Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Modeling Task Systems Using Parameterized Partial Orders*

Fred Houben[2], **Georgeta Igna**[1], **Frits Vaandrager**[1]

[1] ICIS, MBSD group
Radboud University Nijmegen,
The Netherlands
e-mail: {g.igna,f.vaandrager}@cs.ru.nl
[2] ASML
The Netherlands
e-mail: fred.houben@asml.com

September 12, 2012

Abstract. Inspired by work on model-based design of printers, the notion of a parametrized partial order (PPO) was introduced recently. PPOs are a simple extension of partial orders, expressive enough to compactly represent large task graphs with finite repetitive behavior. We present a translation of the PPO subclass to timed automata and prove that the transition system induced by the Uppaal models is isomorphic to the configuration structure of the original PPO. Moreover, we introduce real-time task systems (RTTS), a general model for real-time embedded systems that we have used to describe the data paths of realistic printer designs. In an RTTS, tasks are represented as PPOs and the pace of a task instance may vary, depending on the resources that are allocated to it. We describe a translation of a subclass of RTTSs to Uppaal, and establish, for an even smaller subclass, bisimulation equivalence between the timed configuration semantics of an RTTS and the transition system induced by the corresponding Uppaal translation. Lastly, we report on a series of experiments which demonstrates that the resulting Uppaal models are more tractable than handcrafted models of the same systems used in earlier case studies.

1 Introduction

The complexity of today's real-time embedded systems and their development trajectories is increasing rapidly. At the same time, development teams are expected to produce high-quality and cost-effective products, while meeting stringent time-to-market constraints. A common challenge during development is the need to explore extremely large design spaces, involving multiple metrics of interest (timing, resource usage, energy usage, or cost). The number of design parameters (number and type of processing cores, sizes and organization of memories, interconnect, scheduling and arbitration policies) is typically very large. Moreover, the relation between parameter settings and design choices on the one hand and metrics of interest on the other hand is often difficult to determine. Given these observations, real-time embedded system design trajectories require a systematic approach, that should be automated as far as possible. To achieve high-quality results, design process and tooling need to be model-driven.

Many methods and tools for real-time embedded systems follow the Y-chart pattern [2,3]. This pattern is based on the observation that the development of these systems typically involves the co-development of a set of applications, a platform, and the mapping of the applications onto the platform. In the Y-chart pattern, specification of applications, platforms and mappings are separated. This allows independent evaluation of various alternatives of one of these system aspects while fixing the others. For example, various platform and mapping options are often investigated for a fixed (set of) application(s). Diagnostic information is used to, automatically or manually, improve application, platform, and/or mapping.

Applications are typically described in terms of task graphs representing partially ordered sets of tasks. In practice, we frequently see that certain tasks need to be executed repetitively, for a finite number of times, and that there exists a hierarchical relationship between tasks. For instance, a manufacturing order of a beer brewery consists of several pallets, containing several

* An extended abstract of this paper appeared as [1].The research of Igna and Vaandrager has been carried out as part of the OCTOPUS project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program. This research was also supported by European Community's Seventh Framework Programme under grant agreement no 214755 (QUASIMODO).

crates, each containing several bottles of beer. Another example concerns a wafer scanner manufacturing system from the semiconductor industry. Wafers are produced in batches (lots). A wafer scanner projects a mask on a wafer, using light. Eventually, the projected masks result in Integrated Circuits (ICs). On one wafer, multiple ICs and types of ICs are manufactured. Multiple types of ICs involve multiple masks, and multiple masks are placed on a reticle. As a final example, we mention a copier machine, which has to process a certain number of copies of a file, which in turn consists of a certain number of pages. Due to the nested, finite repetitive behavior, task graphs tend to become very large and no longer practical for specification and analysis of application behavior. Following [4,5], we argue that repetitive task structure of applications plays an important role in real-time embedded systems design, and needs to be addressed in methods for specifying and reasoning about such systems. Repetitive execution of tasks leads to finite repetitive patterns in schedules. In practice, execution of the first few instances and last few instances of a task differ slightly from the rest. This is a large difference with unlimited repetitive ('periodic') behavior, which has received much attention in the scheduling literature.

Within concurrency theory, several semantic models have been proposed that are based on partial ordering of events such as Mazurkiewicz [6] traces, pomsets (partially-ordered multisets) [7], and event structures [8], but these models do not incorporate an explicit notion of repetitive events. Partial orderings of events with repetition can be defined using Colored Petri Nets [9,10], but this is an extremely rich and expressive formalism, which may be considered too complicated for the task at hand.

The Octopus project has developed a Design-Space Exploration (DSE) toolset [11] that aims to leverage existing modeling, analysis, and DSE tools to support model-driven DSE for real-time embedded systems [12]. The Octopus toolset is centered on an intermediate representation, DSEIR (Design-Space Exploration Intermediate Representation), to capture design alternatives. DSEIR models can be exported to various analysis tools. This facilitates reuse of models across tools and provides model consistency between analyses. The use of an intermediate representation also supports domain-specific abstractions and reuse of tools across application domains. The current version of the Octopus DSE toolset integrates CPN Tools [9,10] for stochastic simulation of timed systems, SDF3 [13] for worst-case throughput calculation, and Uppaal [14] for model checking and schedule optimization. Inspired by work on model-based design of printers, the Octopus project has introduced the notion of parametrized partial orders [15]. PPOs are a simple extension of partial orders, but expressive enough to compactly represent large task graphs with repetitive behavior. In DSEIR, applications are represented as PPOs. This intermediate representation can be translated into the input formats of CPN Tools and Uppaal. which in turn can be translated into the input formats of CPN Tools and Uppaal. A translation of PPOs to CPN Tools has recently been described in [15]. In this paper, we define a restricted version of PPOs that is more amenable to model checking. Moreover, we give a translation into timed automata, the semantic model underlying Uppaal.

