Scheduling for multi-threaded real-time programs via path planning

Thao Dang[1] and Philippe Gerner[2]

[1] VERIMAG,
Centre Equation,
2 avenue de Vignate, 38610 Gières, France
[2] MODULOPI
6, place de l'Homme de Fer, 67000 Strasbourg, France

Abstract. The paper deals with the problem of computing schedules for multi-threaded real-time programs. In [14] we introduced a scheduling method based on the geometrization of PV programs. In this paper, we pursue this direction further by showing a property of the geometrization that permits finding good schedules by means of efficient geometric computation. In addition, this geometric property is also exploited to reduce the scheduling problem to a simple path planning problem originating from robotics, for which we developed a scheduling algorithm using probabilistic path planning techniques. These results enabled us to implement a prototype tool that can handle models with up to 100 concurrent threads.

1 Introduction

Increasing demands on new functions and features of embedded systems make these systems more and more complex. Parallel programming is a way to handle their complexity, and embedded platforms can now support such programming, such as in C or Java. On the other hand, a key feature of embedded systems is that they interact with a physical environment in real time. But analyzing real-time behavior of concurrent programs is a difficult task. Indeed when each part of such a program has its own real-time characteristics, their interaction often makes the real-time behaviour of the whole program very complex. In this paper we deal with a class of concurrent programs which consist of several threads that share some (data) resources. Each resource has a limited capacity defined by the number of threads that it can serve at the same time. The real-time characteristics of the threads are known, and we are interested in the behavior of the program when all the threads run concurrently.

In particular, our goal is to find a good real-time schedule for such a program. More precisely, this schedule specifies how the resources should be shared by the threads, so that the serving capacity of each resource is respected and, in addition, the execution time of the program is as small as possible. To determine such a schedule, one needs to resolve the conflicts between two or more threads that happen when their simultaneous demand for the same resource exceeds the serving capacity of that resource. A resolution here means a decision to which threads to give the resource and which threads have to wait until the resource is released. Note that this program may be part of a larger program (for example, the body of an infinite loop). In the design of an embedded system, an important advantage of such a schedule is that it guarantees that all the program executions are deadlock free. Additionally, it provides a guaranteed worst case execution time (also called worst-case response time), and from the schedule the designer can gain a lot of insight about other properties of the program executions, such as the frequency and duration of waits.

The approach we use to solve this problem can be summarized by two main ideas. First, to model the behavior of real-time multi-threaded programs, we use a timed extension of the PV programs. Second, the scheduling problem is solved by combining an abstraction of program executions and the use of geometric properties of the model. These ideas were initiated in our previous work [14], and this paper pursues this direction further. Its main novelty is the discovery of a geometric property of PV programs, which makes the search for good schedules more efficient via simple geometric computation. In addition, this geometric property is also exploited to reduce the scheduling problem to a simple path planning problem in robotics, for which we developed a randomized search algorithm, inspired by probabilistic path planning techniques. These results enabled us to handle models with up to 100 concurrent threads.

The paper has three main parts. In the first part, we describe the PV program model, its geometric representation and then formulate our scheduling problem. The definitions and notions introduced in this part are necessary for the developments that follow. In the second part, we present the above mentioned geometric property of the model and show how it is used to solve the scheduling problem. This constitutes the main theoretical result of the paper. The last part is devoted to some experimental results, a discussion on related work and some concluding remarks.

2 Timed PV Programs and Diagrams

In the paper, letters in bold are often used to denote vectors and subscripts to denote the components of a vector, such as a_i is the i^{th} component of vector $\boldsymbol{a}$. Supercripts are often used to denote the elements of a sequence, such as a^i is the i^{th} element of sequence $\{a^0, a^1, \dots, a^m\}$.

In this section we describe how to model real-time behavior of multi-threaded programs using PV programs, a model introduced by Dijkstra [10]. The reader is referred to [15, 12, 17] and the references therein for the results on the application of this model in the analysis of concurrent programs. We model each thread as a process, and a set of threads running together is modeled as a PV program. In the PV vocabulary, P stands for “lock”, and V stands for “unlock” or “release”; it is however important to emphasize that, in our modeling framework, the actions “P” and “V” model the events of taking and releasing a resource, and they do not necessarily mean locking and unlocking a resource in a concrete implementation. In other words, they are used to specify resource usage constraints, that is some resources need to be used in a certain order and within some amount of time. A classical PV program example, called the Swiss flag example, is as follows:

$$A = \bot_A.P_a.P_b.V_b.V_a.\top_A, \; B = \bot_B.P_b.P_a.V_a.V_b.\top_B \qquad (1)$$

where `a` and `b` are resources whose serving capacity is 1. We assume that the threads can always run concurrently, that is a thread can run as soon as it gets all the required resources. Hence, in this example, both threads `A` and `B` are assumed to have their own processor to run on. A model for the cases where the threads have to share processors was proposed in [14]. In the following, we give the formal definition of the model.

Resources and Threads. The shared resources are represented by a set $\Re$ of `resource names`. Each resource has a serving capacity[3], which is represented by a function $limit : \Re \to \mathbb{N}_+$. To model resource usage, we consider two types of `resource actions`: taking and releasing a resource $r \in \Re$, denoted respectively by P_r and V_r.

We consider a set of N `threads`: $E_1, \dots, E_N$. Each thread E_i is a total order of `events`. Each event e has an associated resource action, for example P_r. The order relation of E_i is denoted by $\sqsubseteq_{E_i}$ and is also written simply $\sqsubseteq$ when the context is clear. Each thread E_i contains at least two special events: its `start event` $\bot_{E_i}$ and its `end event` $\top_{E_i}$, which are respectively the bottom and top elements of the order. The threads are assumed to be *well-behaved*, in the sense that each resource should be released before it is taken again by the same thread. We say that thread E_i is `accessing` resource r at event e iff P_r has occurred before or at e and, additionally, the corresponding release action V_r occurs (strictly) after e.

The running together of N threads is modeled by the product $\mathcal{E} = \prod_{i=1,\dots,N} E_i$. We denote by $\preccurlyeq$ the order of $\mathcal{E}$, which is defined componentwise. An element of $\mathcal{E}$ is called a `state` and often denoted by the letter $\boldsymbol{\varepsilon}$, and thus $\boldsymbol{\varepsilon}_i$ is its event on thread E_i. We denote by $\bot = (\bot_{E_1}, \dots, \bot_{E_N})$ the `bottom state` of $\mathcal{E}$ and by $\top = (\top_{E_1}, \dots, \top_{E_N})$ its `top state`.

If B is a partial order and $b, b' \in B$ are such that $b \sqsubset b'$, the pair of these elements is called an `arc` and denoted by $\langle b, b' \rangle$. Also, if B is a total order and if $b \in B$ and $b \neq \bot_B$, then $\text{pred}_B(b)$ denotes the direct

[3] In the PV vocabulary we say that the resource is protected by a semaphore.

predecessor of b in B, that is $\mathrm{pred}_B(b) \sqsubseteq b' \sqsubset b \implies b' = \mathrm{pred}_B(b)$. When the order is clear from the context, we simply write $\mathrm{pred}(b)$. The notion of direct successors can be defined similarly.

