Toward Parametric Timed Interfaces for Real-Time Components

Youcheng Sun[1,2], Giuseppe Lipari[1,2], Étienne André[3] and Laurent Fribourg[2]
[1]Scuola Superiore Sant'Anna, Pisa, Italy*
[2]LSV, ENS Cachan & CNRS, France
[3]Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France†

We propose here a framework to model real-time components consisting of concurrent real-time tasks running on a single processor, using parametric timed automata. Our framework is generic and modular, so as to be easily adapted to different schedulers and more complex task models. We first perform a parametric schedulability analysis of the components using the inverse method. We show that the method unfortunately does not provide satisfactory results when the task periods are considered as parameters. After identifying and explaining the problem, we present a solution adapting the model by making use of the worst-case scenario in schedulability analysis. We show that the analysis with the inverse method always converges on the modified model when the system load is strictly less than 100%. Finally, we show how to use our parametric analysis for the generation of timed interfaces in compositional system design.

Keywords: Real-Time Scheduling, Parametric Schedulability Analysis, Parametric Timed Automata.

1 Introduction

Designing and analysing distributed real-time systems is a very challenging task. The main source of complexity arises from the large number of parameters to consider: tasks priorities, computation times and deadlines, synchronisation, precedence and communication constraints, etc. Finding the optimal values for the parameters is not easy and often a small change in one parameter may completely change the behaviour of the system and even compromise its correctness. For these reasons, designers are looking for analysis methodologies that allow incremental design and exploration of the parameter space.

We consider here real-time systems consisting of a set of real-time tasks executed concurrently on a single processor platform. Each task can be time-triggered or event-triggered: in the first case, it is activated periodically, and each time it executes a portion of code called *job* or *instance*, after which it self-suspends, waiting for their next periodic activation. In the second case, instances are activated by internal or external events. Each task is characterised by a *relative deadline*, that is the maximum amount of time that must elapse from the activation of one instance to its completion.

A *scheduler* is needed to decide which task to execute at each instant. The scheduler can be *on-line* if the decision is taken while the system is running depending on the current state; or *off-line* if the schedule is pre-computed before the system runs. Fixed Priority Preemptive Scheduling (FPPS) has been standardised in POSIX [1] and is currently available in all commercial and open-source Real-Time Operating Systems.

*The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 246556.
†This work is partially supported by STIC Asie project CATS ("Compositional Analysis of Timed Systems").

É. André and G. Frehse (Eds.): 1st International Workshop on Synthesis of Continuous Parameters (SynCoP'14).
EPTCS 145, 2014, pp. 49–64, doi:10.4204/EPTCS.145.6

One important requirement of real-time systems is to ensure that the system is *schedulable*, i.e. that all tasks will always complete before their deadlines when scheduled by the selected algorithm. Testing the system under different input and state conditions does not guarantee the system *schedulability* (i.e. that the system is schedulable), because the number of possibilities to test is too large to guarantee complete coverage of all possible cases. A better approach is to build an abstract model of the system, and perform analysis on the model.

A large body of research literature has addressed the problem of schedulability analysis of real-time tasks, both using formal methods (e.g. [2, 15, 16]) and mathematical equations (e.g. [12, 25]). In the literature, a task is typically modelled by several parameters, typically (i) a *worst-case computation time* (i.e. an upper bound of the execution time of every instance of the task under every possible condition), (ii) a *deadline*, and (iii) an *activation pattern* (e.g. periodic, sporadic, arrival curve). A periodic task is activated every period; a sporadic task can be activated at any time, but the distance between two activations is lower bounded by a constant *minimum interarrival time*; finally, an arrival curve [28] is a function $\alpha(t)$ that defines an upper bound on the maximum number of activations in any interval of length t.

Real-Time Components and Timed Interfaces

For complex distributed real-time systems, a component-based methodology may help reduce the complexity of the design and analysis phases. This paper is a first step toward the definition of a *timed interface* for a real-time component. Therefore, we now describe our notion of real-time component and timed interface.

We define a *distributed real-time system* as a set of *real-time components*. Each component runs on a dedicated single processor node, and all components are connected to each other by a local network. A component consists of a *provided interface*, a *required interface*, and an *implementation* (see e.g. [17]).

The *provided interface* is a set of methods that a component makes available to other components of the system. Each method is characterised by: (i) the method signature, which is the name of the method and the list of parameters, and (ii) a worst-case activation pattern, which describes the maximum number of invocations the method is able to handle in any interval of time. In this paper, we will describe the worst-case activation pattern by an *arrival curve* [28]. The semantic of invocation of a method can be synchronous (the caller waits for the method to be completed) or asynchronous (the caller continues to execute without waiting for the completion of the operation).

The *required interface* is a set of methods that the component requires for carrying out its services. Each method is characterised by its signature and a worst-case invocation pattern.

The *implementation* of a component is the specification of how the component carries out its work. In our model, a component is implemented by a set of concurrent real-time tasks and by a scheduler. Tasks can be time-triggered, when periodically activated; or event-triggered, in which case they are activated by a call to a provided method of the component. In other words, an event-triggered task implements one method of the provided interface, and in turn it may invoke a method of the required interface.

A graphical representation of a component is shown in Figure 1. In this example, the component provides one single method in the provided interface (pictorially represented by the red rectangle), and does not specify any method in the required interface. The component is implemented by three tasks: tasks τ_1 and τ_3 are time triggered (the green clocks in the picture), whereas task τ_2 implements the method in the provided interface, and hence it is triggered by invocations from external clients.

In a *component-based* design methodology, components are independently designed and developed, and then *integrated* in the final system by connecting them together through their interfaces. It is clear

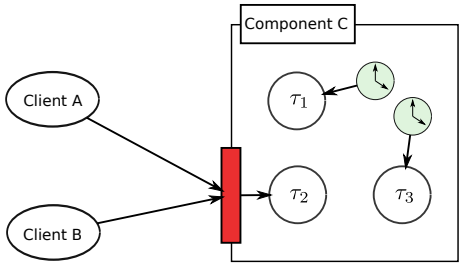


Figure 1: A component with three tasks and one method in the provided interface.

that the interface specification plays an important role in this methodology: for a real-time component, the interface should contain not only the functional specification (i.e. method signature, constraints on the parameters, etc.) but also the *timed behaviour* of the component. In particular, in this paper we enhance the specification of the interface by adding parameters on the activation pattern and on the response delay of a method.