Uppaal [14] is a model checker for networks of timed automata [16]. It has been successfully used in many domains, e.g. for finding optimal solutions for scheduling problems [17,18], performance analysis of real-time distributed systems [19,20], protocol verification [21] and controller synthesis [22]. Within the Octopus project, we aim at harnessing the verification power of Uppaal for DSE of real-time embedded systems. We have applied Uppaal for DSE of industrial printer designs, in particular for computing and optimizing schedules, latencies, and controller strategies [23–25]. Although these case studies demonstrate that Uppaal is able to handle industrial sized designs, the tool is really pushed to its limits. Therefore, it is crucial to have a translation from PPOs to Uppaal that is maximally efficient. By unfolding a PPO into a task graph and introducing a separate automaton for each task in the unfolding, we obtain a general translation of PPOs to Uppaal. However, especially when we have many repetitive events (e.g. print a 300-page document) the translation becomes intractable. Based on the observation that in practice the PPOs often contain tasks that are not auto-concurrent and precedence relations between task instances obey certain monotonicity conditions, we define a subclass of PPOs that allows a more efficient translation.

In the literature, numerous modeling frameworks for real-time task systems have been proposed, see for instance [26–32]. In the design space exploration of printers, a small gain (5%) of performance can be decisive, and consequently detailed models of real-time task systems are needed in which, for instance, we can express that the completion time of a task slightly increases if another task is using the same communication bus simultaneously. To the best of our knowledge, such a refined performance analysis is not possible in existing analysis methods for task systems (see, for instance, the detailled comparison of difference performance analysis methods by Hendriks & Verhoef [19] and by Perathoner et al [20]). Therefore we introduce, in this paper, real-time task systems (RTTS), a general model for real-time embedded systems able to accurately describe the data paths of realistic printer designs. In an RTTS, tasks are represented as PPOs and the pace of a task instance may vary, depending on the resources that are allocated to it. This allows us to accurately model common scenarios in which, for instance, the completion time of a task depends on the varying amount of memory and communication bandwidth allocated to it over time.

This brings us to the main contributions of this paper:

1. A definition of a PPO subclass and its translation to Uppaal together with a correctness proof (the transition system induced by the Uppaal model is isomorphic to the configuration structure of the PPO).
2. A notion of a real-time task systems (RTTS), a general model for real-time embedded systems that we have used to describe the data paths of realistic printer designs. We describe a translation of a subclass of RTTSs to Uppaal, and establish, for an even smaller subclass, bisimulation equivalence between the timed configuration semantics of an RTTS and the transition system induced by the corresponding Uppaal translation.
3. A series of experiments which demonstrates that Uppaal models obtained through this translation are more tractable than handcrafted models built for a printing system described in [23].

The structure of this paper is as follows. Section 2 recalls some preliminary definitions regarding labeled transition systems, the underlying semantic notion used throughout the paper. Section 3 defines PPOs and their semantics, and the translation of a subset of PPOs into networks of timed automata together with a proof of its correctness. Section 4 introduces the new model of real-time task systems. Section 5 explains how the translation of PPOs to Uppaal can be lifted to real-time task systems. Section 6 presents performance evaluation results of models generated by comparing them with handcrafted Uppaal models presented before in our papers. Concluding remarks and future work follow in Section 7. The models generated for Section 6 are available at `www.mbsd.cs.ru.nl/publications/papers/fvaan/HIV12`.

2 Preliminaries

We use $\mathbb{R}_{\geq 0}$ and $\mathbb{R}_{>0}$ to denote the sets of nonnegative and positive real numbers, respectively, and $\mathbb{N}$ to denote the set of natural numbers.

If X and Y are sets then we write $X \hookrightarrow Y$ for the set of partial functions from X to Y. Given a partial function $f \in X \hookrightarrow Y$, we write $f(x)\downarrow$ if $f(x)$ is defined, and $f(x)\uparrow$ if $f(x)$ is undefined, for $x \in X$.

A *labeled transition system (LTS)* is a tuple $\mathcal{L} = (S, s_0, \Sigma, \rightarrow)$, where:

- S is a set of states,
- $s_0 \in S$ is an initial state,
- Σ is a set of action labels, and
- $\rightarrow \subseteq S \times \Sigma \times S$ is a transition relation.

We write $s \xrightarrow{a} s'$ iff $(s, a, s') \in \rightarrow$ and $s \rightarrow s'$ if there exists an action $a \in \Sigma$ such that $s \xrightarrow{a} s'$. A *path* of $\mathcal{L}$ is a sequence of states $\pi = s_0 s_1 \cdots s_n$ such that, for all $0 \leq i < n$, $s_i \rightarrow s_{i+1}$. In this case we say π is a path from s_0 to s_n. A state $s \in S$ is *reachable* in $\mathcal{L}$ if there exists a path from s_0 to s. We say that $\mathcal{L}$ is *deterministic* if, for each state $s \in S$ and for each action label $a \in \Sigma$, $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ implies $s' = s''$.

Two labeled transition systems $\mathcal{L}_1 = (S_1, s_0^1, \Sigma_1, \rightarrow_1)$ and $\mathcal{L}_2 = (S_2, s_0^2, \Sigma_2, \rightarrow_2)$ are *isomorphic* if $\Sigma_1 = \Sigma_2$ and there exists a bijective function $f : S_1 \rightarrow S_2$ such that:

- $f(s_0^1) = s_0^2$ and
- $s \xrightarrow{a}_1 s' \Leftrightarrow f(s) \xrightarrow{a}_2 f(s')$, for all $s, s' \in S_1$, $a \in \Sigma_1$.