Task duration. Our version of timed PV programs [14] is an enrichment of the classic PV program model with a task duration between every two consecutive events of each thread. Indeed, in practical real-time programming, one may estimate the duration of the execution of the program code between two events. Such estimations are usually done to account for the worst cases; this duration is a *worst-case execution time* (WCET). So we associate with each event of a thread the duration (or the WCET) of the task corresponding to the part of the program code which is run between the occurrences of this event and of its direct successor. When event $e \in E_i$ occurs, we say that thread E_i starts task e. We denote by E the union $\bigcup_{i=1,\ldots,N} |E_i| \setminus \{\top_{E_i}\}$, where $|E_i|$ is the set of events of E_i. Thus, the *task durations* are given with a function $d : E \to \mathbb{R}_+$, and for every thread E_i, $d(\top_{E_i}) = 0$. In this work we do not consider tasks with duration 0. As an example, the following is a timed version of the Swiss flag program:

$$A = \bot_A.1.P_a.1.P_b.2.V_b.5.V_a.2.\top_A,\ B = \bot_B.1.P_b.4.P_a.1.V_a.1.V_b.1.\top_B$$

The numbers between the actions are the task durations. For example, the first number 1 in thread A is the duration of the task to be executed between the beginning of thread A and its first action $P_{\mathtt{a}}$.

Forbidden States. A state $\boldsymbol{\varepsilon} \in \mathcal{E}$ is said to be `forbidden` if at $\boldsymbol{\varepsilon}$ there is at least one resource to which the number of concurrent accesses is greater than its limit, that is $\exists r \in \Re : \sum_{i=1,\ldots,N} \mathit{accessing}_i(r, \boldsymbol{\varepsilon}) > \mathit{limit}(r)$ where $\mathit{accessing}_i(r, \boldsymbol{\varepsilon}) = 1$ if thread E_i is accessing resource r at $\boldsymbol{\varepsilon}_i$ and $\mathit{accessing}_i(r, \boldsymbol{\varepsilon}) = 0$ otherwise. We denote by $\mathcal{F}$ the set of all forbidden states of $\mathcal{E}$, and by $\mathcal{A}$ the set of all `allowed` states, which is the complement of $\mathcal{E}$.

Strings. An arc $\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle$ is called a `small step` if $\forall\, i \in \{1, \ldots, N\} :\ \mathrm{pred}(\boldsymbol{\varepsilon}'_i) \sqsubseteq_{E_i} \boldsymbol{\varepsilon}_i \sqsubseteq_{E_i} \boldsymbol{\varepsilon}'_i$.

Definition 1. *A* `string` *s is a total suborder of $\mathcal{E}$ such that for each state $\boldsymbol{\varepsilon}$ in $s \setminus \{\bot_s\}$, the arc $\langle \mathit{pred}_s(\boldsymbol{\varepsilon}), \boldsymbol{\varepsilon} \rangle$ is a small step.*

Note that we define a string as a subset of $\mathcal{E}$. A string that does not contain a forbidden state and hence does not induce any resource access conflicts, is called a `feasible string`. We remark that in our previous work [14], a string is defined as a subset of $\mathcal{A}$ and thus all strings are by definition feasible. The idea of not restricting to elements of $\mathcal{A}$ is to separate time constraints and resource constraints in order to model time more explicitly, as we shall show later.

Timed execution. The above notion of strings does not capture time information. To this end, we introduce the notion of timed state and timed execution. A `timed state` is a pair $\boldsymbol{\mu} = (\boldsymbol{\varepsilon}, t)$ where $\boldsymbol{\varepsilon} \in \mathcal{E}$ and t is a non-negative real number. Given a timed state $\boldsymbol{\mu} = (\boldsymbol{\varepsilon}, t)$, it is called `forbidden` if $\boldsymbol{\varepsilon}$ is a forbidden state. The meaning of $(\boldsymbol{\varepsilon}, t)$ is that at time point t the latest event on thread E_i is $\boldsymbol{\varepsilon}_i$.

A sequence of timed states $\gamma = \boldsymbol{\mu}^1, \ldots, \boldsymbol{\mu}^m = (\boldsymbol{\varepsilon}^1, t^1), \ldots, (\boldsymbol{\varepsilon}^m, t^m)$ is called a `timed execution`. It specifies the exact time points at which the threads perform the resource actions. A timed execution is `feasible` iff none of its elements is forbidden. In addition, a timed execution is said to be `consistent` iff the event order and time constraints of all the threads are respected.

Definition 2. *A timed execution $\gamma = \boldsymbol{\mu}^1, \ldots, \boldsymbol{\mu}^m = (\boldsymbol{\varepsilon}^1, t^1), \ldots, (\boldsymbol{\varepsilon}^m, t^m)$ is* `consistent` *iff the following conditions are satisfied for each thread E_i, $i \in \{1, \ldots, N\}$:*

1. *The sequence $\boldsymbol{\varepsilon}^1_i, \ldots, \boldsymbol{\varepsilon}^m_i$ is a string.*
2. *For each $j \in \{2, \ldots, m-1\}$ such that $\boldsymbol{\varepsilon}^j_i \neq \boldsymbol{\varepsilon}^{j-1}_i$, let $j' \in \{1, \ldots, m-1\}$ be the smallest index strictly greater than j such that $\boldsymbol{\varepsilon}^{j'}_i \neq \boldsymbol{\varepsilon}^j_i$. If such j' exists, then $t^{j'} - t^j \geq d(\boldsymbol{\varepsilon}^j_i)$.*

The duration of γ is defined by $d(\gamma) = t^m - t^1$. A `feasible consistent timed execution` *$\gamma = \boldsymbol{\mu}^1, \ldots, \boldsymbol{\mu}^m = (\boldsymbol{\varepsilon}^1, t^1), \ldots, (\boldsymbol{\varepsilon}^m, t^m)$ with $\boldsymbol{\varepsilon}^1 = \bot$ and $\boldsymbol{\varepsilon}^m = \top$ is called a* `timed schedule`*.*

The first condition guarantees that the required task order (or event order) of each thread is respected, and the second condition guarantees that the task duration constraints are satisfied. The above definition implies the time progress property of a consistent timed execution since its sequence $t^1, \ldots, t^m$ is strictly increasing.

Scheduling problem. We can now formally state our scheduling problem as computing a timed schedule with the shortest duration, which we simply call a `shortest` or `optimal` schedule. To find such a timed schedule, we use strings to abstract timed schedules and geometric properties of PV diagrams to compute this abstraction. This will be discussed in the rest of this section. We defer a discussion on related models and problems, in particular timed automata and job-shop scheduling, to Section 5.

Before continuing, we remark that due to the complexity of real-life systems, finding an optimal schedule often requires prohibitive computation time, and hence a problem of great interest is to compute `good` or `short` schedules (that is, those close to the optimal ones) in a reasonable time. This is indeed our practical goal, and therefore although the theoretical results in the paper address the optimality criterion, we also propose a practical non-exhaustive method to achieve a good trade-off between computation time and optimality.

2.1 Geometrization

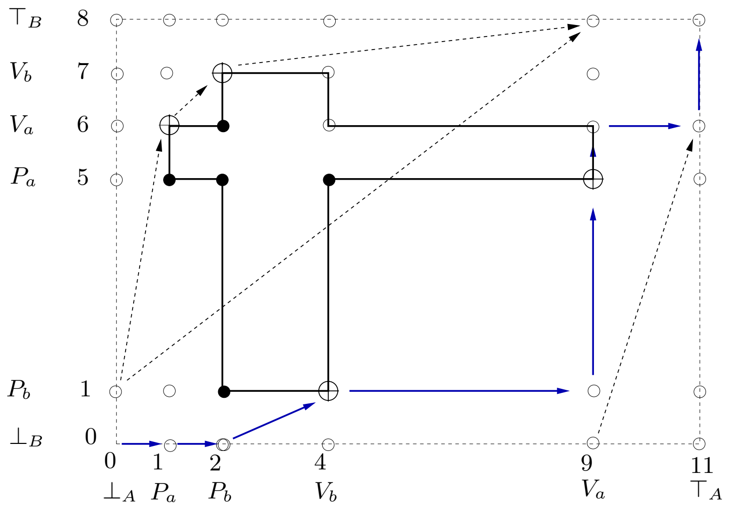

Fig. 1. The PV diagram of the timed Swiss flag program

Each PV program has a geometric representation which is its PV diagram. The PV diagram of the Swiss flag program is shown in Figure 1. In this diagram, a schedule can be represented by a sequence of small steps from $(\bot_A, \bot_B)$ to $(\top_A, \top_B)$. The figure shows a feasible schedule drawn in solid arrows. The black circles indicate the forbidden states. For example, point $(2, 1)$ is forbidden because its associated combination of actions $(P_{\mathtt{b}}, P_{\mathtt{b}})$ means that both threads are accessing resource `b` at the same time, which is not possible since the serving capacity of `b` is 1. An important advantage of such diagrams is that they allow to 'visualize' special behaviors of a program, for example we can see two special cases: point $(1, 1)$ corresponds to a *deadlock* and point $(4, 6)$ to *unreachable state.*