Given a component $\mathscr{C}$, its *provided interface* is thus defined as:

- a set of method signatures $m_1, m_2, \ldots$;
- a *parametric arrival curve* $\alpha_i(t)$ for each method m_i, which represents the activation pattern that the corresponding implementing task will receive;
- a *worst-case response time* D_i parameter for each method m_i.

Similarly, the *required interface* of a component is defined as:

- a set of method signatures $m_1, m_2, \ldots$;
- a *parametric arrival curve* $\alpha_i(t)$ for each method m_i that represents the activation pattern generated by this component;
- for every synchronous method call, a maximum allowed delay R_i in receiving the response.

Finally, the component is characterised by a *set of constraints* on the parameters: for all valuations of the parameters satisfying the constraints, the component is guaranteed to be correct both from the functional point of view (i.e. the component produces correct values) and from the timing point of view (i.e. all tasks complete before their deadlines, and all provided functions return their values within the desired maximum response delay).

The road to realise such a component-based design methodology is long and many theoretical and practical problems need to be solved before the methodology can be used in practice. One important problem is how to compute the set of constraints that define the correct behaviour of a component. In the process of designing and analysing a component in isolation, it is necessary to use parametric arrival curves for describing the activation patterns for event-triggered tasks, and parametric deadlines for bounding their response times. Performing a parametric analysis aims at deriving a set of constraints for these parameters that make the component schedulable. During integration, the correctness of the system is checked by intersecting the constraints of the communicating components to see if there is some feasible assignment of parameters that makes all components schedulable.

Objectives

Our general research agenda, that goes beyond the scope of this paper, is to establish a component-based design methodology and analysis for real- time components. One of the important steps in the proposed

methodology is to be able to perform parametric analysis of a component with respect to its activation patterns.

In this paper, we focus on solving two specific problems:

1. how to build a formal parametric model of a component consisting of a set of real-time tasks, some of which can be periodic, others can be activated by generic arrival curves; and
2. how to perform a parametric analysis of the schedulability of the component, thus deriving a set of constraints that define the space of parameters that make the component schedulable.

For the sake of simplicity, in this paper we focus only on the *provided interface* of a component; that is, we investigate on the parametric analysis of a component with respect to the patterns of activations. The analysis of the required interface is the subject of future work.

Contributions

Our contribution is threefold. First, we propose a formal model of a real-time component based on parametric timed automata [4], a popular formalism for modelling real-time systems. Unlike many similar models proposed in the literature, our modelling framework is completely modular: a system is obtained by combining simpler automata, each one implementing one aspect of the component. In particular, we separate the specification of the task behaviour from the activation pattern (periodic, sporadic or generic arrival curve), and from the scheduler. In this way we can easily and seamlessly change the scheduler and the activation pattern of a task without changing the rest of the component specification, which is very important during the parametric analysis of a component. For the analysis, we use the inverse method [8] and the IMITATOR tool [7].

As a second contribution, we show that, when the activation patterns are parametric, the inverse method does not provide satisfactory results, in the sense that it may output a constraint reduced to a single point. We describe the problem and provide a solution that is valid for periodic (with no offset), sporadic and generic activation patterns that can be described by arrival curves.

Finally, as third contribution, we describe how this model can be used as a basis for synthesising the timed interface of a real-time component.

Organisation of the Paper

The rest of this paper is organised as follows. Section 2 reviews related work. Section 3 recalls the necessary preliminaries, viz. real-time systems, parametric timed automata and the inverse method. Section 4 presents our model of a real-time component using parametric timed automata. Section 5 introduces our parametric analysis, allowing to deal with parametric task activations. Section 6 introduces preliminary work allowing to perform a component-based parametric analysis. Section 7 concludes and present further directions of research.

2 Related Work

Parametric analysis of real-time systems using mathematical equations has already been addressed in the past. Bini et al. [10] proposed a method for parametric analysis of real-time periodic tasks where parameters can be either worst-case computation times or task periods. However, with Bini et al.'s approach, changing the task model requires the development of a new methodology.

A more general approach to scheduling analysis is to use formal methods for modelling a real-time system. A formal framework for scheduling problems using timed automata with stopwatches has been proposed in [2]. Fersman et al. [15] proposed a *Task Automaton*. Similar approaches have been proposed using time(d) Petri nets [11, 22].

It is possible to perform an exploration of the parameter space using timed automata, as in [20]. However, their approach is not fully parametric: the analysis is repeated for all combination of the discrete values of the parameters. Hence, their method does not scale well as the number of parameters and the number of discrete values increases. Furthermore, that approach does not consider non-integer points, and cannot be used to quantify the system robustness.

Full parametric analysis can be performed using specific formalisms. For example, formalisms such as parametric timed automata (PTA) [4] and parametric time Petri nets [29], have been used to model parametric schedulability problem (see, e.g. [14, 27]). In particular, thanks to generality of these modelling languages, it is possible to model a larger class of constraints, and perform full parametric analysis on many different variables, for example task offsets.

The inverse method IM [8] can be used for exploring the space of parameters of a parametric timed automaton (and, more generally, of a parametric *stopwatch* automaton) in the proximity of a valuation point. In this paper we use this method for performing parametric analysis of real-time systems where task activation patterns are modelled with parametric *arrival curves*.

We have used a similar approach in [27], where a distributed real-time system has been modelled using parametric stopwatch automata. However, in [27] the methodology is limited to only use the tasks' computation times as parameters. Here, we investigate a situation where arrival curves are considered as parameters too. Furthermore, our final goal is to be able to perform interface-based parametric analysis.

Our generic modular approach can be seen as a contract-based methodology where "provided" and "required" interfaces are instances of (assumption, guarantee) pairs in the contract terminology. An interface-based approach to the design and analysis of real-time systems using assume/guarantees has already been proposed in the literature [19, 26], but their approach is not parametric. Compositional verification of timed systems, using assume guarantee reasoning, has also been considered in [23] for event-recording automata, a subclass of timed automata; again, this approach is non-parametric.