We say that $\mathcal{L}_1$ and $\mathcal{L}_2$ are *bisimilar* if $\Sigma_1 = \Sigma_2$ and there exists a relation $R \subseteq S_1 \times S_2$ such that $(s_0^1, s_0^2) \in R$ and, for each $(s, r) \in R$,

- if $s \xrightarrow{a}_1 s'$ then there exists a state $r' \in S_2$ such that $r \xrightarrow{a}_2 r'$ and $(s', r') \in R$, and
- if $r \xrightarrow{a}_2 r'$ then there exists a state $s' \in S_1$ such that $s \xrightarrow{a}_1 s'$ and $(s', r') \in R$.

Given an LTS $\mathcal{L} = (S, s_0, \Sigma, \rightarrow)$, we define $\mathsf{reach}(\mathcal{L})$ to be the LTS $(S', s_0, \Sigma, \rightarrow')$, where S' is the set of reachable states of $\mathcal{L}$ and $\rightarrow' = \{(s, a, s') \mid s, s' \in S' \wedge s \xrightarrow{a} s'\}$.

3 Parameterized Partial Orders

A *parametrized partial order (PPO)* is a partial order that comes equipped with some extra structure to capture repetitive behavior. In [15], a PPO is defined at task level and assumes a precedence relation between tasks. Here we view a PPO from a different angle where tasks are decomposed into events and a PPO imposes a partial order relation at event level. This perspective allows us to introduce a subclass of PPOs that can be efficiently translated into networks of automata, and later in this section we establish the correctness of this translation.

3.1 Definition of PPOs

Tasks in a PPO may be executed repeatedly: each task has a collection of parameters and each valuation of these parameters defines a task instance. The events in a PPO are structured and correspond to either the start or the end of a task instance.

Formally, we assume a universe $\mathcal{P}$ of typed variables called *parameters*. A *valuation* of a set $P \subseteq \mathcal{P}$ of parameters is a function that maps each parameter in P to an element of its domain. We assume that the domain of each parameter is a nonempty set. We write $V(P)$ for the set of valuations of variables in P.

A *parameterized partial order (PPO)* is a tuple $\mathcal{A} = (\mathcal{T}, \mathcal{M}, E, U)$ where

- $\mathcal{T}$ is a finite set of *tasks*. We define the set of *event types* by $\mathcal{E} = \{s, e\} \times \mathcal{T}$. Projection functions $\mathsf{task} : \mathcal{E} \rightarrow \mathcal{T}$ and $\mathsf{type} : \mathcal{E} \rightarrow \{\mathsf{s}, \mathsf{e}\}$ are given by $\mathsf{task}((t, T)) = T$ and $\mathsf{type}((t, T)) = t$, and embeddings $\mathsf{start} : \mathcal{T} \rightarrow \mathcal{E}$ and $\mathsf{end} : \mathcal{T} \rightarrow \mathcal{E}$ are given by $\mathsf{start}(T) = (s, T)$ and $\mathsf{end}(T) = (e, T)$, with $t \in \{s, e\}$, and $T \in \mathcal{T}$.

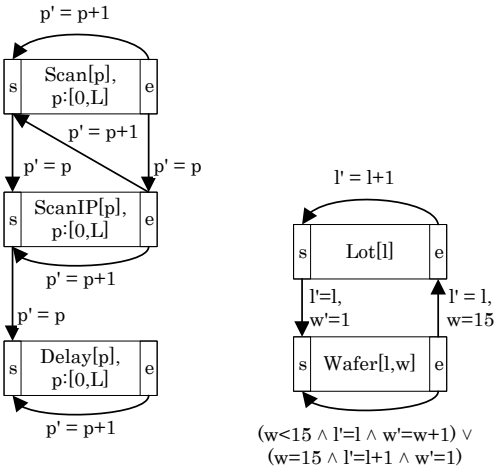
- $\mathcal{M}$ is a function that assigns to each task T a finite set of parameters in $\mathcal{P}$; we write $V(T)$ as a shorthand for $V(\mathcal{M}(T))$.
- $E \subseteq \mathcal{E} \times \mathcal{E}$ is a set of *edges*. We require, for each $T \in \mathcal{T}$, $(\mathsf{start}(T), \mathsf{end}(T)) \in E$.
- For each edge $p = (A, B) \in E$, $U(p) : V(\mathsf{task}(A)) \hookrightarrow V(\mathsf{task}(B))$ is a *precedence function*. We write $A \xrightarrow{u} B$ if $(A, B) \in E$ and $U(A, B) = u$. We require that the start of a task instance precedes the end of that instance, that is, for each task $T \in \mathcal{T}$ and valuation $v \in V(T)$, $U((\mathsf{start}(T), \mathsf{end}(T)))(v) = v$.

Below, we present two examples that illustrate how PPOs can be used to model scheduling applications.

Example 1 (Printer). Figure 1a depicts a part of an application encountered in the printer domain (see [23]). There are three tasks: Scan, ScanIP and Delay, represented by rectangles. The corresponding start and end event types are indicated by subrectangles inscribed with s and e. Edges show the dependencies between event types (the edges from start to corresponding end are not shown). All three tasks have one parameter: p of type $[0, \ldots, L]$ representing the number of the current page processed. The constant $L \in \mathbb{N}$ is a bound for the parameter p. A precedence function $A \xrightarrow{u} B$ is represented by a predicate that may contain both the parameters of $\mathsf{task}(A)$ and primed versions of the parameters of $\mathsf{task}(B)$. For instance, the predicate $\mathsf{p}' = \mathsf{p} + 1$ on the edge from ScanIP to Scan represents the precedence function that maps a valuation v of the ScanIP parameters to the unique valuation v' of the Scan parameters that satisfies $v'(\mathsf{p}) = v(\mathsf{p}) + 1$.