We now formalize the geometric representation. We use the notation "$\bar{\ }$" for the mapping. Each thread E_i is mapped onto a subset of $\mathbb{R}$ by specifying for each event $e \in E_i$ an ordinate $c(e)$. Hence, roughly speaking, a `geometrization` is a mapping of its executions to trajectories in a subset of $\mathbb{R}^N$. In [14], we introduced a geometrization which maps the product of the threads E_i $(1 \leq i \leq N)$ to $\mathbb{Z}^N$. We also proposed a method to find short schedules using this geometrization. This method does not depend on the scaling of the diagram, that is the task durations need not be modeled precisely. To exploit its geometric properties further, in this paper we use a geometrization where the diagram is scaled according to the task durations, which is called `exact scaling geometrization`. The ordinates are chosen so as to visually reflect the task durations. More formally, the ordinates are defined inductively as follows:

$$\begin{cases} c(\bot_{E_i}) = 0, \\ c(e) = c(\mathrm{pred}_{E_i}(e)) + d(\mathrm{pred}_{E_i}(e)) \ \text{ if } \ e \neq \bot_{E_i}. \end{cases}$$

The order of E_i is thus mapped to the order $\leq$ between the real numbers $c(e)$, and $\overline{E_i}$ is the resulting total order $(\{c(e) \mid e \in |E_i|\}, \leq)$. This mapping is clearly an isomorphism of total orders. The geometrization of $\mathcal{E}$ is defined as the product of partial orders $\overline{\mathcal{E}} = \prod_{i=1,\ldots,N} \overline{E_i}$ and is isomorphic to $\mathcal{E}$.

Mapping states. Geometrically speaking, in $\mathbb{R}^N$, the geometrization $\overline{\mathcal{E}}$ forms a non-uniform N-dimensional grid $\mathcal{G}$ over the bounding box $\mathcal{B} = [0, c(\top_1)] \times \ldots \times [0, c(\top_N)] \subset \mathbb{R}^N$. Every state $\boldsymbol{\varepsilon} = (\boldsymbol{\varepsilon}_1, \ldots, \boldsymbol{\varepsilon}_N) \in \mathcal{E}$ is mapped to $\overline{\boldsymbol{\varepsilon}} = (c(\boldsymbol{\varepsilon}_1), \ldots, c(\boldsymbol{\varepsilon}_N)) \in \mathbb{R}^N$ which is a `grid point`.

The set $\mathcal{F}$ of forbidden states is mapped to the set $\overline{\mathcal{F}}$ of `forbidden points`, which have an intuitive geometric interpretation. Given a box $B = [l_1, u_1] \times \ldots \times [l_N, u_N]$, its associated right-open box is defined as $B' = \{\boldsymbol{x} \mid \forall i \in \{1, \ldots, N\} : l_i \leq \boldsymbol{x}_i < u_i\}$ where $\boldsymbol{x}_i$ is the i^{th} coordinate of the point $\boldsymbol{x}$. For every $\boldsymbol{\varepsilon} \in \mathcal{F}$, let $box(\boldsymbol{\varepsilon})$ be the elementary box whose bottom left vertex is $\overline{\boldsymbol{\varepsilon}}$. An elementary box is a box such that all its vertices are grid points and additionally its interior does not contains any grid points. Then, the associated right-open box of $box(\boldsymbol{\varepsilon})$ is called the elementary forbidden box associated with $\boldsymbol{\varepsilon}$, denoted by $obox(\boldsymbol{\varepsilon})$. The union of all such boxes $P_F = \bigcup_{\boldsymbol{\varepsilon} \in \mathcal{F}} obox(\boldsymbol{\varepsilon})$ is called the `forbidden region` (whose closure is indeed a non-convex polyhedron with axis-parallel faces). In Figure 1 the forbidden region has the form of the Swiss flag. Due to the time progress property of timed schedules, we can prove that a feasible schedule never enters the forbidden region. Similarly, the `allowed region` is defined as $P_A = \mathcal{B} \setminus P_F$.

Mapping strings. A string s is mapped to a sequence $\overline{s}$ of grid points in the bounding box $\mathcal{B}$. In view of exploiting continuous geometric properties, we also define the continuous geometrization of an arc $\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle$ as the directed line segment from vertex $\overline{\boldsymbol{\varepsilon}}$ to vertex $\overline{\boldsymbol{\varepsilon}'}$ and denote it by $\overline{\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle}$. The reason for this choice is that there is a relation between the feasible strings and the corresponding line segments that we shall show later.

2.2 Discrete abstraction of timed executions

In our timed PV program model, a string can be thought of as a discrete abstraction of timed executions. Indeed, one string corresponds to a uncountable number of timed executions. We define the `duration of a string` as the duration of a shortest consistent timed execution corresponding to the string. In other words, it is the duration of such a timed execution where all the resource actions are taken as soon as possible. In [14], we showed an algorithm to determine the duration of a string, which can be seen as a constructive definition of this notion.

In particular, we are interested in strings with no unnecessary wait, which are called `eager` strings. A thread waits out of necessity when its next resource is unavailable. In Figure 1, an example of a non-necessary wait is a schedule that would go, for example, through points $(4, 0)$ and $(9, 0)$ before going to $(9, 1)$, which means that thread B waits until thread A releases resource a before accessing resource b while resource b is already available. Notice that a shortest schedule is necessarily eager; the other direction is however not true in general. The notion of bow that we describe in the following is indeed a way to abstract eager strings. The main idea is to see whether it is possible to make a 'big' step instead of small steps as in the definition of strings.

Definition 3. *Given an arc $\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle$ from $\mathcal{A}$,*

- *The* `tightened length` *of the arc $\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle$, denoted by $d(\langle \boldsymbol{\varepsilon}', \boldsymbol{\varepsilon} \rangle)$, is the duration of a shortest feasible string from $\boldsymbol{\varepsilon}$ to $\boldsymbol{\varepsilon}'$ if at least one such feasible string exists; otherwise, $d(\langle \boldsymbol{\varepsilon}', \boldsymbol{\varepsilon} \rangle) = +\infty$.*
- *The* `max distance` *of $\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle$ is $\|\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle\| = \max_{i=1,\ldots,N}(c(\boldsymbol{\varepsilon}'_i) - c(\boldsymbol{\varepsilon}_i))$.*

– *The arc* $\langle \varepsilon, \varepsilon' \rangle$ *is called a* `bow` *iff* $d(\langle \varepsilon, \varepsilon' \rangle) = \|\langle \varepsilon, \varepsilon' \rangle\|$.