3 Preliminaries

3.1 Real-Time Tasks

A real-time task τ_i is a sequence of instances (or jobs) $J_{i,k}$, with $k = 0, 1, \ldots$. Each instance $J_{i,k} = (a_{i,k}, c_{i,k}, d_{i,k})$ is characterised by an arrival time $a_{i,k}$, a computation time $c_{i,k}$, and an absolute deadline $d_{i,k}$. The system is schedulable if the scheduling algorithm orders the execution times of the jobs such that each job executes $c_{i,k}$ units of execution in interval $[a_{i,k}, d_{i,k}]$. Additionally, an instance can only start executing after the previous instances from the same task have completed: if we denote by $f_{i,k-1}$ the finishing time of the $(k-1)$th instance, then each job can only execute in interval $[\max(f_{i,k-1}, a_{i,k}), d_{i,k}]$.

Task τ_i is then characterised by three parameters:

- the Worst-Case Execution Time C_i, which is an upper bound on the execution time of any instance of the task (i.e. $\forall k > 0 : c_{i,k} \leq C_i$);
- the *relative deadline* D_i; the absolute deadline of every instance can be computed as $d_{i,k} = a_{i,k} + D_i$;
- the arrival pattern.

For the arrival pattern, we consider three kinds of schemes:

- **Periodic**: this arrival pattern is characterised by a *period* T_i, and the arrival time of every instance is computed as:

$$a_{i,0} = 0$$
$$\forall k > 0 : a_{i,k} = a_{i,k-1} + T_i$$

- **Sporadic**: this arrival pattern is characterised by a *minimum interarrival time* that we denote again by T_i, and the arrival times of every instance must respect the following constraints:

$$a_{i,0} = 0$$
$$\forall k > 0 : a_{i,k} \geq a_{i,k-1} + T_i$$

- **Arrival curve** [28]: in this case the pattern of arrival must respect a certain function called *arrival curve* $\alpha_i(t) : \mathbb{R} \to \mathbb{N}$. The arrival curve constrains the number n of arrivals in any interval of a given length Δ:

$$\forall k \geq 0, \forall n > 0 : n \leq \alpha_i(a_{i,k+n-1} - a_{i,k})$$

  In other words, the number of arrival events in any interval must not exceed the value of the arrival curve for that interval[1]. Arrival curves are monotonically non-decreasing, and convex, i.e. $\forall t, \delta : \alpha_i(t+\delta) \leq \alpha_i(t) + \alpha_i(\delta)$. The value of the an arrival curve at time 0 is also called *burstiness* and represents the amount of simultaneous arrival events that can be sent to a task. Arrival curves are a generalisation of the sporadic arrival model. In fact, a sporadic task can be represented by an arrival curve with burstiness $\alpha_i(0) = 1$ and a periodic behaviour. However, an arrival curve can have any convex shape.

  The sum of two arrival curves is still an arrival curve. Also, we can define a partial order relationship between arrival curves using the natural ordering between values of the function: $\alpha_i(\cdot) \preceq \alpha_j(\cdot)$ iff $\forall t\ \alpha_i(t) \preceq \alpha(t)$.

  In this paper we deal with parametric arrival curves. In particular, we will use periodic arrival curves of the form:

$$\alpha_{N^u,P}(t) = N^u + \left\lfloor \frac{t}{P} \right\rfloor \tag{1}$$

  where N^u is a discrete parameter that denotes the initial burstiness, and P is a continuous parameter that denotes the period. Using the partial order relationship, a generic arrival curve can always be upper bounded by a periodic arrival curve of the form (1).

3.2 Parametric Stopwatch Automata

We introduce here an extension of parametric timed automata that will be used in Section 4 to model real-time systems. Timed automata are finite-state automata augmented with clocks, i.e. real-valued variables increasing uniformly, that are compared within guards and invariants with timing delays [3]. Parametric timed automata (PTA) [4] extend timed automata with parameters, i.e. unknown constants, that can be used in guards and invariants. We will use here an extension of PTA with *stopwatches* [2], where clocks can be stopped in some control states of the automaton.

Given a set X of clocks and a set U of parameters, a constraint C over X and U is a conjunction of linear inequalities on X and U[2]. Given a parameter valuation (or point) π, we write $\pi \models C$ when

[1] Unlike in [28], for simplicity in this paper we only consider upper bound arrival curves.

[2] Note that this is a more general form than the strict original definition of PTA [4]; since most problems for PTA are undecidable anyway, this has no practical incidence, and increases the expressiveness of the formalism.

the constraint where all parameters within C have been replaced by their value as in π is satisfied by a non-empty set of clock valuations.

Definition 1. *A parametric timed automaton with stopwatches (PSA) $\mathscr{A}$ is $(\Sigma, Q, q_0, X, U, K, I, slope, \rightarrow)$ with Σ a finite set of actions, Q a finite set of locations, $q_0 \in Q$ the initial location, X a set of h clocks, U a set of parameters, K a constraint over U, I the invariant assigning to every $q \in Q$ a constraint over X and U, $slope : Q \rightarrow \{0,1\}^h$ assigns a constant slope to every location, and $\rightarrow$ a step relation consisting of elements (q, g, a, ρ, q'), where $q, q' \in Q$, $a \in \Sigma$, $\rho \subseteq X$ is the set of clocks to be reset, and the guard g is a constraint over X and U.*

The *slope* function is the extension of parametric timed automata to *stopwatch* timed automata, since it allows one to stop the time elapsing of some clock variables in some locations. This expressive power is used in the context of schedulability to model the preemption mechanism.

It is well-known that the parallel composition (using a synchronisation on actions) of several PSA is itself a PSA. Hence, it is common to model a complex system by composing several system components modelled themselves using PSA.