An instance of ScanIP may start as soon as its corresponding instance of Scan has started. These task instances of Scan and ScanIP may then proceed in parallel. However, the next instance of Scan may only start after the current instances of both Scan and ScanIP have ended. Between the ScanIP and Delay tasks, there is also a dependency: only after the occurrence of the start event in the ScanIP task, the start event of the corresponding Delay task may occur.

Example 2 (Wafer production). The PPO displayed in Figure 1b describes the production of an infinite series of lots, where each lot is composed of 15 wafers. This example is inspired by [5]. After the start of each lot, 15 wafer tasks are executed in sequence, followed by the end of the lot.



(a) Printer (b) Wafer production

Fig. 1: PPO representation

3.2 From PPOs to Configuration Structures

The semantics of a PPO can be described in terms of a labeled transition system, referred to as the *configuration structure* of the PPO (see [8,33]). The states of a configuration structure are *configurations*, finite sets of events that have already occurred. Each transition marks the occurrence of a single new event for which all the immediate predecessors have occurred.

Formally, an *event* is a pair (A, v) where A is an event type and $v \in V(\mathsf{task}(A))$ is a valuation of its task parameters. We write $\mathsf{ev\_type}((A, v)) = A$ and $\mathsf{task}((A, v)) = \mathsf{task}(A)$. Also, we write $\mathsf{ev}(\mathcal{A})$ for the set of events of a PPO $\mathcal{A}$. We call event (B, w) an *immediate predecessor* of event (A, v), notation $(B, w) \mapsto (A, v)$, if $(B, A) \in E \wedge U(B, A)(w) = v$.

Let $C \subset \mathsf{ev}(\mathcal{A})$ and $\alpha \in \mathsf{ev}(\mathcal{A})$ with $\alpha \notin C$. We say that C *enables* α, and write $C \vdash \alpha$, if all immediate predecessors of α are in C.

Let $\mathcal{A}$ be a PPO. The set $\mathsf{conf}(\mathcal{A})$ of *configurations* of $\mathcal{A}$ is the smallest subset of the power set $\wp(\mathsf{ev}(\mathcal{A}))$ of events of $\mathcal{A}$ such that:

1. $\emptyset \in \mathsf{conf}(\mathcal{A})$,
2. if $C \in \mathsf{conf}(\mathcal{A})$, and $C \vdash \alpha$ then $C \cup \{\alpha\} \in \mathsf{conf}(\mathcal{A})$.

The *configuration structure* of $\mathcal{A}$ is the LTS $\mathcal{C}(\mathcal{A}) = (\mathsf{conf}(\mathcal{A}), \emptyset, \mathcal{E}, \rightsquigarrow)$, where $(C, A, C \cup \{\alpha\}) \in \rightsquigarrow$ iff $C \in \mathsf{conf}(\mathcal{A})$, $\mathsf{ev\_type}(\alpha) = A$ and $C \vdash \alpha$. We write $C \overset{A}{\rightsquigarrow} C'$ if $(C, A, C') \in \rightsquigarrow$. Also, we sometimes write $C \overset{\alpha}{\rightsquigarrow} C'$ to denote that $C \overset{\mathsf{ev\_type}(\alpha)}{\rightsquigarrow} C'$ and $C' = C \cup \{\alpha\}$.

The above definition implies that each configuration $C \in \mathsf{conf}(\mathcal{A})$ has a *securing*, that is, a sequence $\alpha_1, \ldots, \alpha_n$ of events such that $C = \{\alpha_1, \ldots, \alpha_n\}$ and, for each $1 \leq i \leq n$, $\{\alpha_j \mid j < i\} \in \mathsf{conf}(A)$ and $\{\alpha_j \mid j < i\} \vdash \alpha_i$.

In a PPO there are no conflicts between events: it is not possible that the occurrence of one event disables the occurrence of another event. In fact, it is easy to prove that the set of configurations of a PPO is closed under union: if $C \in \mathsf{conf}(\mathcal{A})$ and $C' \in \mathsf{conf}(\mathcal{A})$ then $C \cup C' \in \mathsf{conf}(\mathcal{A})$. We call an event *reachable* if it occurs in some configuration, and write $\mathsf{rev}(\mathcal{A})$ for the set of reachable events of $\mathcal{A}$. Note that, since in a PPO we allow cyclic predecessor relations, it may occur that some

(or even all) events are not reachable. If α and α' are in $\mathsf{rev}(\mathcal{A})$, we write $\alpha \leq_{\mathcal{A}} \alpha'$, if for each configuration $C \in \mathsf{conf}(\mathcal{A})$, $\alpha' \in C$ implies $\alpha \in C$. The technical lemma below states that the $\leq_{\mathcal{A}}$ contains the immediate predecessor relation:

Lemma 1. *Let $\mathcal{A}$ be a PPO with events α and α' such that $\alpha \mapsto \alpha'$. Then $\alpha' \in \mathsf{rev}(\mathcal{A})$ implies $\alpha \in \mathsf{rev}(\mathcal{A})$ and $\alpha \leq_{\mathcal{A}} \alpha'$.*

Proof. If $\alpha' \in \mathsf{rev}(\mathcal{A})$, then there is a configuration $C \in \mathsf{conf}(\mathcal{A})$ that contains α'. Furthermore, this configuration has a securing, that is, a sequence $\alpha_1, ..., \alpha_n$ such that $C = \{\alpha_1, ..., \alpha_n\}$ and there is a configuration C'_α that we can construct with some of the events of C that enables α'. Since $C'_\alpha \vdash \alpha'$, C'_α contains all immediate predecessors of α'. Let α be an immediate predecessor of α'. Since $\alpha \in C'_\alpha$, then $\alpha \in \mathsf{rev}(\mathcal{A})$. Because $C'_\alpha \subset C$, then $\alpha \in C$, namely α is include in any configuration that contains α', therefore $\alpha \leq_{\mathcal{A}} \alpha'$.

