and $D_j - x_i^d$ for the running task, and B_j, W_j and $D_j - x_i^d$ for waiting tasks, respectively. The discrete part of the queue (the order of the task instances) will be encoded in locations of $E(\mathsf{Sch})$. Since the queue length of schedulable queues is bounded (Lemma 1), there will be only finitely many of these locations. Once the queue becomes non-schedulable, $E(\mathsf{Sch})$ will enter a dedicated error location. The edges and their guards correspond to the comparisons which the scheduling function uses for its decisions. The above expressions for b_i, w_i and d_i will be used in the guards of $E(\mathsf{Sch})$. The correct values of the clocks in these expressions are ensured by the following construction. The release of a new instance of P_j (event $\mathsf{release}_{P_j}$) resets a clock x_i^d, which will keep track of the amount of time for which this task instance p_i

is already released. As soon as the scheduler puts this task instance to the head of the queue (which models the beginning of the task execution), the clock x_i^c is reset, keeping track of the computation progress of the task. The running task instance p_i may finish whenever "$B_j \leq x_i^c \leq W_j$" holds. (In this case, $E(\mathsf{Sch})$ removes p_i from the queue.) And whenever a constraint "$x_i^d > D_j$" is met for a released task instance p_i, an error location "non-schedulable" is entered. Note that only one task is running at a time and the others did not even start their computations because the scheduler is non-preemptive. The resulting product automaton $E(\mathsf{Sch}) \parallel E(A)$ is a timed automaton for which it can be proven that the "non-schedulable" error state is reachable if and only if A is non-schedulable with scheduling strategy Sch.

Note that for this result, the definition of the scheduling function (Definition 6) does not have to be that restrictive in the sense that also diagonal constraints can be allowed for the decisions in the scheduling function. The reason is that the computation time and deadline values in the queue are encoded into (at most) one clock each. The possibility of using diagonal constraints in the scheduler's decisions makes (for this non-preemptive case) encoding even of LSF possible.

Fixed computation time: In this case, tasks are allowed to preempt other already running tasks, which is the main difficulty for handling the computation time of all tasks in the queue. Here, the same construction from before is used, i.e., two automata $E(A)$ and $E(\mathsf{Sch})$ are created. Again, $E(\mathsf{Sch})$ keeps track of computation progress and time pass since the release in clocks x_i^c and x_i^d for each released task p_i. Because of the preemption, tasks waiting in the queue may have already been executed for some time (in contrast to the case above), so their x_i^c clocks may be already running. This is resolved by just letting them run further (even while the task is preempted) and representing the time for which a preempted task p_i was already computed not just by its x_i^c clock, but by a difference $x_i^c - x_m^c$. Here, p_m is the task which directly preempted p_i, in the sense that the newly released p_m replaced p_i at the head of the queue. When a running task p_k finishes, all x_i^c of the preempted tasks p_i are updated by subtracting the computation time which was needed by p_k (and which is a constant natural number because of the fixed computation time property). This introduces subtraction clock-updates (of natural numbers) into the resulting timed automaton $E(\mathsf{Sch})$ (and therefore into the product automaton $E(\mathsf{Sch}) \parallel E(A)$). But since they are bounded by known constants, the reachability problem for this *Timed Automaton with Bounded Subtraction* is decidable as proven in [FKPY07].

Note that here, the diagonal-freeness in guards from the scheduler (see Definition 6) is important, because it is necessary to use clock differences to express the computation time of tasks. (A comparison of a clock difference to a constant is already a diagonal constraint which cannot be further extended by another clock.)

No feedback: Finally, we have the case of a task automaton where the scheduler is preemptive and the tasks can have variable execution times. Further, the finishing time of the tasks may not influence the time of new task releases (no feedback). For a certain class of schedulers which includes EDF and FPS (but not SJF), decidability of the schedulability problem is known. The idea of the proof in [FKPY07] is that a run of the task automaton with variable task computation times where a task misses its deadline will also be a run in the same automaton when all tasks take their worst case execution time, and this run with worst case times will also make a task miss its deadline. This in turn reduces the schedulability problem to the case with fixed computation times of tasks.

3 Undecidability

In this section we show, that the feedback property can be replaced by one bit of scheduler memory or by the utilization of more than one processor, while the undecidability result for schedulability still holds.

3.1 Undecidability for schedulers with finite memory

The undecidability result from [FKPY07] for which we sketched the proof in Section 2.4.1 relies strongly on the feedback property. We will see that such a powerful tool like feedback (in the sense that the precise finishing time of a task may influence new task releases) is not needed. It will be shown that the problem of schedulability is still undecidable if the scheduler has a finite memory even if no feedback is used. A memory of just one bit of information is enough to achieve this result.

Before we sketch the main ideas, the central definition and the main theorem are given:

Definition 9. *(Scheduler with finite memory) Let* $\mathcal{M}em = \{0,1\}^k$ *for a* $k \in \mathbb{N}_{\geq 1}$*, called the* scheduler's memory*. Then* $\mathsf{Sch} : \mathcal{P} \times Q_{\mathcal{P}} \times \mathcal{M}em \mapsto Q_{\mathcal{P}} \times \mathcal{M}em$ *is a* scheduling function with finite memory*, if for each* $m \in \mathcal{M}em$*, the function* $\mathsf{Sch}(\cdot,\cdot,m)$ *with image restricted to* $Q_{\mathcal{P}}$ *is a* scheduling function *(see Definition 6).*

Theorem 2. *The problem of checking, whether a task automaton without feedback but with a scheduler with finite memory is schedulable, is undecidable.*

Main ideas: We start with the same construction as sketched in Section 2.4.1. A task automaton A_M is defined which can simulate a two-counter machine M. The construction is exactly the same as before, but we change three aspects. First, at each position where an edge contains an $x_{done} = 0$ guard for synchronizing on the finishing of a task instance of a task type P, this guard is removed and two additional locations l_1 and l_2 are inserted. The first one releases an instance of a task type T^P_{chk1}, the second one an instance of a task type T^P_{chk2}.

(Both released instances will be put to the end of the queue.) The original edge which contained $x_{done} = 0$ goes to l_1 and the edges from l_1 to l_2 and from l_2 to the old location will contain guards ensuring they are taken in zero time. See Figure 5.