The semantics of a PSA $\mathscr{A}$ is defined in terms of states, i.e. pairs (q, C) where $q \in Q$ and C is a constraint over X and U. Given a point π, we say that a state (q, C) is π-compatible if $\pi \models C$. Runs are alternating sequences of states and actions, and traces are time-abstract runs, i.e. alternating sequences of *locations* and actions. The trace set of $\mathscr{A}$ corresponds to the traces associated with all the runs of $\mathscr{A}$. Given $\mathscr{A}$ and π, we denote by $\mathscr{A}[\pi]$ the (non-parametric) timed stopwatch automaton where each occurrence of a parameter has been replaced by its constant value as in π. Details can be found in, e.g. [8].

3.3 The Inverse Method

The inverse method for PSA [8] exploits the knowledge of a reference point of timing values for which the good behaviour of the system is known. The method synthesises automatically a dense space of points around the reference point, for which the discrete behaviour of the system, that is the set of all the admissible sequences of interleaving events, is guaranteed to be the same.

The inverse method IM proceeds by exploring iteratively longer runs from the initial state. When a π-incompatible state is met (that is a state (q, C) such that $\pi \not\models C$), a π-incompatible inequality J is selected within the projection of C onto U. This inequality is then negated, and the analysis restarts with a model further constrained by $\neg J$. When a fixpoint is reached, that is when no π-incompatible state is found and all states have their successors within the set of reachable states, the intersection of all the constraints onto the parameters is returned.

IM proceeds by iterative state space exploration, and its result comes under the form of a fully parametric constraint. By repeatedly applying the method, we are able to decompose the parameter space into a covering set of "tiles", which ensure a uniform behaviour of the system: it is sufficient to test only one point of the tile in order to know whether or not the system behaves correctly on the whole tile. This is known as the *behavioural cartography* [5]. Both the inverse method and the behavioural cartography are semi-algorithms; that is, they are not guaranteed to terminate but, if they do, their result is correct.

4 A Modular Framework for Modelling Real-Time Systems

In this section we refer to a real-time system as a set of real-time tasks scheduled by a FPPS on a single processor. Of course, the discussion is valid also when considering a single component of a large real-time distributed system.

Our model of a real-time system consists of three kinds of PSA components: the task automata, the task activation automata and the scheduler automaton. We refer to the composition of these PSA components through synchronisation labels as the system automaton.

Each task is modelled using a task automaton. Such a task automaton is shown in Figure 2a. Each task automaton contains two (local) continuous clock variables c and d. Clock c counts the execution of the task and clock d counts the time passed since last job arrival. Since we consider generic activation patterns (periodic, sporadic or arrival curves), a new instance may be activated while the previous ones have not yet completed. Hence, there could be several active jobs from the same task at the same time. A discrete[3] variable N is used to count the number of simultaneous active instances for the task.

Initially, a task is in location Idle. The synchronisation label arrival_event notifies that a new instance from this task is activated and triggers a transition to a committed location ActEvent. A committed location is a location where time elapsing is not allowed, represented graphically using a double circle location. The label arrival is used between a task and the scheduler. The task will then go to location Waiting and wait there for the scheduler's decision whether to occupy the CPU. If a task has the highest priority among the active tasks in the system, the scheduler will send dispatch to trigger the transition from Waiting to Running. While a task is in Running, the scheduler could revoke the CPU for a higher priority task through synchronisation label preemption.

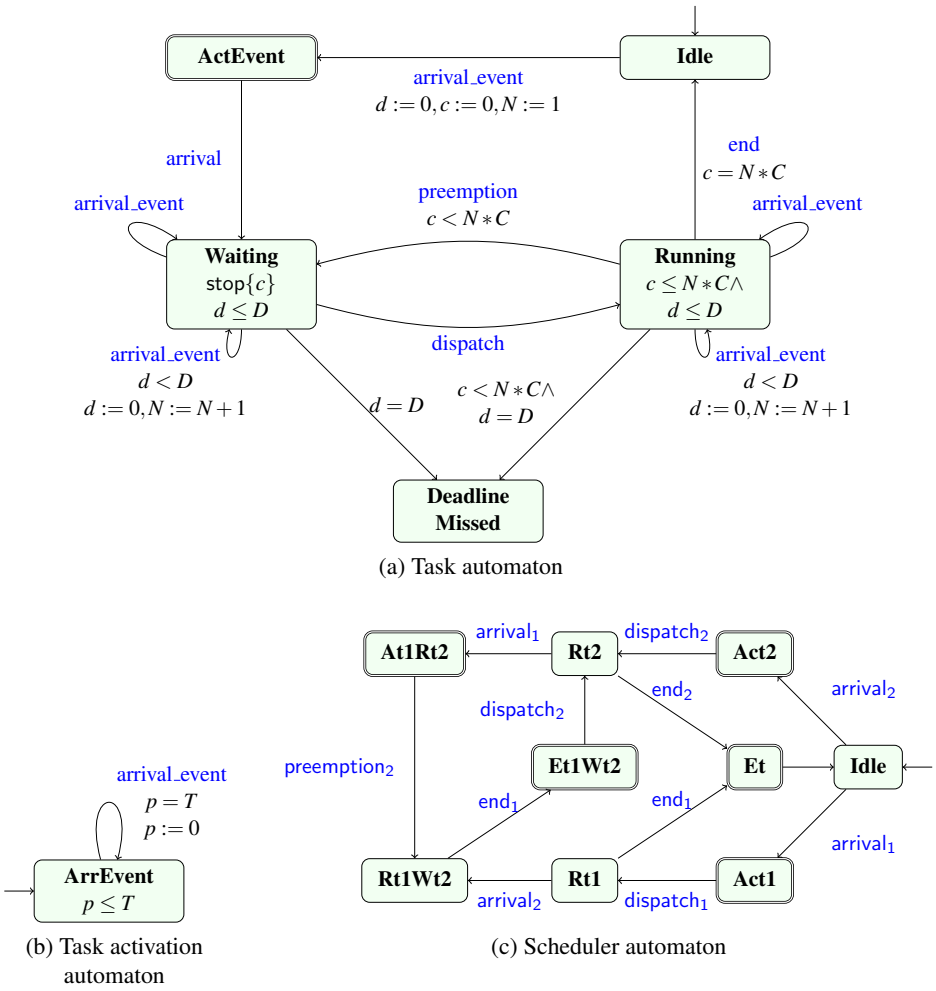
Clock d always progresses and the execution time clock variable c is stopped if a task is waiting. When a task is waiting for the CPU or running on the CPU, to react to new activations, it will non-deterministically choose to increase the counter N of active instances by 1. When a job misses its deadline ($d = D$) before completing its execution, it will go to DeadlineMissed. When a task finishes its execution ($c = N * C$), it will go back to initial location Idle.