The following lemma states that a parametrized partial order (PPO) induces a partial ordering relation on its (reachable) events.

Lemma 2. *Let $\mathcal{A}$ be a PPO, then $\leq_{\mathcal{A}}$ is a partial order on $\mathsf{rev}(\mathcal{A})$.*

Proof.
1. (Reflexivity). We need to prove that $\alpha \leq_{\mathcal{A}} \alpha$ is true, for any $\alpha \in \mathsf{rev}(\mathcal{A})$, which holds.
2. (Antisymmetry). Let $\alpha, \alpha' \in \mathsf{rev}(\mathcal{A})$. We should prove that if $\alpha \leq_{\mathcal{A}} \alpha'$, and $\alpha' \leq_{\mathcal{A}} \alpha \implies \alpha = \alpha'$. Assume that $\alpha \neq \alpha'$. If $\alpha \leq_{\mathcal{A}} \alpha'$, there exists a securing whose configuration C'_α contains α and enables α'. Further, if $\alpha' \leq_{\mathcal{A}} \alpha$, there exists a securing whose configuration C_α contains α' and enables α and $C'_\alpha \subset C_\alpha$. This implies that $\alpha \in C_\alpha$ and $C_\alpha \vdash \alpha$, which is impossible.
3. (Transitivity). Let $\alpha, \alpha', \gamma \in \mathsf{rev}(\mathcal{A})$. We should prove that if $\alpha \leq_{\mathcal{A}} \alpha'$, and $\alpha' \leq_{\mathcal{A}} \gamma \implies \alpha \leq_{\mathcal{A}} \gamma$. Assuming that $\alpha \leq_{\mathcal{A}} \alpha'$, this implies that any configuration that contains α' also contains α. Further, if $\alpha' \leq_{\mathcal{A}} \gamma$, any configuration that contains γ also contains α'. Since any configuration that contains α', also contains α, this allows us to conclude that any configuration that contains γ also contains α, and therefore $\alpha \leq_{\mathcal{A}} \gamma$.

3.3 Restricted PPOs

We explore the behavior of PPOs using the Uppaal model checker, and for this we need to translate PPOs to the input language of Uppaal. Here we describe a translation of a subclass of PPOs in which no two instances of a task can run concurrently. It is possible to translate arbitrary PPOs to Uppaal (provided the parameter domains are finite) but this translation leads to networks of automata that are much harder to analyze.

We call a PPO $\mathcal{A}$ *restricted* if it satisfies the following five conditions, for all tasks T and T', for all precedence functions $A \xrightarrow{u} B$ with $\mathsf{task}(A) = T$ and $\mathsf{task}(B) = T'$, and for all valuations $v, w \in V(T)$:

- **C0**: The only edges between events of the same task are the ones from the start event to the end event, and from the end event to the start event:

  $$\mathsf{task}(A) = \mathsf{task}(B) \Rightarrow ((A, B) \in E \Leftrightarrow A \neq B)$$

  We write $\mathsf{next}(T)$ for the function $U((\mathsf{end}(T), \mathsf{start}(T))$, and let $<_T$ be the least transitive relation on valuations in $V(T)$ satisfying $v <_T \mathsf{next}(T)(v)$. Write $v \leq_T w$ iff $v <_T w$ or $v = w$.
- **C1**: There is exactly one valuation of the parameters of T that does not appear in the range of $\mathsf{next}(T)$. This valuation is referred to as the *initial valuation* of T, and is written v^0_T.
- **C2**: $\mathsf{next}(T)$ is injective
- **C3**: u is only defined for reachable valuations:

  $$u(v) \downarrow \Rightarrow v^0_T \leq_T v$$

- **C4**: u is *monotonic*:

  $$v \leq_T w \wedge u(w) \downarrow \Rightarrow u(v) \downarrow \wedge\ u(v) \leq_{T'} u(w)$$

Axioms **C0**, **C1** and **C2** impose precedence restrictions between event instances of the same task that exclude auto-concurrency. Axiom **C0** implies that we have an edge from the end event type of a task to the corresponding start event type. Axiom **C1** implies that, for each task, there is only one event that does not depend on some other event of the same task: necessarily this is going to be the first event of the task that will occur. Axiom **C2** implies that each event of a task, except the initial one, has a unique immediate predecessor event that belongs to the same task. Axioms **C0-C2** still allow cyclic precedence edges between events of the same task, but axiom **C3** implies that u is not defined for such "ghost events". Axiom **C4**, finally, states that a precedence function that links events of different tasks is monotonic w.r.t the event ordering within tasks. The reader may check that the examples of subsection 1 are restricted.

Lemma 3. *Let $\mathcal{A}$ be a restricted PPO. Given a task T and valuation v Then*

1. $(\mathsf{end}(T), v) \in \mathsf{rev}(\mathcal{A})$ *implies* $(\mathsf{start}(T), v) \in \mathsf{rev}(\mathcal{A})$ *and* $(\mathsf{start}(T), v) \leq_{\mathcal{A}} (\mathsf{end}(T), v)$.
2. $(\mathsf{start}(T), \mathsf{next}(T)(v)) \in \mathsf{rev}(\mathcal{A})$ *implies* $(\mathsf{end}(T), v) \in \mathsf{rev}(\mathcal{A})$ *and* $(\mathsf{end}(T), v) \leq_{\mathcal{A}} (\mathsf{start}(T), \mathsf{next}(T)(v))$.
3. $\leq_{\mathcal{A}}$ *is a total order on the set* $\{\alpha \in \mathsf{rev}(\mathcal{A}) \mid \mathsf{task}(\alpha) = T\}$ *of reachable events of* T.