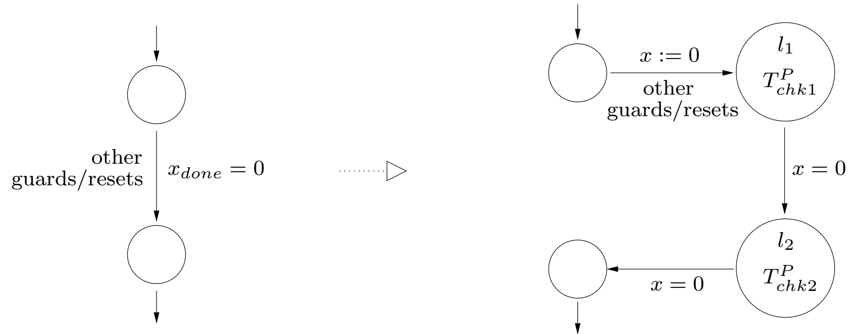

Figure 5: Replacing the check for $x_{done} = 0$ with two additional locations and an additional clock

Second, the scheduler not just implements FPS as in Section 2.4.1 but contains finite memory, see Definition 9. Note that the *scheduling function* from Definition 6 is the special case where $k = 0$, and that this "finite memory" property for $k > 0$ is a strict extension. The scheduler in our case needs a memory m of just one bit ($m \in \{0, 1\}$), which is initially set to zero ($m := 0$). When tasks of type T^P_{chk1} or T^P_{chk2} are released, the scheduler will check if a task of type P is in the queue: If it *is not* in the queue at the release of T^P_{chk1}, the scheduler sets $m := 1$. If it *is* in the queue at the release of T^P_{chk2}, the scheduler also sets $m := 1$. (See Figure 6.) By this, $m = 0$ will hold afterwards if and only if between both task releases (which happen at the same time point) the task disappeared from the queue. This is exactly what the reset of x_{done} expresses. This property of m holds inductively for all states of the automaton.

The third thing which is changed from the construction of Section 2.4.1 is the halt location. Here, instead of a selfloop releasing unboundedly many task instances, two tasks $R_1(1, 1, 1)$ and $R_2(1, 1, 2)$ are released at the same time, and then the automaton enters a sink location where no tasks are released. (See Figure 7.) The scheduler lets R_2 be computed first, making R_1 miss its deadline, if and only if $m = 0$. This means that a deadline miss occurs if and only if $m = 0$, which in turn can happen only if all checks with T^P_{chk1}/T^P_{chk2} invocations were "successful", i.e., all task instances finished when we wanted them to finish.

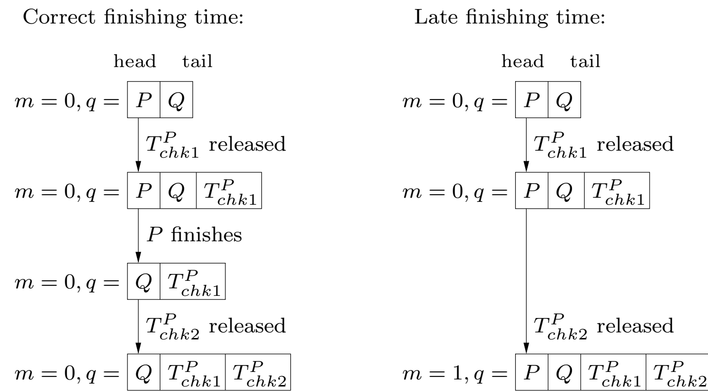

Figure 6: Two examples of automaton runs: On the left, task P finishes between the releases of T^P_{chk1} and T^P_{chk2}, therefore m stays at 0. On the right, P does not finish at this timepoint and the scheduler remembers this by setting $m := 1$.

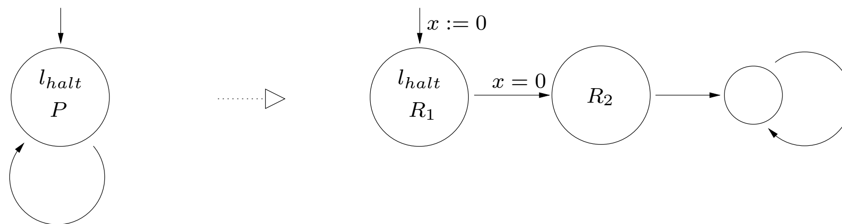

Figure 7: Replacing the halt location with two task releases (which happen at the same time using an additional clock)

Note that these released "checking" tasks T^P_{chk1}/T^P_{chk2} of course need to be executed some time because they also have a deadline. This can be done by adding sufficiently many "idle cycles" of the wrapping-construction after each instruction to give the CPU time to execute them and thus remove them from the queue by the scheduler. (The computation time and deadline requirements of T^P_{chk1}/T^P_{chk2} can be set sufficiently small/big to achieve this.) This also keeps the needed invariant "empty queue after each instruction" true.

As in Section 2.4.1, the following lemma states the property needed for the reduction of the halting problem.

Lemma 3. *For a given two-counter machine M, the constructed task automaton A_M is not schedulable with the constructed scheduler if and only if M halts.*

Proof (Sketch). If the two-counter machine M *can* reach the halt state, then A_M can simulate the same instructions leading there. While doing this, the executions of P and Q can always finish at the same time they would also finish in the construction from Section 2.4.1. This keeps $m = 0$ all the time. When A_M then enters its halt location, R_1 and R_2 will be executed in the order which makes the queue unschedulable.

If in turn M *cannot* reach its halt state, then A_M has to "cheat" to reach its halt location, which formally means that the following holds for the task instances of P and Q: It cannot be true that all executions of P and Q finish at the time at which $x_{done} = 0$ would hold in the automaton from Section 2.4.1 (in particular, this is checked at a reset of $x_{sys} := 0$). The reason is that then the automaton from Section 2.4.1 would reach its halt location, and therefore the calculation of M would lead to M's halt state. Consequently, because P or Q finishes at least once at a different time, this "cheating" of A_M gets detected by the scheduler through T^P_{chk1}/T^P_{chk2}, setting $m := 1$. This means that A_M can then only reach the halt location with $m = 1$, making the scheduler execute R_1/R_2 in the order which allows both to meet their deadlines. Provided that during the run up to this point all tasks met their deadlines (which is ensured by construction), the automaton A_M is schedulable in this case. □

Consequently, this lemma proves Theorem 2.

3.2 Undecidability with more than one CPU

The undecidability result from the previous section extends easily for the case of multi-processor scheduling, where it even holds if the scheduler has no memory. (We show this result more detailed in [KSY07].) First, we give again the central definition and theorem. The definition is a direct extension of the single-processor variant given in Definition 6.

In the multi-processor case, the scheduling function takes k queues and a task type as an input and returns the queues with the new task instance inserted at some position in one of the queues. It can use the information from all the queues for its decision. Each timed automaton corresponding to a scheduling policy then contains clocks for all the instances in all the queues. To simplify the notation, we assume that for k queues $q_1, \ldots, q_k$ with discrete parts $\tilde{q}_1, \ldots, \tilde{q}_k$,

all the task instances are indexed by one index i ranging from 1 to $\sum_{1\leq j\leq k}|q_j|$, where $|q_j|$ denotes the number of the task instances in the queue q_j. E.g., the index of the first task instance in q_2 is $|q_1|+1$. Also, all possible positions where a new task can be inserted are indexed by one index ranging from 1 to $k+\sum_{1\leq j\leq k}|q_j|$. E.g., the $|q_1|+2$-th (global) position denotes the first (head) position of the second queue (the first task instance in q_2 after insertion). By b_i, w_i and d_i we denote as before the continuous queue information for the task instance p_i.