Geometrically speaking, the max distance $\|\langle \varepsilon, \varepsilon' \rangle\|$ is the longest side of the box whose bottom left and top right vertices are $\overline{\varepsilon}$ and $\overline{\varepsilon'}$. It is easy to see that one cannot expect to obtain a string from ε and ε' with duration shorter than $\|\langle \varepsilon, \varepsilon' \rangle\|$ since it is exactly the time needed to execute the longest thread without waiting (i.e. without interruption). On the other hand, if there is at least one string from ε and ε' with duration equal to $\|\langle \varepsilon, \varepsilon' \rangle\|$, then the arc $\langle \varepsilon, \varepsilon' \rangle$ is called a bow. As an example, in Figure 1, the arc $\langle (9, 0), (11, 6) \rangle$ is a bow, while the arc $\langle (0, 1), (9, 8) \rangle$ is not. Indeed, the latter has the max distance $\|\langle (0, 1), (9, 8) \rangle\| = 9$, while its tightened length is 13 since a shortest string $(0, 1), (1, 6), (2, 7), (9, 8)$ exchanges resource b at point $(1, 6)$, and thread A has to wait until this exchange for at least 4 time units.

Remark 1. Given a feasible string s consisting of two consecutive small steps $\langle \varepsilon, \varepsilon' \rangle$ and $\langle \varepsilon', \varepsilon'' \rangle$, let $d(s)$ denote the duration of s. Then, $d(s) \leq \|\langle \varepsilon, \varepsilon' \rangle\| + \|\langle \varepsilon', \varepsilon'' \rangle\|$.

The above remark can be explained with a simple program that has 2 concurrent threads. We first determine the smallest time δ needed to follow the first small step $\langle \varepsilon, \varepsilon' \rangle = \langle (\varepsilon_1, \varepsilon_2), (\varepsilon'_1, \varepsilon'_2) \rangle$. If $\varepsilon_i \neq \varepsilon'_i$, we know that at least $[c(\varepsilon'_i) - c(\varepsilon_i)]$ time units have passed on thread E_i. Suppose that $\varepsilon_1 = \varepsilon'_1$ and $\varepsilon_2 \neq \varepsilon'_2$. Since no new event has occurred on E_1, the lower bound of the time lapse on E_1 is 0, and thus the global time lapse $\delta = [c(\varepsilon'_2) - c(\varepsilon_2)]$. We proceed with the second small step $\langle \varepsilon', \varepsilon'' \rangle$ and consider the following two cases:

– Case 1: No new event has occurred on E_1, the lower bound of the time lapse on thread E_1 is still 0, and the smallest time needed to follow these two consecutive steps is $[c(\varepsilon''_2) - c(\varepsilon'_2)] + [c(\varepsilon'_2) - c(\varepsilon_2)]$.
– Case 2: A new event occurred on E_1, which allows us to know that the lower bound of the time lapse on thread E_1 is $[c(\varepsilon''_1) - c(\varepsilon'_1)]$. If again $\varepsilon''_2 \neq \varepsilon'_2$, combining the lower bounds of the time lapses on both threads, the smallest time lapse of these two consecutive steps is $max\{[c(\varepsilon''_1) - c(\varepsilon'_1)], [c(\varepsilon''_2) - c(\varepsilon'_2)] + [c(\varepsilon'_2) - c(\varepsilon_2)]\}$. By definition, this is exactly the duration of the string s. We can see that $d(s) \leq max_i\{c(\varepsilon'_i) - c(\varepsilon_i)\} + max_i\{c(\varepsilon''_i) - c(\varepsilon'_i)\} = \|\langle \varepsilon, \varepsilon' \rangle\| + \|\langle \varepsilon', \varepsilon'' \rangle\|$.

The intuition behind this is that only when an event e occurs on a thread we can determine a lower bound of the time lapse on this thread since the occurrence of the previous event. This lower bound imposes a constraint on the global time at event e. When a new event simultaneously occurs on two or more threads, the global time is determined by combining the constraints imposed by all these threads, and we say that in this situation these threads 'synchronize'. Hence, the way of describing the time constraints of each thread on a separate dimension in a timed PV program can be thought of as de-synchronizing them, and the threads need to be re-synchronized only when their interaction affects the global behavior. □

To define the abstraction of eager strings, we additionally need the notion of critical exchange states, which are states where an eager string should wait. An `exchange state` is an element $\varepsilon \in \mathcal{A}$ where a resource can be exchanged, that is there exists at least one resource $r \in \Re$ and two indices i, j such that $\varepsilon_i = V_r$ and $\varepsilon_j = P_r$. An exchange state ε such that $\exists r : \ accessing(r, \varepsilon) = limit(r)$ is called a `critical exchange state`. In the Swiss flag example, the critical exchange states are indicated by circled addition symbols. We sometimes call these states 'exchange states' for short. It is possible to characterize these states geometrically. We can prove that their geometrizations are indeed the boundary points of the forbidden polyhedron that belong to at least one positive face [9]. Note that every face of the forbidden polyhedron is parallel to one of the axes, and a face is called positive if its normal vector points to the positive direction of the axis.

The *abstraction of all the eager strings* (and hence also of all the shortest schedules) is the graph that has all the critical exchange states together with $\perp$ and $\top$ as nodes and bows as edges. We call this graph `the abstraction graph`. More formally, we denote by C the union of all the ctritical exchange states and $\{\perp, \top\}$. The abstraction graph is the weighted graph defined by the binary relation $G \subseteq C \times C$ satisfying: $\varepsilon \, G \, \varepsilon' \iff \langle \varepsilon, \varepsilon' \rangle$ is a bow; and the cost (or weight) of the edge from ε to ε' is the max distance $\|\langle \varepsilon, \varepsilon' \rangle\|$. The cost of a path in the graph G is the sum of the costs of its edges. For $\varepsilon, \varepsilon' \in C$ with $\varepsilon \preccurlyeq \varepsilon'$, we denote by $l(\varepsilon, \varepsilon')$ the duration of a shortest (or the least costly) path in G from ε to ε', and this duration is $+\infty$ if there is no path from ε to ε'. The following theorem [14] states an important property of the abstraction graph G.

Theorem 1 (Abstraction graph). *The duration of a shortest schedule from $\perp$ to $\top$ is the cost of a shortest path in G from $\perp$ to $\top$, that is $d(\langle \perp, \top \rangle) = l(\perp, \top)$.*

The intuitive meaning of the theorem can be explained as follows. It shows a special property of the shortest paths of G: if π is a shortest path, then the cost of π is the duration of a shortest schedule, and π is a abstraction of this schedule. The problem of searching for a shortest schedule is thus reformulated as that of finding a shortest path in the graph G. To construct the abstraction graph, it suffices to consider the critical exchange states. The existence of a bow between two such states indicates that there exists an *eager feasible string* or 'direct route' between these two states.

Computationally speaking, this abstraction is useful only if one can efficiently determine whether an arc is a bow. Checking the condition given in Definition 3 could be complicated since it requires computing the tightened length $d(\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle)$, which is not trivial if one wants to avoid enumerating all feasible strings connecting $\boldsymbol{\varepsilon}$ to $\boldsymbol{\varepsilon}'$. In [14] we proposed a method to do so using a spatial decomposition of the allowed region into boxes. Essentially, if a box does not contain any forbidden points, then any arc whose geometrization is inside the box is a bow. In this work, we intend to exploit the geometrization further in order to be able to quickly find long bows (i.e. long direct routes) allowing to speed up the search for short schedules.

3 Discrete abstraction in relation with geometrization

The following theorem states an important property of bows in relation with the Euclidean intersection in the exact scaling geometrization. This property enables us to efficiently determine whether an arc is a bow in order to construct the abstraction graph.