Proof. Statements (1) and (2) follow by Lemma 1.
For (3), first observe that $\leq_{\mathcal{A}}$ is a partial order on $\mathsf{rev}(\mathcal{A})$ by Lemma 2. Hence it is also a partial order on the subset of reachable events of T. Let $\alpha, \alpha' \in \mathsf{rev}(\mathcal{A})$ with $\mathsf{task}(\alpha) = T$ and $\mathsf{task}(\alpha') = T$. It suffices to prove that either $\alpha \leq_{\mathcal{A}} \alpha'$ or $\alpha' \leq_{\mathcal{A}} \alpha$. Assuming that $\alpha = (t_\alpha, v_\alpha), \alpha' =$

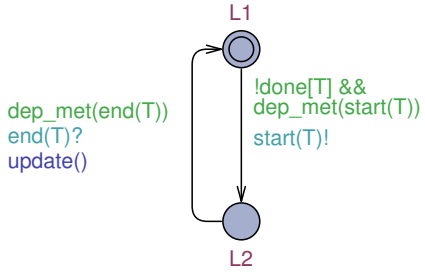


Fig. 2: Automaton for task T

(t'_α, v'_α), with $t_\alpha, t'_\alpha \in \{\mathsf{end}(T), \mathsf{start}(T)\}$. If $v_\alpha = v'_\alpha$, then by applying the first case of this lemma, it follows that $\alpha \leq_{\mathcal{A}} \alpha'$ or $\alpha' \leq_{\mathcal{A}} \alpha$. If not, without loss of generality, we assume $v_\alpha <_T v'_\alpha$. Then using the first two cases of this lemma, it follows that $(t_\alpha, v_\alpha) \leq_{\mathcal{A}} (\mathsf{start}(T), \mathsf{next}(T)(v_\alpha)) \leq_{\mathcal{A}} \ldots \leq_{\mathcal{A}} (t'_\alpha, v'_\alpha)$, which by transitivity implies that $\alpha \leq_{\mathcal{A}} \alpha'$.

3.4 From Restricted PPOs to Networks of Automata

We will show how each restricted PPO can be translated into a Uppaal-style parallel composition of a number of automata in such a way that (the reachable part of) the LTS induced by the composition of these automata is isomorphic to the configuration structure of the PPO. We refer the reader to [14] for an introduction to Uppaal.

Let $\mathcal{A}$ be a PPO as above. We define $\mathcal{N}(\mathcal{A})$ to be the LTS induced by the parallel composition of automata that instantiate the template displayed in Figure 2, for each task $T \in \mathcal{T}$. Below we explain the various predicates and functions occurring in Figure 2. The composed system $\mathcal{N}(\mathcal{A})$ has the following set of global shared variables:

$$\{T.\mathsf{p}, \mathsf{loc}[T], \mathsf{done}[T] \mid T \in \mathcal{T} \wedge \mathsf{p} \in \mathcal{M}(T)\}.$$

Variable $\mathsf{loc}[T]$ records the current location of the task automaton for T, which can be either $\mathsf{L1}$ or $\mathsf{L2}$. Boolean variable $\mathsf{done}[T]$ records whether the last event of T has been executed. Since different tasks may use the same parameter names, we make a copy $T.\mathsf{p}$ of each parameter $\mathsf{p} \in \mathcal{M}(T)$. As long as task T has not yet been completed, variable $T.\mathsf{p}$ gives the value of p in the next event of T that will occur. Variable $\mathsf{loc}[T]$ is initialized to $\mathsf{L1}$, variable $\mathsf{done}[T]$ is initialized to false, and variable $T.\mathsf{p}$ is initialized to $v_T^0(\mathsf{p})$, for each parameter $\mathsf{p} \in \mathcal{M}(T)$.

For a given state of the automaton for task T, let function $\mathsf{val}(T)$ return the current valuation of the parameters of task T. For each event type A with $\mathsf{task}(A) = T$, function $\mathsf{done}(A)$ returns true iff the last event of A has occurred:

$$\begin{aligned}\mathsf{done}(A) = \mathsf{done}[T] \vee (\mathsf{loc}[T] = \mathsf{L2} \wedge \\ \mathsf{type}(A) = \mathsf{s} \wedge \mathsf{next}(T)(\mathsf{val}(T)) \uparrow)\end{aligned}$$

If the last event of A has not occurred, function $\mathsf{next}(A)$ gives the valuation of the parameters for the next event of A:

$$\mathsf{next}(A) = \begin{cases}\mathsf{next}(T)(\mathsf{val}(T)), & \text{if } \mathsf{loc}[T] = \mathsf{L2} \wedge \mathsf{type}(A) = \mathsf{s} \\ \mathsf{val}(T) & \text{, otherwise}\end{cases}$$

Suppose that the last event of type A has not occurred, then in order to decide whether the next event of A may occur, we check for each incoming precedence edge $B \xrightarrow{u} A$ whether the dependency induced by that edge has been met:

$$\begin{aligned}\mathsf{dep\_met}(A) = \forall B, u : B \xrightarrow{u} A \wedge \mathsf{task}(B) \neq \mathsf{task}(A) \implies \\ \mathsf{dep\_met}(B, u, A)\end{aligned}$$