Definition 10. *(Multi-processor Scheduler) Let $k\in\mathbb{N}$ be the* number of processors. *Then* $\mathsf{Sch}_k : \mathcal{P}\times(Q_{\mathcal{P}})^k \to (Q_{\mathcal{P}})^k$ *is a* multi-processor scheduling function, *if for each task type $P(B,W,D)$ and* discrete parts *of queues $\tilde{q}_1,\ldots,\tilde{q}_k$, there can be effectively constructed a diagonal-free timed automaton with*

- *Clocks $y_1^b, y_1^w, y_1^d, \ldots, y_K^b, y_K^w, y_K^d$ where $K=\sum_{1\leq j\leq k}|\tilde{q}_j|$,*
- *$K+k+1$ locations $l_0, l_1, \ldots, l_{K+k}$ and*
- *$K+k$ edges from l_0 to l_j for $1\leq j\leq K+k$,*

such that the function $\mathsf{Sch}(P(B,W,D), q_1,\ldots,q_k)$ inserts $P(B,W,D)$ at the m-th (global) position if and only if l_m is the only location reachable from (l_0,u) where $u(y_i^b)=b_i, u(y_i^w)=w_i, u(y_i^d)=d_i$ for all $1\leq i\leq K+k$.

Note that this definition does not allow for moving tasks between processors after they got assigned to one processor at the moment when they are released, that is, we do not allow task migration.

The definition of task automata extends in a straightforward way to this multi-processor variant by assigning released tasks to one of the k queues for the k processors. We call this extension a *k-multi-processor task automaton.* For this extended automata model, we have the following result:

Theorem 4. *The problem of checking, whether a k-multi-processor task automaton without feedback is schedulable, is undecidable for $k\geq 2$.*

Again, we sketch the proof: Let $k\geq 2$. We use the above construction from Section 3.1 to build a task automaton A'_M for a given two-counter machine M. Instead of using a memory bit m, we will use the task queue of a second processor to remember the results of the "checks" done by the task releases of T^P_{chk1} and T^P_{chk2}.

This is implemented by using the first processor with queue q_1 as before (except that T^P_{chk1} and T^P_{chk2} will be scheduled in q_2), but continuously keeping the processor utilization of the second processor (with q_2) at 100%. We will use an additional task type T_{mark} and the already introduced task types T^P_{chk1} and T^P_{chk2} to store the needed information. Directly before (and at the same time of) the release of T^P_{chk1}, an instance of T_{mark} is released to the end of q_2. If the following release of T^P_{chk1} would set $m:=1$ (or if q_2 is already in a state equivalent to $m=1$), the scheduler will put T^P_{chk1} directly *after* T_{mark} in q_2, otherwise directly *before* T_{mark}. The same goes for T^P_{chk2}. This way, the

information $m = 0$ of A_M is equivalent to the information in q_2, that the last task in q_2 is of type T_{mark}.

All task types T^P_{chk1}, T^P_{chk2} and T_{mark} will have fixed computation times and it is possible to adjust these to values such that the queue is never empty (thus not losing the information) while the involved tasks will meet their deadlines. We show the decrement part of A_M in Figure 14 in Appendix A.2, an extension from the one presented in [FKPY07].

4 Decidability for Variable Task Computation Time

This section introduces an algorithm to approximate the schedulability problem for the general case of task automata without feedback. Note that the setting considered here is the original one, i.e., a single-processor scheduler without memory. Only for certain classes of schedulers, it is known that the schedulability problem is decidable when there is no feedback used in the task automaton.

We give an introductory example why the decidability result sketched in Section 2.4.2 is only applicable for these certain cases, but not in general. The proof there was based on the property of the scheduler, that a run of the task automaton which leads to an unschedulable state[2] will as well lead to an unschedulable state if all tasks released along this run finish at their worst case execution times. While this holds for, e.g., EDF and FPS, the example in Figure 8 shows that this is not the case for the Shortest Job First scheduling strategy (SJF).

3 task types: $P(1,3,3)$, $Q(4,4,9)$, $R(2,2,2)$

$x := 0$ → P → $x = 0$ → Q → $x = 4$ → R

P takes worst case time:

Queue when R released at $t = 4$: $[R(2,2,2), Q(3,3,5)]$

P takes shorter time:

R misses its deadline

Queue when R released at $t = 4$: $[Q(1,1,5), R(2,2,2)]$

Figure 8: Example for the case that with SJF, deadline misses might occur when tasks do not take their worst case execution time.

[2]That is a state where a task missed its deadline.

In this example, there are three task types $P(1,3,3)$, $Q(4,4,9)$ and $R(2,2,2)$ with best/worst case execution times and relative deadline. Note that only P has variable execution time. Instances of P, Q and R are released at absolute time points 0, 0 and 4. In the first drawn case, P takes its worst execution time of 3 time units. At time point 4, when R is released, Q will have 3 time units of computations left. Thus SJF schedules R on the *first position* of the queue and all tasks will meet their deadlines. In the second case, P takes its best case execution time of 1 time unit. Consequently, when R is released at $t = 4$, Q has just one time unit left to compute, and thus R is scheduled *behind* Q in the queue, which will ultimately lead to a deadline miss of R at time point 6.

As this example shows, the result described in 2.4.2 is not applicable for the general case of any given scheduler. Since the decidability in this general case is not known, we present an algorithm which overapproximates the behaviours of task automata for this case. This means that the algorithm is always right when answering "schedulable" to a given task automaton/scheduler pair, but might be wrong when answering "non-schedulable". However, although it is an open question, we conjecture that the construction is exact (answering correctly also in this case), thus deciding the schedulability problem for the general case of variable computation time without task feedback.

4.1 Main ideas

Our construction is based on the same technique already used in the decidability proofs of Section 2.4.2: We encode the remaining computation times (for best and worst case) and the remaining deadlines of the released task instances using dedicated clocks x_i^c and x_i^d for each task instance p_i in the queue. As in Section 2.4.2, the main difficulty is to maintain the clock values during the preemption of tasks. The problem to be solved is that in addition to the preemption, we have to deal with the variable task computation times within the interval of best and worst case times. Remember that the computation time clocks x_i^c of preempted tasks p_i cannot be stopped while these tasks are preempted. Therefore, when the execution of a task p_j finishes, the value of its computation time clock x_j^c has to be subtracted from all computation time clocks x_i^c of tasks p_i preempted at this time point. But in contrast to Section 2.4.2, this value can now be any real number within the interval $[B_j, W_j]$ between best and worst case execution time. (Remember that in Section 2.4.2, we had fixed computation time via $B_j = W_j$, which made it possible to encode this into a *timed automaton with bounded subtraction*.) A clock update would be needed which subtracts the clock x_j^c from clock x_i^c. For timed automata, it is known that general subtraction updates render the reachability problem undecidable.

For this reason, we cannot use a straightforward translation into a timed automaton as done in Section 2.4.2. Instead, as it was done for *timed automata* in the form of the *region graph*, we directly construct a quotient transition system called *symbolic semantics* of the transition system from Definition 7 (semantics of task automata). To emphasize the difference, we call the semantics from Definition 7 *concrete semantics*. This construction of the new LTS works like

the *region graph*, with some added information to the state (like the queue) and with special transitions for task finishing. (These task finishing transitions will represent the described clock subtractions, which are not possible to be expressed in timed automata.)