Theorem 2. *Let $\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle$ be an arc from $\mathcal{A}$. If $\overline{\langle \epsilon, \epsilon' \rangle} \cap P_F = \emptyset$ where P_F is the forbidden region, then $\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle$ is a bow.*

As mentioned earlier, the tightened length $d(\langle \boldsymbol{\varepsilon}', \boldsymbol{\varepsilon} \rangle)$ cannot be smaller than $\|\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle\|$. This means that to prove the theorem, it suffices to find a concrete feasible string from $\boldsymbol{\varepsilon}$ to $\boldsymbol{\varepsilon}'$ whose duration is exactly $\|\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle\|$. By definition of the duration of a string, this also means finding a feasible timed execution with the required duration. The idea of the proof is to construct such a timed execution that we call a *witness timed execution.* This is done by a clipping procedure explained in the following.

3.1 Constructing a witness timed execution

Clipping. Let $\{\boldsymbol{x}^j\}_{1 \le j \le m}$ be the sequence of all intersecting points of the directed line segment $\overline{\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle}$ with the grid planes. A *grid plane* is a hyper-plane which is parallel to one of the axes and contains at least one grid point. We denote this by $\{\boldsymbol{x}^j\}_{1 \le j \le m} = clip_{\mathcal{G}}(\overline{\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle})$ and call this sequence of intersecting points the *clipping* of $\overline{\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle}$ on the grid $\mathcal{G}$. An example of clipping is shown in Figure 2 where $\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^7$ is a sequence of intersecting points. We derive a timed execution from this sequence of intersecting points by mapping each intersecting point to a timed state. Note the time value of a timed state is the absolute time lapse from the beginning of the execution, and in order to determine this absolute time lapse one need to consider the time lapses relative to the occurrence of the events. This is captured by the notion of `relatively-timed events`.

Relatively-timed events. We first remark that a grid point can be directly mapped back to a state in $\mathcal{E}$; however, an intersecting point in the clipping is not necessarily a grid point. Note that a state in $\mathcal{E}$ corresponds to the moment where all the E_i take a resource action. An intersecting point which is not a grid point indeed corresponds to a situation where not all the threads simultaneously perform a resource action.

Given $e \in E_i$ and a real number $\beta \in [0, d(e))$ where $d(e)$ is the duration of task e, $e \oplus \beta$ denotes the relatively-timed event at which *at least* β time units have elapsed since the occurrence of the event e (or since task e is started); β is called the relative time lapse of $e \oplus \beta$. Intuitively, regarding thread E_i, $e \oplus \beta$ is a fictious event because its occurrence is not associated with any resource actions by E_i; however, as we shall see later, it is used to indicate a time point at which at least one or more other threads perform a resource

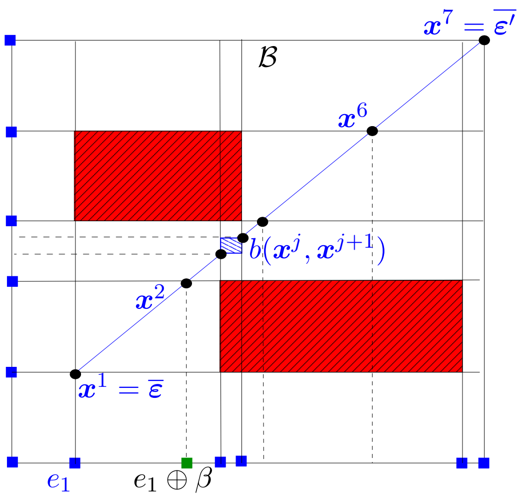

Fig. 2. Geometric realization

action. Then, the set of relatively-timed events of thread E_i is: $\Upsilon_i = \{e \oplus \beta \mid e \in |E_i| \ \wedge \ \beta \in [0, d(e))\}$ where $|E_i|$ is the set of events of E_i. For each relatively-timed event $\rho = e \oplus \beta \in \Upsilon_i$, $event(\rho)$ gives the associated event e.

We associate with each relatively-timed event $e \oplus \beta$ an ordinate $c(e \oplus \beta) = c(e) + \beta$. Hence, an event $e \in E_i$ can be indeed written as a relatively-timed event of the form $e \oplus 0$. The definition of order on the relatively-timed events in Υ_i can be defined as: $e \sqsubseteq_{\Upsilon_i} e'$ iff $c(e) \leq c(e')$.

Remark 2. With respect to a thread, the difference between the relative time lapse β of a relatively-timed event $e \oplus \beta$ and the time component t of a timed event (e, t) (defined in Section 2) is that the latter is an absolute time (i.e. the time lapse from the beginning of the execution), while the former is a relative time (i.e. the time lapse from the occurrence of the last event, which is e).

Relatively-timed states. A vector of relatively-timed events $\boldsymbol{\upsilon} = (\boldsymbol{\upsilon}_1, \ldots, \boldsymbol{\upsilon}_N)$ where $\boldsymbol{\upsilon}_i \in \Upsilon_i$ is called a `relatively-timed state`. We denote $events(\boldsymbol{\upsilon}) = (event(\boldsymbol{\upsilon}_1), \ldots, event(\boldsymbol{\upsilon}_N))$. The order $\preccurlyeq$ on relatively-timed states is defined componentwise, namely $\boldsymbol{\upsilon} \preccurlyeq \boldsymbol{\upsilon}'$ iff $\forall i \in \{1, \ldots, N\} : \ \boldsymbol{\upsilon}_i \sqsubseteq_{\Upsilon_i} \boldsymbol{\upsilon}'_i$, or equivalently $\forall i \in \{1, \ldots, N\} : \ c(\boldsymbol{\upsilon}_i) \leq c(\boldsymbol{\upsilon}'_i)$.

Let us explain the intuitive meaning of relatively-timed states. At relatively-timed state $\boldsymbol{\upsilon} = (\boldsymbol{\varepsilon}_1 \oplus \beta_1, \ldots, \boldsymbol{\varepsilon}_N \oplus \beta_N)$, if the relative time lapse $\beta_i = 0$, then thread E_i is performing the action associated with $event(\boldsymbol{\upsilon}_i)$; if $\beta_i > 0$, thread E_i is performing no resource action and, in addition, at least β_i time units have elapsed since the occurrence of $\boldsymbol{\varepsilon}_i$. It should be noted that by "performing no resource action" we mean that the thread does not take or release a resource but it might continue the current task if this task is not yet finished.

Given two relatively-timed states $\boldsymbol{\upsilon}, \boldsymbol{\upsilon}' \in \Upsilon$ such that $\boldsymbol{\upsilon} \preccurlyeq \boldsymbol{\upsilon}'$, then $\langle \boldsymbol{\upsilon}, \boldsymbol{\upsilon}' \rangle$ is called a `small timed step` if $events(\boldsymbol{\upsilon}) = events(\boldsymbol{\upsilon}')$ or $\langle events(\boldsymbol{\upsilon}), events(\boldsymbol{\upsilon}') \rangle$ is a small step. In other words, $\boldsymbol{\upsilon}$ and $\boldsymbol{\upsilon}'$ may have the same associated events but differ in the relative time lapse vector.

Definition 4. *If $\langle \boldsymbol{\upsilon}, \boldsymbol{\upsilon}' \rangle$ is a small timed step, the time lapse between $\boldsymbol{\upsilon}$ and $\boldsymbol{\upsilon}'$ is defined as: $\Delta(\boldsymbol{\upsilon}, \boldsymbol{\upsilon}') = max_{i \in \{1, \ldots, N\}} \{c(\boldsymbol{\upsilon}'_i) - c(\boldsymbol{\upsilon}_i)\}$.*

The meaning of a small timed step $\langle \boldsymbol{\upsilon}, \boldsymbol{\upsilon}' \rangle$ is that following the arc $\langle \boldsymbol{\upsilon}, \boldsymbol{\upsilon}' \rangle$ involves letting each thread E_i start the task $\boldsymbol{\upsilon}_i$ and run for exactly $\Delta(\boldsymbol{\upsilon}, \boldsymbol{\upsilon}')$ time. If there exists a thread E_i such that $c(\boldsymbol{\upsilon}'_i) - c(\boldsymbol{\upsilon}_i) < \Delta(\boldsymbol{\upsilon}, \boldsymbol{\upsilon}')$, we say that when taking the small step $\langle \boldsymbol{\upsilon}, \boldsymbol{\upsilon}' \rangle$ the thread E_i 'has to wait' because the time lapse required for the thread E_i to reach $event(\boldsymbol{\upsilon}'_i)$ from $event(\boldsymbol{\upsilon}_i)$ is smaller than $\Delta(\boldsymbol{\upsilon}, \boldsymbol{\upsilon}')$.