Note that the task automaton already takes care of the dependencies induced by precedence functions between pairs of start and end events of T. In order to decide whether the dependencies induced by $B \xrightarrow{u} A$ are met, we first check if $\mathsf{done}(B)$ evaluates to true. If so then all events of B have occurred and hence all dependencies induced by $B \xrightarrow{u} A$ have been met. Next we check whether $u(\mathsf{next}(B))$ is defined. If not then, by monotonicity, all dependencies induced by $B \xrightarrow{u} A$ have been met. Finally, we check whether $\mathsf{next}(A)$ precedes $u(\mathsf{next}(B))$. If so, then for any immediate predecessor of $\mathsf{next}(A)$, that is, for any parameter valuation v of B with $u(v) = \mathsf{next}(A)$, monotonicity implies $v < \mathsf{next}(B)$. Formally,

$$\begin{aligned}\mathsf{dep\_met}(B, u, A) = \mathsf{done}(B) \vee u(\mathsf{next}(B)) \uparrow \\ \vee\ \mathsf{next}(A) <_T u(\mathsf{next}(B))\end{aligned}$$

Finally, function $\mathsf{update}()$ sets $\mathsf{done}[T]$ to true if the last event for task T has occurred, and otherwise updates the parameters of T according to function $\mathsf{next}(T)$.

Lemma 4. *For all reachable states s of $\mathcal{N}(\mathcal{A})$ and for all tasks $T \in \mathcal{T}$, the following invariant properties hold:*

1. $v_T^0 \leq_T s.\mathsf{val}(T)$
2. $s.\mathsf{done}[T] \Rightarrow \mathsf{next}(T)(s.\mathsf{val}(T)) \uparrow$
3. $s.\mathsf{done}[T] \Rightarrow s.\mathsf{loc}[T] = \mathsf{L1}$

Proof. Straightforward by induction on the length of the shortest path leading to s.

Theorem 1. *Let $\mathcal{A}$ be a PPO. Then LTSs $\mathcal{C}(\mathcal{A})$ and $\mathsf{reach}(\mathcal{N}(\mathcal{A}))$ are isomorphic.*

Proof. Let $\mathcal{N}(\mathcal{A}) = (S, s_0, \mathcal{E}, \rightarrow)$. If $s \in S$ is a state and e is an expression containing variables of $\mathcal{N}(\mathcal{A})$, then we write $s.e$ for the result of evaluating expression e in state s. For each event type $A \in \mathcal{E}$, we define a function $\Re_A : S \rightarrow 2^{\mathsf{ev}(\mathcal{A})}$ that associates to each state of $\mathcal{N}(\mathcal{A})$ a set of events of type A. Intuitively, this is the set of events of type A that have occurred before reaching state s. Suppose $\mathsf{task}(A) = T$. Then

$$\begin{aligned}\Re_A(s) = \textbf{if } s.\mathsf{done}(A) \textbf{ then} \\ \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v \leq_T s.\mathsf{val}(T)\} \\ \textbf{else} \\ \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{next}(A)\} \\ \textbf{fi}\end{aligned}$$

Let function $\Re : S \to 2^{\mathsf{ev}(\mathcal{A})}$ be defined by:

$$\Re(s) = \bigcup_{A \in \mathcal{E}} \Re_A(s)$$

We will prove that $\Re$ is an isomorphism from $\mathsf{reach}(\mathcal{N}(\mathcal{A}))$ to $\mathcal{C}(\mathcal{A})$.

Claim 1. $\Re(s_0) = \emptyset$.

Proof. Let A be an event type. Let $\mathsf{task}(A) = T$. By definition of s_0 we have $s_0.\mathsf{done}(A) = \mathsf{false}$ and $s_0.\mathsf{next}(A) = v_T^0$. Hence, by definition of $\Re_A$, $\Re_A(s_0) = \{(A, v) \mid v <_T v_T^0\}$. But since, by condition **C1**, v_T^0 does not appear in the range of $\mathsf{next}(T)$, there exists no v such that $v <_T v_T^0$. Hence $\Re_A(s_0) = \emptyset$. Since A has been chosen arbitrarily, it follows that also $\Re(s_0) = \emptyset$.

Claim 2. If s is a reachable state and $s \xrightarrow{A} s'$ then $\Re(s) \vdash (A, s.\mathsf{val}(T))$.

Proof. Let $v = s.\mathsf{val}(T)$. Assume that $s \xrightarrow{A} s'$ and assume that (B, w) is an immediate predecessor of (A, v). It suffices to prove that $(B, w) \in \Re_B(s)$.

If $\mathsf{task}(B) = \mathsf{task}(A)$ and $A = \mathsf{start}(T)$ then, by **C0**, $B = \mathsf{end}(T)$ and $\mathsf{next}(T)(w) = v$. Since $s \xrightarrow{A} s'$, $s.\mathsf{done}[T] = \mathsf{false}$. This implies $s.\mathsf{done}(B) = \mathsf{false}$. Also $s.\mathsf{next}(B) = s.\mathsf{val}(T) = v$. We infer that

$$\Re_B(s) = \{(B, x) \in \mathsf{ev}(\mathcal{A}) \mid x <_T v\}$$

Since $w <_T v$ it follows that $(B, w) \in \Re_B(s)$, as required.

If $\mathsf{task}(B) = \mathsf{task}(A)$ and $A = \mathsf{end}(T)$ then $B = \mathsf{start}(T)$ and $w = v$. If $s.\mathsf{done}(B)$ holds then $(B, w) \in \Re_B(s)$ and we are done. If $s.\mathsf{done}(B)$ does not hold then $\mathsf{next}(T)(\mathsf{val}(T))) \downarrow$ and $\mathsf{next}(B) = \mathsf{next}(T)(\mathsf{val}(T)))$. It follows that $(B, w) \in \Re_B(s)$.