Special care has to be taken for these task finishing transitions, because as we will see, a *clock region* can have several different *successor regions* for this transition, depending on the precise value of the clock that is to be subtracted. (The region represents a set of these values and different subsets will lead into different successor regions.) Since different successor regions are possible, the transition system has to choose one non-deterministically among all possible ones.

In our first construction, we just define all of these possible regions as successor regions, but this choice is too broad, as there are successor regions formally possible that cannot be reached with concrete clock valuations. This is because the construction of an automaton can impose restrictions of possible concrete clock valuations reachable within a region, while those restrictions are not reflected in the pure clock region information. We address this problem by defining and using *Clock Difference Relations* (CDRs), a concept originally introduced in [KP05] to characterize reachability relations of timed automata. We will use these relations to reduce the set of possible successor regions (by imposing new restrictions on valuations in the clock regions) and keep them as additional state information.

4.2 Alternative concrete semantics

Before defining the symbolic semantics, we first introduce *alternative concrete semantics* (ACS). This is just for technical reasons and formalizes the already introduced concept: Originally, the computation time and deadline information is kept directly in the queue (see Definition 7). But to be able to handle this information with tools for timed automata (like a region graph equivalent), it would be better to keep it in clocks instead. Since the detailed definition is rather technical, we give it in Appendix A.1. We instead just describe the main points differing from the semantics of task automata from Definition 7 and it is clear that both semantics can be used equivalently. (Appendix A.1 defines a bisimulation relation between this alternative concrete semantics and the concrete semantics from Definition 7.)

To keep the computation and deadline information in clocks, the alternative concrete semantics augments the set $\mathcal{C}$ of clocks to a set $\hat{\mathcal{C}}$ with two additional clocks x_i^c and x_i^d for each possible task instance in a schedulable queue. (This is for each automaton a constant number, since the length of schedulable queues is bounded.) Instead of the time information which was formerly kept in the queue, these new clocks are now used to express the computation time and deadline values in the scheduler, and will be updated to reflect that, in the same way already described in detail earlier in Section 2.4.2. In turn, the queue now only maintains "discrete" information like task order, type and status (whether an instance is running, preempted or still waiting for execution). Note that this

definition of a queue was called ”discrete part of a queue” in the old semantics. The scheduling function can operate as before, but uses clock values and their differences now instead of the time information (for elapsed computation and deadline times) directly from the queue. All other objects and operations naturally adapt to this new set of clocks and the new queue structure without problems.

4.3 A simple approach: Very rough overapproximation

This *alternative concrete semantics* is a basis for the construction we build now. (From here on, we mean this ACS when we talk about ”concrete semantics”.) Since the state space of this LTS is infinite, we construct a quotient system called *symbolic semantics* with finite state space, which makes it possible to check reachability properties (and therefore the schedulability question, whether a state with a non-schedulable queue is reachable). However, before presenting a more refined version in the next section, the construction here is a strict overapproximation of the concrete semantics because there are states in its finite state space which cover no state from the concrete semantics. (Therefore, a state which is actually not reachable within the concrete semantics state space might be reachable in the symbolic semantics state space.)

The construction basically works like the *region graph* for *timed automata*. The standard *region equivalence* (Definition 3) for clock valuations over the *augmented* set of clocks imposes a finite quotient of the set of clock valuations. A location l of the automaton together with such a clock region D and a queue q form a state $s = (l, D, q)$ of the LTS called *symbolic semantics*. Note that this queue q contains only discrete information since the basis of this construction is the alternative concrete semantics from Section 4.2 above. As in the concrete semantics, we have three types of transitions which reflect the transitions from concrete semantics:

Event: The automaton receives/produces a symbol from $\mathcal{A}ct$ and changes the location while potentially a new task instance is released (which would be of the task type associated with the new location). In a state (l, D, q) of the LTS, this transition changes the location l, as in the region graph. Additionally, if a new task was released, the queue q gets the new instance inserted according to the scheduling policy, and in the clock region D, the corresponding clocks x_i^d and potentially x_i^c (in case the new task was inserted on top of the queue) are reset to reflect the release. The clock region D is not changed otherwise.

Time pass: As in the region graph, this transition changes only the clock region D and goes to all possible *time-successors*. (See Section 2.2.) Note that this automatically reflects progress in computation and deadlines, since the information about that is kept in clocks.

Task finishing: This transition type does not exist in the region graph. Here, l stays unchanged and the task which just finishes is removed from the queue. The most important change happens in the clock region D. The transition has to reflect the clock subtraction to update the computation time clocks x_j^c of all preempted tasks p_j. In detail, all concrete clock valuations covered by the clock region D have a successor valuation within the concrete semantics. The transition in the symbolic semantics has to go to a clock region which contains this concrete successor. As illustrated in in Figure 9, there might be different successor regions possible, depending on the value of the computation time clock x_1^c of the task which just finishes.

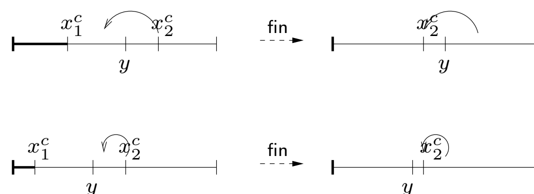

Figure 9: In both examples, x_1^c and x_2^c are computation time clocks of a finishing task and a preempted task, respectively. Both transitions start in the same region, but the successor valuations are in different regions.

So which region is the right one? The choice of the successor clock region will depend on the actual precise value of x_1^c. Since the original region D covers many different values of x_1^c, there are also (potentially) different successor regions possible. The LTS of the symbolic semantics has to reach all of them, so there are edges to all of them in the LTS presented in this section. (We will refine this in the next section.)

The constructed LTS has now two properties regarding the represented states from the concrete semantics. All reachable states from the concrete semantics are covered, but there are too many states from the concrete semantics covered, in the sense that there might be states reachable in the symbolic semantics, for which none of the covered states from the concrete semantics is reachable. We will now take a look at both properties.

All states are covered: This property expresses that the construction is at least an overapproximation: If the concrete semantics can reach a state (l, ν, q), then the symbolic semantics can reach a state (l', D, q') which covers this state, i.e., $l = l'$, $\nu \in D$ and $q = q'$. In particular, this means that if all reachable queues in the symbolic semantics are schedulable for a given scheduling strategy, then also all reachable queues in the concrete semantics are schedulable, i.e., the automaton is schedulable.

A proof for this would just have to show that for each reachable state (l, ν, q) of the (alternative) concrete semantics, a state (l, D, q) in the described symbolic semantics is reachable with $\nu \in D$. This follows directly from the above construction of the three types of transitions.

Too many states are covered: This second property in turn shows that the construction is indeed a strict overapproximation: There are symbolic states (l', D, q') reachable, for which no state (l, ν, q) is reachable in the concrete semantics with $l = l'$, $\nu \in D$ and $q = q'$. Intuitively, this is caused by relations between differences of clock valuations which are implied by the choice of certain successor regions in the task finishing step. To illustrate that, we give a complete example in Figure 10.