There could be many different activation patterns for a task, such as periodic, sporadic or according to arrival curves. We only require that the activation automaton synchronises with the task automaton on label arrival_event. As a demonstration, Figure 2b shows the activation model for a periodic task. Every period T, the automaton sends the signal arrival_event to inform the arrival of a new job.

In this paper, we assume tasks are scheduled according to a fixed-priority fully-preemptive scheduler (FPPS). The scheduler automaton synchronises with the tasks and decides which task will occupy the CPU at each time. The structure of the automaton is completely fixed given a number of tasks.

Figure 2c shows a scheduler for two tasks, where task 1 has higher priority. The scheduler automaton can be expanded in a similar form to deal with a task set with more tasks. In the scheduler automaton, the labels arrival, dispatch, preemption and end are the same as in task automaton; we append a label with index i, e.g. end_i, to denote that this label synchronises with task i. The convention we use for naming the location encodes the status of the tasks: Rtx means the task τ_x is running; Atx means task τ_x is just activated; Wtx means task τ_x is waiting; Et is saying the task just finished its execution.

[3] Discrete variables are not part of the original PTA/PSA formalisms, but can seen as syntax sugar to increase the number of discrete states (locations). Such discrete variables are supported by most tools for (parametric) timed automata.



(a) Task automaton

(b) Task activation automaton

(c) Scheduler automaton

Figure 2: The modelling framework for a real-time system

5 Parametric Schedulability Analysis of Real-Time Components

5.1 Convergence Problem

We first show that the application of the inverse method IM to a system with parametric task activations does not yield satisfactory results. Consider a task set with two periodic tasks $\tau_1 = (31, T_1, T_1), \tau_2 = (49, T_2, T_2)$ with implicit deadlines (i.e. deadlines always equal to periods). If we use IM with initial values $T_1 = 60$ and $T_2 = 120$, respectively, the final constraints obtained will be $T_1 = 60$ and $T_2 = 120$. That is, the result produced by IM is a single point, the initial valuation.

Such result is caused by an important property of the schedule. The inverse method synthesises a set of constraints that delimit the values for the parameters that result in the same exact traces as the initial valuation. The schedule generated by a set of periodic real-time tasks is itself periodic with period H (also called *hyperperiod*). In particular, the sequence of scheduling events repeats itself every H, and different H will result in different traces of task execution. The hyperperiod can be computed as the least common multiple of all task periods: $H = \mathsf{lcm}(T_1, \ldots, T_n)$. When periods are parametric, and since

function lcm() is highly non linear, a small variation on one period can cause very large variations in the hyperperiod. For example, consider the two previous tasks with initial valuation of the periods $T_1 = 60$ and $T_2 = 120$, respectively. Their hyperperiod is 120. When we increase the second period to 121, the hyperperiod becomes 7260. Clearly, in this second case the traces are much longer and contain many more events. This explains why IM only converges to the initial valuation.

Of course, things become even more complex when considering generic arrival patterns. The next section solves this convergence problem by exploiting a well-known result from classical scheduling theory.

5.2 An Improved Model of the System

As discussed in Section 5.1, it is infeasible to apply IM directly to a system model with parametric arrival patterns. We will try to avoid this situation by adapting the system automaton (Figure 2) by exploiting the concept of critical instant.

For a set of periodic or sporadic tasks scheduled by FPPS on a single processor it is possible to define a *critical scenario*, which is the situation that arises when all tasks are simultaneously activated (*critical instant*) and every task τ_i generates subsequent jobs as soon as it is allowed. According to the seminal work by Liu and Layland [24], the worst-case response time of a task can be found in the *busy period* (i.e. interval in which the processor is continously busy) that starts at the critical instant.

This means that, if we want to check the schedulability of a set of periodic or sporadic real-time tasks, it is sufficient to activate all tasks at time zero and check that no deadline is missed in the first busy period starting at time 0. Therefore, as soon as the processor becomes idle we can stop our search.

In the system automaton in Section 4, each trace corresponds to a possible schedule of the task set. However, we now know that to check the schedulability of a task set, it is sufficient to analyse traces starting from the critical instant till the first idle time in CPU. So, we adapt the system automaton as follows:

- The task activation automaton is required to release its first job at time 0 and it will emit the subsequent jobs as fast as the task is allowed;
- In the scheduler automaton, after all tasks complete their execution, instead of going back to Idle, it will transit from Et to a new location Stop, where this is no outgoing edge.

The first point is used to simulate the worst-case behaviour of tasks at the critical instant. Rather than going to Idle and waiting for new task releases, the scheduler automaton (also the system automaton) simply stops. We call this adapted scheduling model as the *idle-time scheduler automaton*.

The idle-time scheduler automaton actually simulates the longest *busy period*, which starts from the critical instant and ends at the first idle time of the processor. The length of this busy period depends both on the execution time and on activation periods of the tasks. However, the dependence from the periods is not so strong as with the hyperperiod. Let us consider again the previous set of two periodic tasks $\tau_1 = (C_1 = 31, T_1 = D_1 = 60)$ $\tau_2 = (49, 120, 120)$. The schedule for the first busy period is shown in Figure 3. Task τ_1 executes twice before the first instance of τ_2 can complete.

The length of the busy period in this case is $2C_1 + C_2 = 111$. By doing some simple calculation, it is easy to see that changing T_1 to any value in $[56, 79]$ does not change the sequence of events in the busy period: in facts, for any value of T_1 in that interval, τ_1 will still execute two times before the first instance of τ_2 completes. Also, changing T_2 to any value $T_2 \geq 112$ does not change the busy period.