Mapping points to relatively-timed states. Given a real number $y \in [0, c(\top_i)]$, let e be the event in thread E_i and e' is its direct successor such that $c(e) \leq y$ and $c(e') > y$. Such an ordinate $c(e)$ is denoted by $\lfloor y \rfloor$. Then, we define $\natural_i(y) = e \oplus \beta$ where $\beta = y - \lfloor y \rfloor$.

Definition 5. *1. Given a point $\boldsymbol{x} = (x_1, \ldots, x_N) \in \mathcal{B}$, the map $\natural$ of points to relatively-timed states is defined as: $\natural(\boldsymbol{x}) = (\natural_1(x_1), \ldots, \natural_N(x_N))$.*
2. Given a sequence of points $\{\boldsymbol{x}^j\}_{1\leq j\leq m}$, $\natural(\{\boldsymbol{x}^j\}_{1\leq j\leq m}) = \{\boldsymbol{v}^j\}_{1\leq j\leq m}$ where $\boldsymbol{v}^j = \natural(\boldsymbol{x}^j)$ for all j.

Witness timed execution construction. Using the map $\natural$, from the clipping $clip_{\mathcal{G}}(\overline{\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle}) = \{\boldsymbol{x}^j\}_{1\leq j\leq m}$ we construct the following sequence of relatively-timed states: $\phi = \boldsymbol{v}^1, \ldots, \boldsymbol{v}^m = \natural(\boldsymbol{x}^1), \ldots, \natural(\boldsymbol{x}^m)$. It is not hard to see that each arc $\langle \boldsymbol{v}^i, \boldsymbol{v}^{i+1} \rangle$ is a small timed step and $event(\boldsymbol{v}^1), \ldots, event(\boldsymbol{v}^m)$ is a string. Then, from ϕ we construct a timed execution as follows:

$$\gamma = (event(\boldsymbol{v}^1), t^1), \ldots, (event(\boldsymbol{v}^m), t^m) \tag{2}$$

such that $t^1 = 0$ and for $j > 1 : t^j = \sum_{k=2,\ldots,j} \Delta(\boldsymbol{v}^{k-1}, \boldsymbol{v}^k)$. It is easy to verify that the timed execution γ is consistent. □

To summarize, the construction of a witness timed execution for a bow $\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle$ consists of three steps. In the first step, the clipping of the geometrization $\overline{\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle}$ gives a sequence of intersecting points, each of which corresponds to a moment where at least one thread performs a resource action. In the second step, the intersecting points are mapped to a sequence of relatively-timed states that specify the time lapses necessary to evolve from one state to another in this sequence. These time lapses indeed represent the local time constraints of each thread. In the last step, we combine all the local time constraints to derive the global time constraints in the timed execution γ.

3.2 Proof of Theorem 2

To prove that γ is a witness timed execution, we need to prove that: γ is feasible and its duration is indeed $\|\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle\|$.

We begin by proving the first part, that is, γ does not induce any resource conflicts. Due to space limitation, we present only the main idea of the proof: if a point $\boldsymbol{x}$ is non-forbidden, then the state $events(()\natural(\boldsymbol{x}))$ is non-forbidden (see [9] for a detailed proof). The intuitive meaning of this is that with respect to resource usage, a relatively-timed event $e \oplus \beta \in \Upsilon_i$ with $\beta \in (0, d(e))$ is equivalent to $e \in E_i$, since during the time interval between the occurrences of $e \oplus 0$ and $e \oplus \beta$ no resources have been taken or released by thread E_i. □

We proceed to prove that the duration of γ is $\|\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle\|$. The following intermediate result is a direct consequence of the definition of the duration of a timed execution.

Lemma 1. *The duration of the timed execution defined in (2) is*

$$d(\gamma) = t^m - t^1 = \sum_{1\leq j\leq m-1} \Delta(\boldsymbol{v}^j, \boldsymbol{v}^{j+1}).$$

Geometrically speaking, Definition 4 implies that $\Delta(\boldsymbol{v}^j, \boldsymbol{v}^{j+1})$ is equal to the length of the longest side of the box that has $\boldsymbol{x}^j$ as its bottom left vertex and $\boldsymbol{x}^{j+1}$ as its top right vertex, denoted by $b(\boldsymbol{x}^j, \boldsymbol{x}^{j+1})$ (see Figure 2 for an example). Let k be the dimension corresponding to the longest side of the box $b(\boldsymbol{x}^1, \boldsymbol{x}^m)$. Note that this box has $\overline{\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle}$ as diagonal. It is easy to see that k is also the dimension corresponding to the longest side of each box $b(\boldsymbol{x}^j, \boldsymbol{x}^{j+1})$. Combining this with Lemma 1, we have

$$\begin{aligned} d(\gamma) &= \sum_{1\leq j<m} \Delta(\boldsymbol{v}^j, \boldsymbol{v}^{j+1}) \\ &= \sum_{1\leq j<m} c(\boldsymbol{v}_k^{j+1}) - c(\boldsymbol{v}_k^j) \\ &= c(\boldsymbol{v}_k^m) - c(\boldsymbol{v}_k^1) = c(\boldsymbol{\varepsilon}'_k) - c(\boldsymbol{\varepsilon}_k) = \|\langle \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}' \rangle\| \end{aligned}$$

■

The proof of Theorem 2 is now complete.

Remark 3. The proof of the theorem also provides a method for concretizing strings. Hence, after finding a shortest path in the abstraction graph, one can use the construction in the proof to define a concrete shortest timed schedule.

Intersection test. Before continuing we briefly discuss how we implemented the intersection test. In [14] the forbidden region P_F is represented as a non-convex orthogonal polyhedron [6]. This representation has the advantage of being compact since one needs to keep only one polyhedron. However, due to the complexity of this representation (depending on the number of vertices and faces that is exponential in dimension), the test of intersection between of a line segment and P_F may be expensive in high dimensions. We therefore represent the forbidden region as a list of the forbidden boxes (each of which corresponds to the constraints involving the limited number of accesses to a resource). Then, we test the intersection between the line segment with each box separately using an extension of ray tracing techniques to general dimensions. Since a box can be represented by $2N$ linear constraints, the complexity of this test is polynomial in dimension N.