In this example, we have two tasks with variable execution time, $P(0, 1, 2)$ and $Q(0, 1, 2)$. Additionally, there are four tasks to "check" existence of P and Q in the queue and remember the result of the check. T_{mark}, T^{P}_{chk1}, T^{Q}_{chk1}, T^{Q}_{chk2} have all $(5, 5, 100)$ as task parameters. The automaton has two parts. In the first part, the four clocks x_1, x_2, y_1 and y_2 are reset and at the reset of x_1 and y_1 the tasks P and Q are released, respectively. Q preempts P. The region diagrams next to the edges illustrate the clock region during a run of the automaton. The check tasks T^{Q}_{chk1} and T^{Q}_{chk2} are released after P and Q and will be scheduled to the end of the queue. They go *behind* T_{mark} if Q was in the queue (for T^{Q}_{chk1}) or not in the queue (for T^{Q}_{chk2}) at their release, respectively. This is checked by the scheduler. The same holds for T^{P}_{chk1}. (Note that these checks are done in the same way as in Section 3 where they are used to simulate a guard $x_{done} = 0$.) Through this, the following holds when the automaton takes the transition to the second (right) part: If all check tasks are *behind* T_{mark} in the queue, then $\overline{x_2 x_1} < \overline{y_2 y_1}$ has to hold for all clock valuations ν in a state taking this transition. In the symbolic semantics, this knowledge is not preserved.

In the second part of the automaton, the same is repeated but with the roles of the x and y clocks swapped. Consequently, $\overline{y_2 y_1} < \overline{x_2 x_1}$ would have to hold for all clock valuations ν in a state reaching the last location, if all check tasks are behind T_{mark} in the queue. Since the x and y clocks are not reset between both parts of the automaton, and therefore the clock differences do not change, this clock difference relation can obviously not hold for states in concrete semantics, while the symbolic semantics can still reach a state with this shape of the queue in the last location. Thus, the resulting symbolic state does not cover any reachable concrete clock valuation in the concrete semantics.

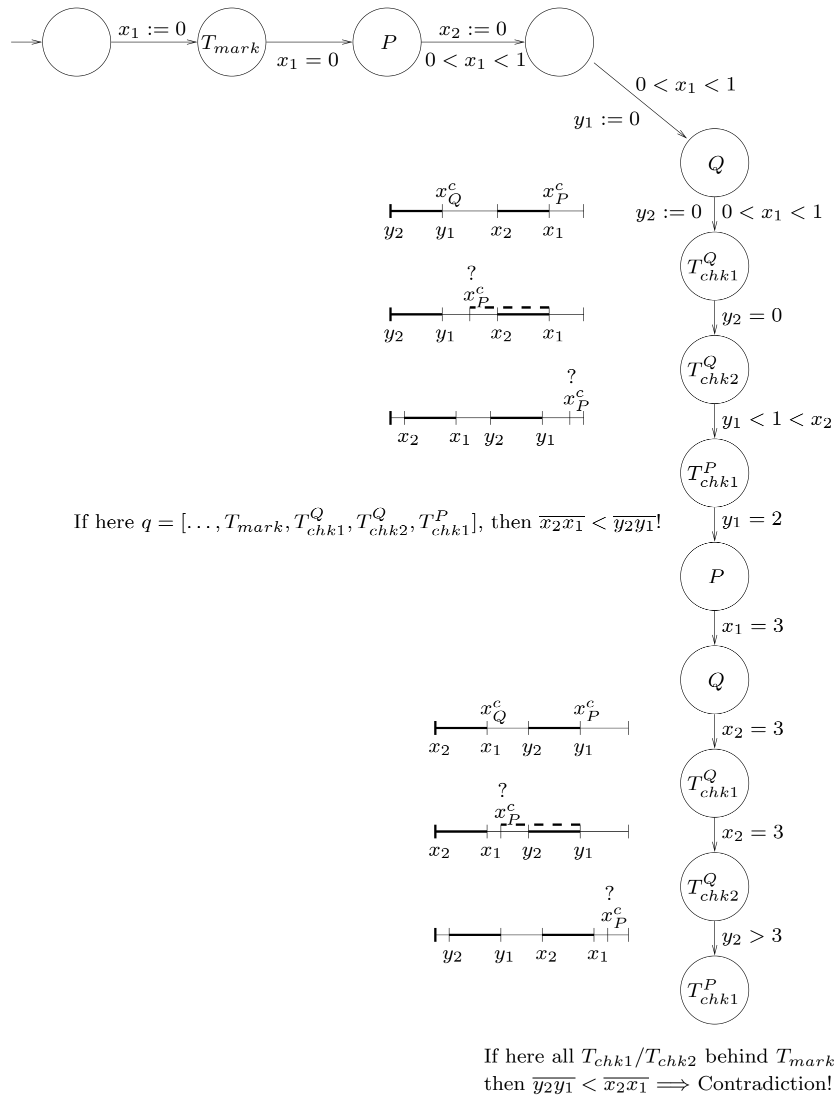

Figure 10: Example that the concept of Section 4.3 is a strict overapproximation.

4.4 Better approach using CDRs

We address this problem now by introducing *Clock Difference Relations* (CDRs). A CDR imposes relations between differences of clock values. This concept has already been used in [KP05] where clocks from two different regions are compared. We apply this now to do comparisons directly within a region and develop a slightly more flexible notation. The following Definition 11 is illustrated in Figure 11 using *region diagrams*.

Definition 11. *A* Clock Difference Relation (CDR) *is an expression of the form*

$$\overline{x_1x_2} \bowtie \overline{y_1y_2}$$

where $x_1, x_2, y_1, y_2 \in \mathcal{C}$ *are clocks and* $\bowtie \in \{<, =\}$ *is a relation symbol. Let* ν *be a clock valuation, then:*

$$\nu \models \overline{x_1x_2} \bowtie \overline{y_1y_2} \;:\Longleftrightarrow\; \nu(\overline{x_1x_2}) \bowtie \nu(\overline{y_1y_2})$$

where

$$\nu(\overline{x_1x_2}) := \begin{cases} \langle\nu(x_2)\rangle - \langle\nu(x_1)\rangle & \text{if } \langle\nu(x_1)\rangle \leq \langle\nu(x_2)\rangle, \\ 1 - \big(\langle\nu(x_1)\rangle - \langle\nu(x_2)\rangle\big) & \text{otherwise.} \end{cases}$$

For a set $C = \{c_1, \ldots, c_n\}$ *of CDRs* c_j *we define:*

$$\nu \models C \;:\Longleftrightarrow\; \forall j : \nu \models c_j$$

The set of all CDRs over a set $\mathcal{C}$ *of clocks is denoted by* $\mathcal{CDRs}(\mathcal{C})$*.*

Note that we might use the relation symbol $>$ if the CDR with both sides swapped is meant, and $\leq$ or $\geq$ if we mean both CDRs with $=/<$ or $=/>$ in conjunction, respectively. Note further that a clock difference $\overline{x_1x_2}$ "wraps" around the region boundary if the fractional value of x_1 is larger than of x_2. E.g., $\nu(\overline{x_1x_2}) = 1 - \nu(\overline{x_2x_1})$.

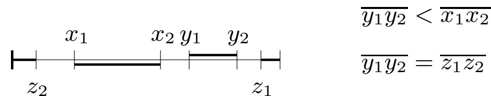

Figure 11: Example for two CDRs that hold for this particular clock valuation.