Hence, we can apply IM on the new model and avoid the convergence problem as in Section 5.1. Let us assume $T_1 \in [40, 120]$, $T_2 \in [80, 200]$ and let us apply the behavioural cartography to obtain the

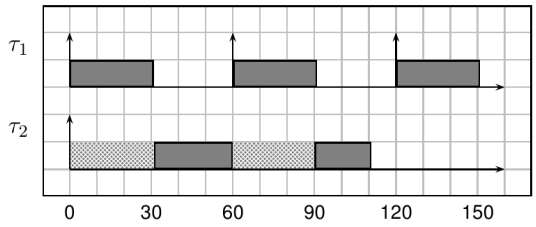


Figure 3: Schedule of the first busy period of the example task set

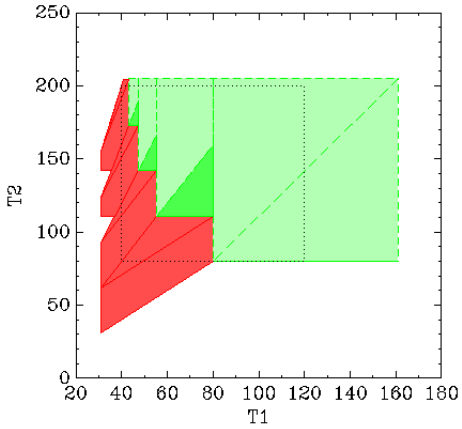


Figure 4: Constraints on T_1 and T_2 obtained by the behavioural cartography

constraint space of T_1, T_2 that keeps the task set schedulable. The result is given in Figure 4 in a graphical form. The red part (on the left) is the constraint space on T_1 and T_2 in which the system misses τ_2's deadlines, whereas the green part (on the right) is where no deadline is missed.

When applying the behavioural cartography to the idle-scheduler automaton, there may exist a combination of parameters that cause the system to go into overload, i.e. there will be no idle time in the schedule. For example, in case of periodic tasks, this happens when the total system utilisation is such that $\sum_{i=1}^{n} \frac{C_i}{T_i} > 1$. In the previous example, $(T_1 = 40, T_2 = 80)$ is one such point. Of course, this will surely cause a deadline miss, because it means that the total amount of work to be performed (utilisation) exceeds the amount of available processor time.

To solve this case, we put an upper bound on the maximal depth of the traces computed by IM. This bound is always computable in the case of periodic real-time tasks, and corresponds to computing an upper bound to the time where a deadline miss will happen. A method for computing such a bound can be built by using the concept of *demand bound function* [9].

5.3 Applicability of the Idle-Time Scheduler

It is possible to prove that the concepts of critical instant and maximal busy periods are valid also when considering tasks activated by generic arrival curves [28]. In particular, the critical scenario corresponds to the time instant in which all tasks are activated with their initial burstiness (critical instant), and their successive instances arrive as soon as possible without violating their arrival curves. Then, the worst-case response time can be found in the busy period starting at the critical instant and corresponding to

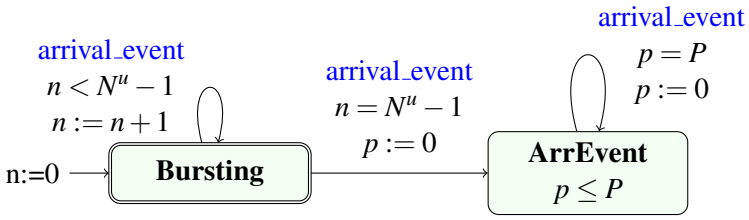


Figure 5: Arrival curve automaton

the critical scenario. Therefore, we will use the same technique also for generic arrival curves.

In Figure 5 we show the simple PSA model for a parametric periodic arrival curve described by Equation 1. Initially, the arrival curve automaton is in a committed location Bursting with $n = 0$, where n is a discrete variable counting the number of initial client requests. The automaton emits N^u activations for a task (τ_2 in our case) within 0 time elapse and then moves to location ArrEvent where is starts behaving as a periodic activation automaton as in Figure 2b, and produces activations events every P.

For other different task models there is no critical instant. For example, when considering periodic tasks with initial offset different from zero, there is no worst-case scenario in the schedule. Instead, it is necessary to analyse all busy periods in the interval $[0, 2H + \Phi_{\mathsf{max}}]$, where Φ_{max} is the largest initial offset [21].

Given a task set $\mathscr{T}$ of periodic real-time task with offsets, we can build a task set $\mathscr{T}'$ that contains the same tasks with the same parameters except that their initial offsets are all set to zero. In this case, it is possible to prove that, if $\mathscr{T}'$ is schedulable, then also $\mathscr{T}$ is schedulable. However, the converse does not hold. Therefore, it is possible to perform a parametric analysis of $\mathscr{T}'$ using our idle-time scheduler, and the set of values of the parameters produced by the analysis is a subset of the set of valid parameters for the original system $\mathscr{T}$. A more precise analysis requires point-by- point exploration of the parameter space.

Finally, in this paper we assume that task are independent from each other, and do not self-suspend waiting for other events different from the activation event. An example of self-suspending task is a task that performs a remote procedure call, and self-suspends waiting for the response. Again, in this case there is not a single critical scenario for the task set, therefore our simplified model cannot be used.

6 Towards Timed Interfaces

In this section we show how it is possible to define a *timed interface* of a real-time component using parametric analysis.

Consider the system of Figure 1: it consists of 3 tasks τ_1, τ_2 and τ_3 running on a single processor with FPPS. A task with smaller index has higher priority. τ_1 and τ_3 are periodic tasks with $\tau_1 = (C_1 = 2, D_1 = 8, T_1 = 8)$ and $\tau_3 = (C_3 = 20, D_3 = 50, T_3 = 50)$. Task τ_2 has $C_2 = 5$ and implements the method provided in the interface. We assume that this component is linked to a local networks, and task τ_2 receives the requests from clients running on other nodes of the network. We would like to know how many clients can ask requests to the system, with which frequency, and the maximum delay that is going to pass from the request to the response. Therefore, we need to study the possible activation patterns of task τ_2 and its worst-case response time. For modelling the activation patterns, we use a parametric arrival curse as described by Equation 1. For example, $N^u = 2$ and $P = 100$ means that we can connect at most 2 independent clients, and that between any two consecutive requests after the first two there must be at

$N^u = 1$	when $(20 \leq P \leq 26) \vee (27 \leq P \leq 34) \vee (35 \leq P \leq 50) \rightarrow D_2^{min} = 10$
$N^u = 2$	when $(24 \leq P \leq 26) \vee (27 \leq P \leq 34) \vee (35 \leq P \leq 50) \rightarrow D_2^{min} = 14$
$N^u = 3$	when $(P = 47) \vee (48 \leq P \leq 50) \rightarrow D_2^{min} = 21$

Table 1: The final interface

most 100 units of time.