4 Finding a good schedule via path planning

4.1 Randomized search

Using the bow condition in Theorem 2, we can construct the abstraction graph and then search for a shortest path of the graph. The main problem with this approach is that the number of critical exchange states, which is much smaller than the number of all states, still grows exponentially with the dimension. We therefore propose a non-exhautive solution that uses a randomized search, inspired by the RRT (Rapidly Explored Random Tree), which is one of the successful path planning methods in robotics (see [21] for a survey on the RRT method). Indeed, in the geometrization framework, given two points corresponding to two critical exchange states, by Theorem 2, if the line segment connecting these two points does not intersect with the forbidden region, then a feasible schedule with no unnecessary wait between two states exists (and we can compute it). The problem of constructing the abstraction graph is thus similar to a path planning problem, namely computing a collision-free path between a start point and a goal point in an environment with known obstacles. The constraints on the solution path in the path planning problem arise from the geometry of the obstacles and in our scheduling problem from the geometry of the forbidden region. While the RRT approach, to construct the paths, considers all the points in the obstable-free space, using Theorem 1 only the critical exchange states need to be considered. In addition, the paths we are interested in should satisfy the time progress condition; therefore, the resulting path planning problem is indeed a simple motion planning problem[4] where the robot's motion is governed by the constant derivative dynamics of a clock. This dynamics is easily handled by considering the arcs (which by definition satisfy the time progress condition). In general, the path and motion planning problems are hard, for example the problem of finding a shortest path in 3 dimensions is known to be NP-hard [20]. It is however important to note the simplicity of the obstacles in our problem: they are in fact axis-aligned boxes. Additionally, since our practical goal is to quickly find a good schedule, the good coverage properties of the RRT approach allows achieving a good trade-off between the computation time and the quality of the results.

The method we propose is summarized in Algorithm 1. Essentially, we randomize the selection of the critical exchange states. In order to avoid enumerating *all* such states, the randomized selection is done as follows. Let B_F be the set of all forbidden boxes, hence the forbidden region can be written as $P_F = \bigcup B_F$. We first randomly choose a box in B_F and then randomly choose a vertex of B_F. The procedure is repeated until the sampled vertex is a critical exchange state. The test of critical exchange states is done using the geometric charaterization of these points, mentioned in the previous section. When a new critical exchange point x_g is selected, we call it a (current) *goal* point. We then find the the graph a nearest neighbor x_n of x_g

[4] In a path planning problem, the dynamics of robots are not considered.

Algorithm 1 Randomized search

```
$C = \{\bot, \top\}$, $k = 0$
**repeat**
    $box = random(B_F)$
    $x_g = random(\textsc{Vertices}(box))$
    **if** ($x_g \notin C \ \wedge \ x_g$ is a critical exchange point) **then**
        $x_n = \textsc{Neighbor}(G, x_g)$
        $x = \textsc{feasibleBest}(x_n, x_g)$
        **if** ($x \neq x_n$) **then**
            $\textsc{newEdge}(G, x_n, x)$
        **end if**
        $k{+}{+}$
    **end if**
    $\pi = \textsc{ShortestPath}(G, \bot, \top)$
**until** ($k = K_{max}$)
```

(in the max distance). The computation of the function FEASIBLEBEST is as follows. It checks whether the line segment from x_n to x_g (which should correspond to an arc to guarantee the time progress condition) intersects with the forbidden region. If this intersection is empty, $x = x_g$ and a new edge from x_n to x_g is added in the graph, otherwise it tries to grow the graph from x_n towards the goal point x_g as far as possible, which results in $x \neq x_g$. When the number of nodes reaches K_{max} which is a user-defined parameter, the algorithm searches for a shortest path in the graph and stops if there is no request to proceed by the user. An important ingredient in Algorithm 1 is the search for a nearest neighbor in the graph G. To do so, we use a method to store the coordinates of points in a kd-tree [13] while the bows are still stored as the edges of the graph G. Due to the use of the max distance, the operations on the kd-tree we construct is slightly different than those on classic kd-trees. Nonetheless, due to space limitation we do not describe these computations, which can be found in [9]. When running the above algorithm, the graph grows towards the end point $\top$; it is also possible to grow the graph towards both $\top$ and $\bot$.

In addition, we can prove that when every goal point has a strictly positive probability of being sampled, then the probability that the algorithm discovers a given schedule is always strictly positive. This property is called 'complete resolution' in the context of RRTs (see for example [8]). Moreover, it is possible to biase the exploration using the intuitions provided by the geometrization. Indeed, the max distance and the Euclidian distance are closely related with respect to the definition of duration. For example, a schedule that is close in the Euclidian distance to the diagonal of the bounding box (i.e. connecting $\bot$ and $\top$) is likely to be a short schedule. Therefore, one can use the Euclidian distance as a measure to define a non-uniform sampling of the goal points. More precisely, we can define an heuristics which favors the sampling of the critical exchange states that are close to the diagonal of the bounding box in the Euclidian distance.

4.2 Experimental results

We have implemented the above algorithm for randomized search together with a possibility of biased explorations. The experimental results obtained using the prototype tool on a number of examples are shown in the tables of Figure 4 and Figure 5. We also include in Figure 3 an illustrative picture of the forbidden region and the computed schedule (in white line) of a 3-dimensional example. The first set of examples contains a number of timed versions of the Dining Philosophers problem in various dimensions, which we call the timed N-philosophers problems. The second set contains some well-known job-shop scheduling (JSS) benchmarks. The durations of the schedules obtained using our prototype tool are shown in the column "Duration" of the tables. The goal of this experimentation is to evaluate the scalability and the precision of our geometric approach.

Since there are variants of job-shop problems, we first briefly describe the problems we solved. In these problems, each job is a sequence of operations, one on a machine, and the serving capacity of

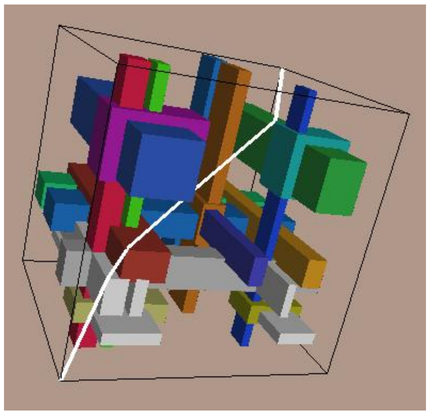

Fig. 3. Schedule for a 3D example

each machine is 1. Indeed, the machines can be modeled by resources and the jobs by threads. Note that the constraints in job-shop scheduling are rather specific. When a machine is needed by two different jobs E_i and E_j, their simultaneous access to this machine corresponds to the following forbidden box: $[c(\bot_1), c(\top_1)] \times [c(\boldsymbol{\varepsilon}_i), c(\boldsymbol{\varepsilon}_{i+1})] \times [c(\boldsymbol{\varepsilon}_j), c(\boldsymbol{\varepsilon}_{j+1})] \ldots [c(\bot_N), c(\top_N)]$ where $\boldsymbol{\varepsilon}_i$ and $\boldsymbol{\varepsilon}_j$ are the events of taking the resource in question by thread E_i and thread E_j, and $\boldsymbol{\varepsilon}_{i+1}$ and $\boldsymbol{\varepsilon}_{j+1}$ are the events of releasing this resource. This box covers the whole range of the bounding box on the dimensions of all other threads E_k with $k \neq i$ and $k \neq j$. On the other hand, the JSS problems are by definition deadlock-free. Note that a general timed PV program allows nested resource actions and thus are richer than these JSS models. However, to test the performance of our approach, we did not try to exploit this particularity and treated the job-shop scheduling problems as if they were problems with more complex types of constraints.

Concerning the timed N-philosophers problems, we do not know their optima, but the computed solutions are clearly non-trivial and appear good in comparison with the worst-case upper bound estimation. On the JSS problems, our method found fairly good schedules in reasonable CPU time. On some of these JSS problems the optima were found. In operation research, there exist numerous methods in operation research specific for JSS (see for example [23]) and recently verification techniques for timed automata were also used to solve the JSS problems. These timed automata based methods employ sophisticated search-order strategies, such as branch-and-bound or using estimates of remaining costs [7, 1]. Our experimental results on JSS problems are not as good as the results obtained by these methods, but they are still reasonably comparable.