These CDRs now can be used to further restrict the set of concrete clock valuations represented by a clock region in the symbolic semantics. To do this, a set of CDRs will be maintained, extending the state information of the symbolic semantics and reflecting these restrictions.

The key observation for this is, that after a *task finishing* transition, certain (additional) relations hold for the clock valuations of both the original and the successor region. We give a simple example for that in Figure 12.

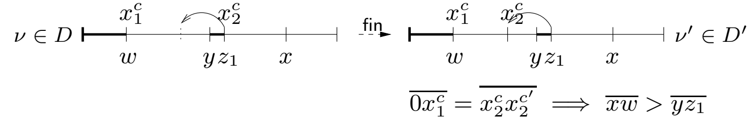

Figure 12: A *task finishing* transition implicitly imposes constraints on covered concrete valuations. With ${x_2^c}'$ in the CDR expression we mean the new position of x_2^c in D'.

This figure illustrates how a *task finishing* transition may change a clock region D to a clock region D'. Two concrete valuations from both regions are shown. In addition to four "usual" clocks w, x, y and z_1, there are two computation time clocks x_1^c and x_2^c. Of these two, x_1^c belongs to the task that finishes and x_2^c to a preempted task, therefore x_2^c is moved. Observe that $\nu(x_1^c)$ is so large, that y and x_2^c swap their order from D to D', since, by definition, $\nu(x_1^c) = \nu(x_2^c) - \nu'(x_2^c)$. This has an indirect influence on a difference relation between other clocks in the region: After this transition, $\overline{xw} > \overline{yz_1}$ holds for all $\nu' \in D'$. Further, the CDR has to hold even *before* the transition takes place, namely for all $\nu \in D$, since the values of the four concerned clocks are not changed by this transition. (Note that the CDR would even hold if $\nu(z_1)$ would be slightly smaller, as long as it is greater than $\nu(y)$.) This means in particular, that from this D, the *task finishing* transition must not go to this region D' in case there is already some contradicting knowledge about the clock differences. (E.g., if the transition would impose a CDR $\overline{x_1x_2} < \overline{x_3x_4}$ but the maintained set of CDRs already contains $\overline{x_1x_2} > \overline{x_3x_4}$.)

A slightly different example is shown in Figure 13. Instead of clock z_1 with the same value as x_2^c *before* the transition, we now have a clock z_2 with the same value x_2^c will have *after* the transition. Again, the transition will indirectly impose a restriction on the set of concrete valuations covered by D and D', in particular the CDR $\overline{xw} > \overline{z_2y}$ holds for all $\nu \in D$ and $\nu' \in D'$. (Note again that this would even hold if $\nu(z_2)$ would be slightly greater as long as it is smaller than $\nu(y)$.) And again, a possible contradiction to previous knowledge would prevent this *task finishing* transition to go from D to this particular D'.

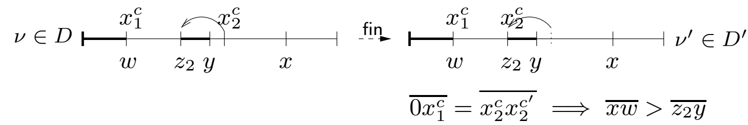

Figure 13: Another example for implicitly imposed constraints.

Both examples show, that the *task finishing* transition introduces knowledge about the clock difference between both old and new value of computation time clocks x_2^c of tasks which are preempted at that time. As we saw, this knowledge indirectly extends to the whole set of clocks. The rest of this section deals with

applying this insight to the new *refined symbolic semantics*.

First, we formally introduce the knowledge about CDRs into the state information. The state has now an extended form (l, D, C, q) where C is a (finite) set of CDRs that is updated during each transition, reflecting the newly gained knowledge about clock difference relations. Implicitly, this set C always retains the property that for each CDR $\overline{x_1x_2} \bowtie \overline{y_1y_2} \in C$ there is also $\overline{y_2y_1} \bowtie \overline{x_2x_1} \in C$, for symmetry reasons.

To be able to keep track of all indirect knowledge we gain by inserting one single CDR into the set, we define two constructs:

$\mathsf{RegionCDRs}(D)$**:** For a clock region D, this function $\mathsf{RegionCDRs} : \mathcal{R}(\mathcal{C}) \rightarrow \mathcal{C}DRs(\mathcal{C})$ returns a set of all CDRs implicitly defined by the clock relations of the region. (E.g., if $\langle x_1 \rangle < \langle x_2 \rangle < \langle x_3 \rangle < \langle x_4 \rangle$ holds for the region, then $\overline{x_1x_2} < \overline{x_1x_3}$ and $\overline{x_2x_3} < \overline{x_1x_4}$ will be in this set, among others.)

$\mathsf{Closure}_D(C)$**:** Given a clock region D, this function $\mathsf{Closure}_D : \mathcal{C}DRs(\mathcal{C}) \rightarrow \mathcal{C}DRs(\mathcal{C})$ returns the transitive closure of all CDRs in the set. (E.g., $\overline{x_1x_2} < \overline{y_1y_2} \in C$ and $\overline{y_1y_2} = \overline{z_1z_2} \in C$ will add $\overline{x_1x_2} < \overline{z_1z_2}$ to the set.)

Note that both functions operate on the given sets in a purely syntactical way.

The key tool to handle now the actual new knowledge in the *task finishing* transition, is the use of "virtual clocks". These are clocks which are used in the clock region and the CDR set during the construction of the successor state, but do not appear in any state at all. In particular, we use a virtual clock $\tilde{x}_0$ and for each preempted task p_j a virtual clock $\tilde{x}^c_j$, where $\tilde{x}_0$ will be used to temporarily denote a clock with $\lfloor D(\tilde{x}_0) \rfloor = 0$ and the clocks $\tilde{x}^c_j$ will be placeholders for the new value of the computation time clocks x^c_j of the preempted tasks p_j.

We now describe the whole process of creating a successor state (l', D', C', q') to a given state (l, D, C, q) in this *refined symbolic semantics* by the *task finishing* transition. We have $l' = l$ and q' is q with the running task removed and the status information in the other tasks adjusted. Second, D' and C' are constructed by the following steps, where x^c_1 denotes the computation time clock of the task which just finishes:

1. Insert a virtual clock $\tilde{x}_0$ and for each preempted p_j a virtual clock $\tilde{x}^c_j$ into D, where $\lfloor D(\tilde{x}_0) \rfloor = 0$ and all $\tilde{x}^c_j$ will get those positions in D which will be the new ones of the x^c_j clocks in D' after the transition (see Section 4.3). Note that there are different choices for the positions of the $\tilde{x}^c_j$ possible, depending on the chosen successor region (as described in Section 4.3). Apply the functions $\mathsf{RegionCDRs}$ and $\mathsf{Closure}_D$ afterwards to ensure proper integration of the new clocks.

2. Add the new direct CDR knowledge to C: For each preempted task p_j, that will be the CDR $\overline{\tilde{x}_0x^c_1} = \overline{\tilde{x}^c_jx^c_j}$.