Both N^u and P are parameters we are going to synthesise with our parametric analysis. Another parameter is the delay (deadline) D_2 of τ_2. We are interested in the parameter space that guarantees all the tasks are schedulable.

First, we construct the activation automaton for $\alpha(t)$ as in Figure 5. Following the method described in Section 4, and using the idle-time scheduler automaton, we then compose the final automaton.

Given that $C_2 = 5$, it is easy to see that the burst (N^u) of the arrival curve automaton cannot be larger than 3, otherwise τ_3 will be doomed to miss its deadline, because $D_3 < C_3 + 4C_2 + 5C_1$. Additionally, we assume P and D_2 lie in following intervals:

$$P \in [20,50],\ D_2 \in [10,50]$$

N^u is a discrete parameter that must be treated separately from the other parameter. Our strategy is to instantiate N^u with 1, 2 and 3 individually and apply IM to each case in order to synthesise constraints over P and D_2 that keep the system schedulable. The resulted parameter spaces for the three cases are visualised in Figure 6.

We can use these values to build a *timed interface specification* for the component.

- the number of distinct independent clients that can be connected to the service must respect the constraint $1 \leq N^u \leq 3$;
- Depending on the number of clients, the relationship between minimum request period P and worst case response time D_2 is specified in Table 1.

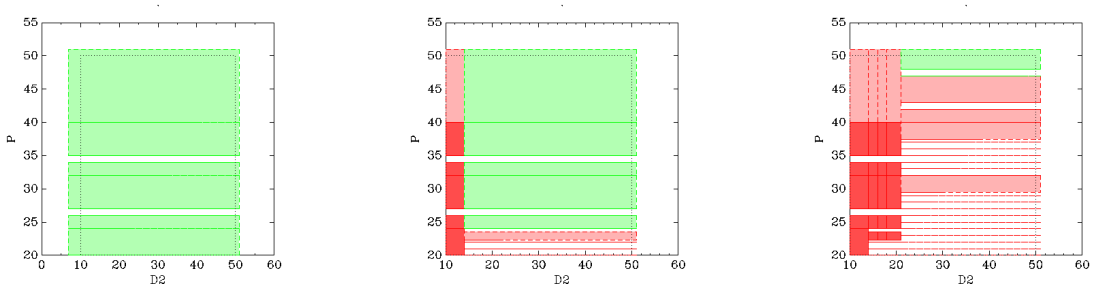
Reducing the number of regions

As it is possible to see in Figure 6 and in Table 1, the parameter space returned by IM consists of a set of disjoint tiles. Each tile is a convex region and the resulting interface is the union of (maybe a large number of) these convex regions. Such an interface may not be easy to use due to the large number of disjoint regions.

In some cases, it is possible to perform a "merge" operation between the tiles, as explained in [6], in order to reduce the number of convex regions composing the final interface. Two convex regions are mergeable if their convex hull equals to their union. Given tiles returned from IM, we repeatedly replace mergeable tiles, satisfying this condition, with their union till there are no mergeable tiles. If we restrict ourselves to integers solutions, we may further merge adjacent tiles. For example, the constraint $(20 \leq P \leq 26)$ can be merged with $(27 \leq P \leq 34)$, thus obtaining $(20 \leq P \leq 34)$. We are currently investigating efficient methods for automatically merging tiles resulting from IM cartography.

7 Conclusion and Future Work

In this paper we have presented a PTA model of a real-time systems scheduled by FPPS. We have shown how to perform a parametric analysis using IM with a specific model of the scheduler that stops at the



(a) $N^u = 1$ (b) $N^u = 2$ (c) $N^u = 3$

Figure 6: Parameter space (green) for N^u, P and D_2

first idle time. Finally, we have shown how to use parametric analysis for the design and the specification of the interface of a real-time component.

We wish to continue along this line of research and investigate about the possibility to systematically use parametric analysis for interface specification. We are currently investigating efficient methods for reducing the complexity of the set of regions produced by IM, either by using more sophisticated merging techniques, or by using conservative approximations. Also, we plan to extend the analysis to more complex task models like self-suspending tasks and task dependencies.

More specifically on the parameter synthesis techniques, it would be interesting to reuse some technique for integer parameter synthesis recently proposed in [18]; on the negative side, only integer points are synthesised, thus preventing the interpretation of the result for robustness analysis (in the sense of infinitesimal variations of the parameters); on the positive side, these techniques are efficient and guaranteed to terminate. Also, combining the inverse method with IC3 [13] is an interesting future direction of research.

A more general (and challenging) objective is also to be able to derive (possibly non-linear) constraints relating the discrete and continuous parameters, e.g. relating the number of clients ("N^u" in Section 6) with the timing parameters ("P" and "D_2" in Section 6).

Acknowledgment

We would like to thank anonymous reviewers for their useful comments.

References

[1] (1999): *IEEE Standard for Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API)- Amendment D: Additional Real time Extensions [C Language]*, doi:10.1109/IEEESTD.1999.91515.

[2] Yasmina Adbeddaïm & Oded Maler (2002): *Preemptive Job-Shop Scheduling using Stopwatch Automata*. In: *TACAS*, *Lecture Notes in Computer Science* 2280, Springer-Verlag, pp. 113–126, doi:10.1007/3-540-46002-0_9.

[3] Rajeev Alur & David L. Dill (1994): *A theory of timed automata*. Theoretical Computer Science 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[4] Rajeev Alur, Thomas A. Henzinger & Moshe Y. Vardi (1993): *Parametric real-time reasoning*. In: *STOC*, ACM, pp. 592–601, doi:10.1145/167088.167242.