In summary, we observe that our method is efficient for quickly finding a reasonable solution of large problems. On the other hand, the number of forbidden boxes is a leading factor for complexity (since it not only determines the number of critical exchange states but also the number of intersection tests to perform). In a JJS problem with M machines and J jobs, the number of forbidden boxes is equal to $\frac{1}{2}MJ(J+1)$ and thus grows quadraticaly with the number of jobs (i.e. the dimension). Therefore, our method is suitable for the problems which could be in high dimension but with a reasonable number of forbidden boxes.

5 Related work

A comparison of our timed PV model with some related models [11, 16, 4] can be found in our previous paper. In this section, we present a comparison of our approach of scheduling with the existing approaches using timed automata [4]. The problem of scheduling using timed automata has been studied in a number of publications [5, 2, 19, 7, 3, 18, 22, 1] and more general optimality criteria (other than execution time) are also considered in some of these work (for example [5, 19, 7]). It is easy to see that timed automata are more expressive than timed PV programs since the former allow to describe more complex synchronization

program	Dim	#forb. boxes	Duration	CPU time (s)
20 phil.	20	20	100	64
50 phil.	50	50	304	248
80 phil.	80	80	490	625
100 phil.	100	100	608	921

Fig. 4. Computation results for some timed N-philosophers problems

program	#j	#m	#forb. boxes	Duration	Known optimum	CPU time (s)
ft06	6	6	90	56	55	66
ft10	10	10	450	992	930	318
abz6	10	5	225	1142	943	851
la01	10	5	225	666	666	646
la05	10	5	225	596	593	87
la16	10	10	450	1047	945	247
la19	10	10	450	1050	842	42
la20	10	10	450	989	902	125
la24	15	10	450	1048	935	269
abz9	20	15	2850	820	679	310

Fig. 5. Computation results for some JSS problems

mechanisms. In addition, a timed PV program can be directly rewritten as a product of timed automata. Each automaton corresponds to a thread and its locations represent the events in our model. Its transitions representing the time constraints have the guards of the form $x > d(e)$ (where x is a clock variable) and clock resets. Thus, using timed automata, one could address the scheduling problem for more complex real-time systems.

Naturally, the geometry resulting from the time constraints in timed automata is more complex than that in timed PV programs. Indeed, in a timed automaton each time constraint is represented by a half-space that could have a slope following the derivatives of the clocks, while in a timed PV program, the half-spaces are all axis-parallel. As mentioned earlier, the reason for this is that in a timed PV program each thread is described separately on one dimension, which can be thought of as a way of 'desynchronizing' them, and then when analyzing the global behavior, the 'synchronization' of local time constraints is handled by the use of max distance. Hence, the geometrization of the product of the threads is, on one hand, very easy to construct, and on the other hand provides a lot of useful insight. The computation is performed on boxes, a geometric object simpler than zones in timed automata. Moreover, the geometry of a PV diagram permits modular combination of its discrete properties (such as to identify special points that contribute to the optimal schedules) and continuous properties (such as to test feasibility of some long direct paths). However, it should be noted that for special cases, such as JSS, one can derive efficient heuristics without manipulating zones (see for example [1]).

6 Concluding remarks

In this paper we described a framework for computing schedules of multi-threaded real-time programs. This framework is based on a combination of techniques from different domains: concurrent processes, motion planning, and computational geometry. The originality of the paper is the way to transform the scheduling problem to a simple path planning problem for which well-developed techniques in robotics can be applied. The paper also shows the computational advantages of PV programs. In fact, their geometry is simple enough to benefit from efficient geometric computations on boxes. The experimental results are encouraging, and we intend to continue this work in various directions. One direction is to extend the model towards more complex specifications, such as those with deadlines and branching. Another direction is to focus on problems with particular geometry (such as the JSS problems). Indeed, it is possible to include optimization that exploits the special structure of the forbidden boxes in these problems. Studying properties of other geometrizations including non-exact-scaling ones is also an interesting theoretical problem to address.

References

1. Y. Abdeddaïm, E. Asarin and O. Maler, Scheduling with timed automata. In *Theoretical Computer Science*, 354(2), pages 272–300, 2006.
2. Y. Abdeddaïm and O. Maler. Job-shop scheduling using timed automata. In *Proc. of the 13th Int. Conf. on Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 478–492. Springer, 2001.
3. K. Altisen, G. Gößler, and J. Sifakis. Scheduler modelling based on the controller synthesis paradigm. *Journal of Real-Time Systems*, 23:55–84, 2002. Special issue on control-theoretical approaches to real-time computing.
4. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
5. R. Alur, S. La Torre, and G. Pappas. Optimal paths in weighted timed automata. In *Proc. of Fourth Int. Workshop on Hybrid Systems: Computation and Control*, volume 2034 of *LNCS*, pages 49–62. Springer, 2001.
6. O. Bournez, O. Maler, and A. Pnueli. Orthogonal polyhedra: Representation and computation. In *Proc. of Hybrid Systems: Computation and Control (HSCC'99)*, volume 1569 of *LNCS*, pages 46–60. Springer, 1999.
7. G. Behrmann and A. Fehnker, Efficient Guiding Towards Cost-Optimality in UPPAAL, In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2031, Springer, 2001.
8. P. Cheng and S. M. LaValle, Resolution complete rapidly-exploring random trees, In Proc. IEEE Int'l Conference on Robotics and Automation, pages 267–272, 2002.
9. T. Dang and Ph. Gerner. On scheduling using PV programs. Technical report, Verimag, IMAG, April 2006.
10. E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–110. Academic Press, New York, 1968.
11. U. Fahrenberg. The geometry of timed PV programs. In *Electronic Notes in Theoretical Computer Science*, volume 81. Elsevier, 2003.
12. L. Fajstrup, E. Goubault, and M. Raussen. Detecting deadlocks in concurrent systems. In *Proc. CONCUR'98*, pages 332–347, 1998.
13. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.
14. P. Gerner and T. Dang. Computing schedules for multithreaded real-time programs using geometry. In Y. Lakhnech and S. Yovine, editors, *Joint International Conferences on Formal Modelling and Analysis of Timed Systems FORMAT and Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT*, LNCS 3253, pages 325–342. Springer-Verlag, 2004.
15. E. Goubault. Schedulers as abstract interpretations of higher-dimensional automata. In *Proc. of PEPM'95 (La Jolla)*. ACM Press, June 1995.
16. E. Goubault. Transitions take time. In *Proc. of ESOP'96, LNCS 1058*, pages 173–187. Springer, 1996.
17. E. Goubault. Geometry and concurrency: A user's guide. *Mathematical Structures in Computer Science*, 10(4), August 2000.
18. C. Kloukinas, C. Nakhli, and S. Yovine. A methodology and tool support for generating scheduled native code for real-time java applications. In R. Alur and I. Lee, editors, *Proc. of the Third Int. Conf. on Embedded Software (EMSOFT'03)*, LNCS 2855, volume 2855 of *LNCS*, pages 274–289. Springer, 2003.
19. K. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. S. Hune, P. Petterson and J. Romijn. As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. *In Proceedings of CAV*, LNSC 2102, pages 493–505, Springer, 2001.
20. J. S. B. Mitchell and M Sharir, New Results on Shortest Paths in Three Dimensions. *Proc. 20th Annual ACM Symposium on Computational Geometry*, pages 124-133, June 9-11, 2004.
21. S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In B. R. Donald, K. M. Lynch, and D. Rus, editors, *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2001.
22. J. I. Rasmussen, K. G. Larsen, and K. Subramani. Resource-optimal scheduling using priced timed automata. In *Proc. of the 10th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, LNCS, pages 220–235. Springer, 2004.
23. W. Nuijten and C. Le Pape. Constraint-Based Job Shop Scheduling with ILOG SCHEDULER. In *J. Heuristics*, 3(4), pages 27–286, 1998.