3. Apply the $\mathsf{Closure}_D$ function to spread this direct knowledge into indirect knowledge for other clocks.

4. If C now contains contradictions, stop the process since no concrete valuation covered by the chosen new region D' would be reachable in concrete semantics.

5. Replace the virtual clocks $\tilde{x}_j^c$ by their "real" counterparts x_j^c in region D and CDR set C. This is done by removing all CDRs from D containing clocks x_j^c, removing x_j^c from the region D and then renaming all $\tilde{x}_j^c$ to x_j^c in D and C. This results in the new region D' and CDR set C'.

While it is quite complex to get the successor state (l', D', C', q') from state (l, D, C, q) by the *task finishing* transition, the other two transition types are straightforward adaptions from Section 4.3: The *event* transition is the same for l', D' and q' and has to do minor adjustmends of the CDR set C'. (Since some clocks are reset, certain CDRs from C' which potentially do not hold anymore are removed, and new CDRs are included by applying $\mathsf{RegionCDRs}$ and $\mathsf{Closure}_{D'}$ after resetting the clocks). And for the *time pass* transition, $C' = C$ and the rest of the state information changes as before.

Overapproximation: The described LTS *refined symbolic semantics* is still an overapproximation as the one described in Section 4.3, since the only change is the CDR set in the state information. This additional state information is constructed in a way that for each state (l, ν, q) reachable in the (alternative) concrete semantics, there is still a state (l, D, C, q) in the described refined symbolic semantics reachable with $\nu \in D$ and $\nu \models C$. However, we conjecture that this approximation is even *exact*, although this remains an open question since there is no proof for this.

5 Conclusion

In this work, the borderline of decidability for certain classes of the schedulability problem is studied. The focus is on the class of variable execution time task models with a preemptive scheduler and without feedback in the single- and multi-processor settings.

First, the problem is surprisingly shown to be undecidable in the multi-processor setting. As a variation of this result, it is also shown undecidable for an extended variant of schedulers in the single-processor setting, namely schedulers with finite memory. This shows that the definition of a scheduler has to be done carefully since a "small" extension like adding just one bit of memory may directly lead to undecidability. It may further give some insight into the structure of non-determinism in these models, since as well as the use of feedback, the memory can be seen as a method to check and remember non-deterministic choices during automata runs. The ability to use this memory is powerful enough to lead to undecidability in both cases.

Second, although the problem for single-processor models is not shown to be decidable, an approximation algorithm has been developed. It is an open question whether this algorithm is exact. It is conjectured that it in fact is,

that is the algorithm solves the decidability problem. To prove this is subject of future work.

A Appendix

A.1 Alternative concrete semantics

We give here the formal definition of the *alternative concrete semantics* of a task automaton as described in Section 4.2.

This semantics uses an *augmented* set of clocks. Let N be the maximal schedulable queue length, then $\hat{\mathcal{C}} = \mathcal{C} \cup \{x_j^c, x_j^d\}_{j=1}^N$ is an extension of the original set $\mathcal{C}$ with additional two clocks for each task instance in the queue. A clock valuation for this augmented set is denoted with $\hat{\nu}$. Further, a *discrete queue* $[P_1(s_1), \ldots]$ is a task queue keeping just information about the task type and the task state $s_j \in \{\mathsf{Running}, \mathsf{Preempted_by_}k, \mathsf{Waiting}\}$. The set of all discrete queues is denoted by $Q_{\mathcal{P}}^{\mathsf{discrete}}$. Because of this discrete queue, we also use a *discrete* version $\hat{\mathsf{Sch}}$ of the scheduling function Sch. Instead of using time values b_j, w_j and d_j for each task instance in the queue, it uses the clocks x_j^c, x_j^d for decisions (by substituting b_j, w_j and d_j by expressions using the constants B_i, W_i and D_i and the clocks x_j^c, x_j^d as in Section 2.4.2). The function $\hat{\mathsf{Sch}}$ takes also care of changing the task states s_j correctly. We use another function $\mathsf{InsertPos} : \mathcal{P} \times Q_{\mathcal{P}}^{\mathsf{discrete}} \to \mathbb{N}$ which returns the position of the queue, where $\hat{\mathsf{Sch}}$ inserts the new task. This function $\mathsf{InsertPos}$ is directly derived from Sch and can as well be implemented as a timed automaton accordingly. (In fact, Sch can be defined using $\mathsf{InsertPos}$.)

The state $s \in S := N \times \mathcal{V}(\hat{\mathcal{C}}) \times Q_{\mathcal{P}}^{\mathsf{discrete}}$ of a task automaton in this semantics is a tuple of a location, an augmented clock valuation and a discrete queue. The labels $\Sigma = \mathcal{A}ct \cup \mathbb{R}_{\geq 0} \cup \{\mathsf{fin}\}$ are either an event, a time pass value or the special task finishing label fin.

Definition 12. *Given a scheduling strategy* Sch*, the* alternative concrete semantics *of an automaton* $A = \langle N, l_0, E, I, M, x_{done} \rangle$ *is a labeled transition system* $[\![A_{\mathsf{Sch}}]\!] = \langle S, s_0, \Sigma, T \rangle$ *with the states* S*, initial state* $s_0 = (l_0, \tilde{\nu}_0, [])$*, labels* Σ *and transitions* T *defined by the following rules:*

- $(l, \hat{\nu}, q) \xrightarrow{a}_{\mathsf{Sch}} (l', \hat{\nu}'[\hat{r}], \hat{\mathsf{Sch}}(M(l'), q))$ *if* $l \xrightarrow{g\,a\,r} l'$*,* $\hat{\nu} \models g$*, and* $\hat{\nu}'[\hat{r}] \models I(l')$ *where*
  - $\hat{r} = r$ *if* $M(l')$ *not defined,* *otherwise* $\hat{r} = r \cup \{x_j^c, x_j^d\}$ *for* $j = \mathsf{InsertPos}(M(l'), q)$*,*
  - $\hat{\nu}'$ *is* $\hat{\nu}$ *with clocks shifted according to* $\mathsf{InsertPos}(M(l'), q)$
- $(l, \hat{\nu}, q) \xrightarrow{t}_{\mathsf{Sch}} (l, \hat{\nu} + t, q)$ *if* $t \in \mathbb{R}_{\geq 0}$*,* $(\hat{\nu} + t) \models I(l)$ *and in case* $q \neq []$ *it holds that* $(\hat{\nu} + t)(x_1^c) \leq W_1$
- $(l, \hat{\nu}, q) \xrightarrow{\mathsf{fin}}_{\mathsf{Sch}} (l, \hat{\nu}', q')$ *if* $B_1 \leq \hat{\nu}(x_1^c) \leq W_1$ *where:*

- *B_1 and W_1 are the worst/best case calculation time constants for the leading task in q,*
- *q' is q with the first task P_1 removed and the state s_2 of the (new leading) task P_2 is set to* Running *in case $|q| > 1$,*
- *$\hat{\nu}'$ is created from $\hat{\nu}$ by*
  1. *Subtracting $\hat{\nu}(x_1^c)$ from $\hat{\nu}(x_j^c)$ for all $1 < j \leq |q|$ where $s_j =$* Preempted_by_k *for some k,*
  2. *Shifting all clock values 1 index down and*
  3. *Resetting the clock value for the new x_1^c in case $s_2 =$* Waiting.