[5] Étienne André & Laurent Fribourg (2010): *Behavioral Cartography of Timed Automata*. In: *RP*, Lecture Notes in Computer Science 6227, Springer, pp. 76–90, doi:10.1007/978-3-642-15349-5_5.

[6] Étienne André, Laurent Fribourg & Romain Soulat (2013): *Merge and Conquer: State Merging in Parametric Timed Automata*. In: *ATVA*, Lecture Notes in Computer Science 8172, Springer, pp. 381–396, doi:10.1007/978-3-319-02444-8_27.

[7] Étienne André, Laurent Fribourg, Ulrich Kühne & Romain Soulat (2012): *IMITATOR 2.5: A Tool for Analyzing Robustness in Scheduling Problems*. In: *FM*, Lecture Notes in Computer Science 7436, Springer, p. 33–36, doi:10.1007/978-3-642-32759-9_6.

[8] Étienne André & Romain Soulat (2013): *The Inverse Method*. ISTE Ltd and John Wiley & Sons Inc., doi:10.1002/9781118569351.

[9] Sanjoy K. Baruah, Louis E. Rosier & Rodney R. Howell (1990): *Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor*. Real-Time Systems 2(4), pp. 301–324, doi:10.1007/BF01995675.

[10] Enrico Bini, Marco Di Natale & Giorgio C. Buttazzo (2006): *Sensitivity Analysis for Fixed-Priority Real-Time Systems*. In: *ECRTS*, p. 13–22, doi:10.1007/s11241-006-9010-1.

[11] Giacomo Bucci, Andrea Fedeli, Luigi Sassoli & Enrico Vicario (2004): *Timed state space analysis of real-time preemptive systems*. IEEE Transactions on Software Engineering 30(2), pp. 97–111, doi:10.1109/TSE.2004.1265815.

[12] Giorgio C. Buttazzo (2004): *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS.

[13] Alessandro Cimatti, Alberto Griggio, Sergio Mover & Stefano Tonetta (2013): *Parameter synthesis with IC3*. In: *FMCAD*, IEEE, pp. 165–168, doi:10.1109/FMCAD.2013.6679406.

[14] Alessandro Cimatti, Luigi Palopoli & Yusi Ramadian (2008): *Symbolic Computation of Schedulability Regions Using Parametric Timed Automata*. In: *RTSS*, IEEE Computer Society, pp. 80–89, doi:10.1109/RTSS.2008.36.

[15] Elena Fersman, Leonid Mokrushin, Paul Pettersson & Wang Yi (2003): *Schedulability Analysis Using Two Clocks*. In: *TACAS*, pp. 224–239, doi:10.1007/3-540-36577-X_16.

[16] Elena Fersman, Paul Pettersson & Wang Yi (2002): *Timed Automata with Asynchronous Processes: Schedulability and Decidability*. In: *TACAS*, Springer-Verlag, pp. 67–82, doi:10.1007/3-540-46002-0_6.

[17] Gomaa Hassan (2012): *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*. Cambridge University Press, doi:10.1145/1988997.1989008.

[18] Aleksandra Jovanović, Didier Lime & Olivier H. Roux (2013): *Integer Parameter Synthesis for Timed Automata*. In: *TACAS*, Lecture Notes in Computer Science 7795, Springer, pp. 401–415, doi:10.1007/978-3-642-36742-7_28.

[19] Kai Lampka, Simon Perathoner & Lothar Thiele (2013): *Component-based system design: analytic real-time interfaces for state-based component implementations*. International Journal on Software Tools for Technology Transfer 15(3), p. 155–170, doi:10.1007/s10009-012-0257-7.

[20] Thi Thieu Hoa Le, Luigi Palopoli, Roberto Passerone & Yusi Ramadian (2013): *Timed-automata based schedulability analysis for distributed firm real-time systems: A case study*. International Journal on Software Tools for Technology Transfer 15(3), pp. 211–228, doi:10.1007/s10009-012-0245-y.

[21] Joseph Y.-T. Leung & Jennifer Whitehead (1982): *On the complexity of fixed-priority scheduling of periodic, real-time tasks*. Performance Evaluation 2(4), pp. 237–250, doi:10.1016/0166-5316(82)90024-4.

[22] Didier Lime & Olivier H. Roux (2009): *Formal verification of real-time systems with preemptive scheduling*. *Real-Time Systems* 41(2), pp. 118–151, doi:10.1007/s11241-008-9059-0.

[23] Shang-Wei Lin, Étienne André, Yang Liu, Jun Sun & Jin Song Dong (2014): *Learning Assumptions for Compositional Verification of Timed Systems*. *Transactions on Software Engineering*, doi:10.1109/TSE.2013.57. To appear.

[24] Chung Laung Liu & James W Layland (1973): *Scheduling algorithms for multiprogramming in a hard-real-time environment*. *Journal of the ACM* 20(1), pp. 46–61, doi:10.1145/321738.321743.

[25] Jane W. Liu (2000): *Real-time systems*. Prentice Hall PTR.

[26] Insik Shin & Insup Lee (2008): *Compositional real-time scheduling framework with periodic model*. *ACM Transactions on Embedded Computing Systems (TECS)* 7(3), p. 30, doi:10.1145/1347375.1347383.

[27] Youcheng Sun, Romain Soulat, Étienne Lipari, Giuseppe André & Laurent Fribourg (2013): *Parametric Schedulability Analysis of Fixed Priority Real-Time Distributed Systems*. In Cyrille Artho & Peter Ölveczky, editors: *Second International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS'13)*, *Communications in Computer and Information Science* 419, Springer. To appear.

[28] L. Thiele, S. Chakraborty & M. Naedele (2000): *Real-time calculus for scheduling hard real-time systems*. In: *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems, ISCAS 2000, Geneva.*, 4, IEEE, pp. 101–104, doi:10.1109/ISCAS.2000.858698.

[29] Louis-Marie Traonouez, Didier Lime & Olivier H. Roux (2009): *Parametric Model-Checking of Stopwatch Petri Nets*. *Journal of Universal Computer Science* 15(17), pp. 3273–3304, doi:10.3217/jucs-015-17-3273.