This transition system is bisimilar to the *concrete semantics* from Definition 7 and can be considered equivalent. It can be proven that the following relation is indeed a bisimulation.

Definition 13. *Let $A = \langle N, l_0, E, I, M, x_{done} \rangle$ be a task automaton. We define $(l, \nu, q) \sim (\hat{l}, \hat{\nu}, \hat{q})$ for two states of* concrete semantics *and* alternative concrete semantics *with:*

$l = \hat{l}$, $\forall x \in \mathcal{C} : \nu(x) = \hat{\nu}(x)$ and for the queues $q = [P_1(b_1, w_1, d_1), \dots]$ and $\hat{q} = [\hat{P}_1(\hat{s}_1), \dots]$ the following holds for all j:

- $P_j = \hat{P}_j$,
- *If $\hat{s}_j =$* Running *then: $j = 1, b_1 = \hat{B}_1 - \hat{\nu}(x_1^c), w_1 = \hat{W}_1 - \hat{\nu}(x_1^c), d_1 = \hat{D}_1 - \hat{\nu}(x_1^d)$*
- *If $\hat{s}_j =$* Preempted_by_k *then: $j > 1, b_j = \hat{B}_j - \left(\hat{\nu}(x_j^c) - \hat{\nu}(x_k^c)\right), w_j = \hat{W}_j - \left(\hat{\nu}(x_j^c) - \hat{\nu}(x_k^c)\right), d_j = \hat{D}_j - \hat{\nu}(x_j^d)$,*
- *If $\hat{s}_j =$* Waiting *then: $j > 1, b_j = \hat{B}_j, w_j = \hat{W}_j, d_j = \hat{D}_j - \hat{\nu}(x_j^d)$*

A.2 Decrement part of the reduction automaton

In Figure 14 we give a part of the reduction automaton described in Section 2.4.1 and extended in Section 3.2. The figure shows a part corresponding to a decrement operation of the simulated two-counter machine. Note that the calculation times for the T_{chk1}, T_{chk2} and T_{mark} tasks (which are scheduled to q_2 instead of q_1 like P and Q) sum up to 16. This is also the time for the whole decrement procedure, therefore keeping the load of the second processor at exactly 100%. Note further that for the sake of presentation, we talked in Section 2.4.1 about clocks x_C and x_{Cnew}, where x_{Cnew} will contain the new value for x_C. In this figure here, the construction is a bit different. The value of x_C is first copied to a clock x_{copy}, so that x_C directly gets its new value. There is no structural difference otherwise in the construction.

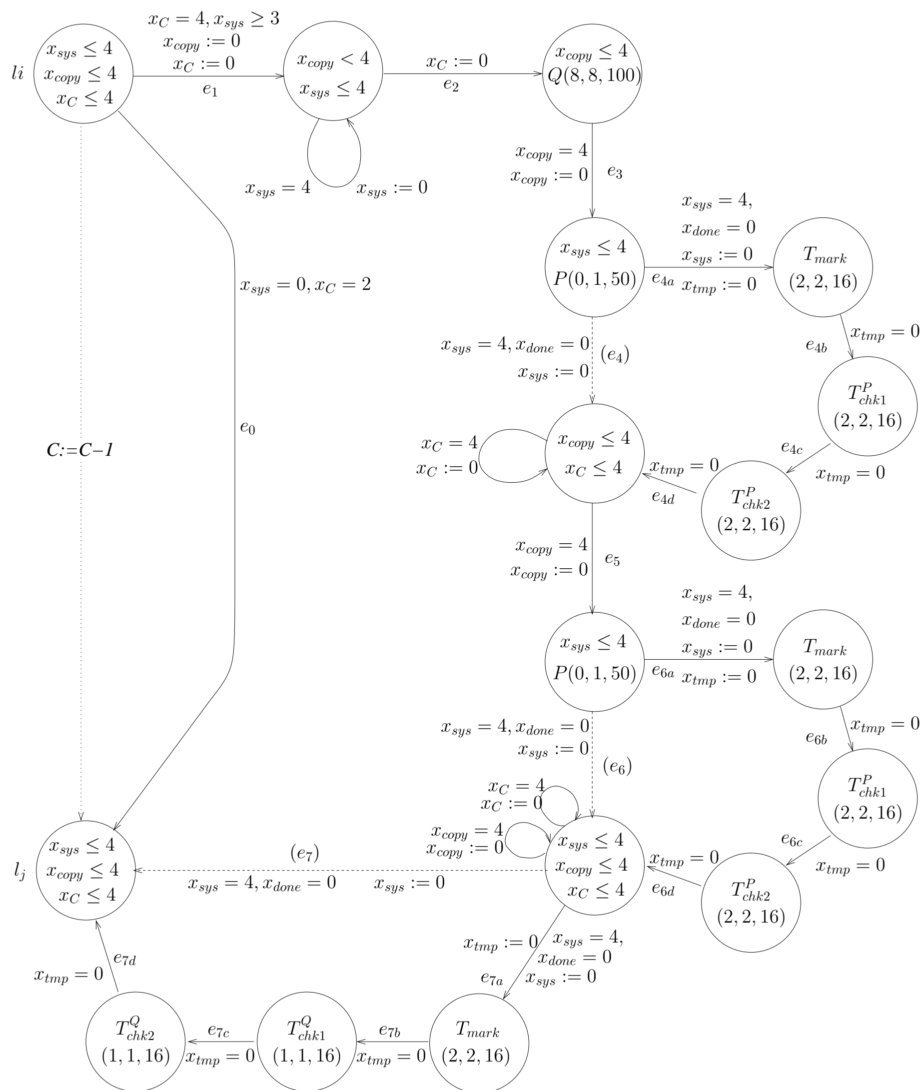

Figure 14: A part of the reduction automaton which corresponds to a decrementation of C. The wrapping edges for clocks x_C, x_{copy}, and for all clocks in locations l_i, l_j are omitted. The location invariants $x_C \le 4, x_{copy} \le 4$, and $x_{done} \le 4$ are also omitted as well as transitions preventing timestops. Note furter that dashed lines are no real edges but the edges that are replaced to remove all $x_{done} = 0$ guards.

References

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *Lecture Notes on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer–Verlag, 2004.

[EWY99] C. Ericsson, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proc. of the 6*th *International Conference on Real-Time Computing Systems and Applications*, pages 182–189. IEEE Computer Society Press, 1999.

[FKPY07] E. Fersman, P. Krcal, P. Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 2007. To appear.

[HKPV98] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[KP05] Pavel Krčál and Radek Pelánek. On sampled semantics of timed systems. In R. Ramanujam and Sandeep Sen, editors, *Proceedings of FSTTCS'05, Hyderabad, India.*, volume 3821 of *Lecture Notes in Computer Science*, pages 310–321. Springer-Verlag, 2005.

[KSY07] Pavel Krcal, Martin Stigge, and Wang Yi. Multi-processor schedulability analysis of preemptive real-time tasks with variable execution times. 2007. Submitted to FORMATS'07.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.