We may therefore assume that $\mathsf{task}(B) \neq \mathsf{task}(A)$. Let $U(B, A) = u$ and $\mathsf{task}(B) = T'$. Then $u(w) = v$. Since $s \xrightarrow{A} s'$, $s.\mathsf{dep\_met}(B, u, A)$ holds. This means that one of the following three cases applies:

- $s.\mathsf{done}(B)$.
  Using the first invariant of Lemma 4, we infer $v_{T'}^0 \leq_{T'} s.\mathsf{val}(T')$. Using the second invariant of Lemma 4, we infer that $\mathsf{next}(T')(s.\mathsf{val}(T')) \uparrow$. Condition **C3** implies that $v_{T'}^0 \leq_{T'} w$. It follows that $w \leq_{T'} s.\mathsf{val}(T')$. Hence $(B, w) \in \Re_B(s)$, as required.
- $s.\mathsf{done}(B) = \mathsf{false}$ and $u(s.\mathsf{next}(B)) \uparrow$.
  By monotonicity imposed by condition **C4**, we do not have $s.\mathsf{next}(B) <_{T'} w$. Condition **C3** implies $v_{T'}^0 \leq_{T'} w$, and Lemma 4 implies $v_{T'}^0 \leq_{T'} s.\mathsf{next}(B)$. Hence $w <_{T'} s.\mathsf{next}(B)$ and thus $(B, w) \in \Re_B(s)$.
- $s.\mathsf{next}(A) <_T u(s.\mathsf{next}(B))$.
  Since $s \xrightarrow{A} s'$, $s.\mathsf{next}(A) = s.\mathsf{val}(T) = v$. As in the previous case, we use conditions **C3**, **C4** and Lemma 4 to argue that $w <_{T'} s.\mathsf{next}(B)$, and thus $(B, w) \in \Re_B(s)$.

Claim 3. If $s \xrightarrow{A} s'$ then $\Re(s') = \Re(s) \cup \{(A, s.\mathsf{val}(T))\}$.

Proof. Assume $s \xrightarrow{A} s'$. It is easy to check that for all event types B with $\mathsf{task}(B) \neq \mathsf{task}(A)$, $\Re_B(s') = \Re_B(s)$. Let $\overline{\_} : \mathcal{E} \to \mathcal{E}$ be the function given by $\overline{\mathsf{start}(T)} = \mathsf{end}(T)$ and $\overline{\mathsf{end}(T)} = \mathsf{start}(T)$, for all T. We claim that $\Re_A(s') = \Re_A(s) \cup \{(A, s.\mathsf{val}(T))\}$ and $\Re_{\overline{A}}(s') = \Re_{\overline{A}}(s)$. We consider four cases:

- $A = \mathsf{start}(T)$ and $\mathsf{next}(T)(s.\mathsf{val}(T)) \uparrow$.
  Since $s \xrightarrow{A} s'$, $s.\mathsf{next}(A) = s.\mathsf{val}(T)$ and $s.\mathsf{done}(A) = \mathsf{false}$. Hence
  $$\Re_A(s) = \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\}$$
  Since $s \xrightarrow{A} s'$, $s'.\mathsf{loc}[T] = \mathsf{L2}$ and $s'.\mathsf{val}(T) = s.\mathsf{val}(T)$. Since $\mathsf{next}(T)(s'.\mathsf{val}(T)) \uparrow$, then $s'.\mathsf{done}(A)$. Hence
  $$\Re_A(s') = \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v \leq_T s.\mathsf{val}(T)\}$$
  Thus $\Re_A(s') = \Re_A(s) \cup \{(A, s.\mathsf{val}(T))\}$. Since $s \xrightarrow{A} s'$, $s.\mathsf{done}(\mathsf{end}(T)) = \mathsf{false}$ and $s'.\mathsf{done}(\mathsf{end}(T)) = \mathsf{false}$. Moreover $s'.\mathsf{next}(\mathsf{end}(T)) = s.\mathsf{next}(\mathsf{end}(T)) = s.\mathsf{val}(T)$. Hence
  $$\begin{aligned}\Re_{\overline{A}}(s') &= \Re_{\overline{A}}(s) \\ &= \{(\overline{A}, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\}\end{aligned}$$
- $A = \mathsf{start}(T)$ and $\mathsf{next}(T)(s.\mathsf{val}(T)) \downarrow$.
  Since $s \xrightarrow{A} s'$, $s.\mathsf{next}(A) = s.\mathsf{val}(T)$ and $s.\mathsf{done}(A) = \mathsf{false}$. Hence
  $$\Re_A(s) = \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\}$$
  Since $s \xrightarrow{A} s'$, $s'.\mathsf{loc}[T] = \mathsf{L2}$ and $s'.\mathsf{val}(T) = s.\mathsf{val}(T)$. Since $\mathsf{next}(T)(s'.\mathsf{val}(T)) \downarrow$, then $s'.\mathsf{done}(A) = \mathsf{false}$ and $s'.\mathsf{next}(A) = \mathsf{next}(T)(s'.\mathsf{val}(T))$. Hence
  $$\Re_A(s') = \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T \mathsf{next}(T)(s.\mathsf{val}(T))\}$$
  By **C2**, $\Re_A(s') = \Re_A(s) \cup \{(A, s.\mathsf{val}(T))\}$. Since $s \xrightarrow{A} s'$, $s.\mathsf{done}(\mathsf{end}(T)) = \mathsf{false}$ and $s'.\mathsf{done}(\mathsf{end}(T)) = \mathsf{false}$. Moreover $s'.\mathsf{next}(\mathsf{end}(T)) = s.\mathsf{next}(\mathsf{end}(T)) = s.\mathsf{val}(T)$. Hence
  $$\begin{aligned}\Re_{\overline{A}}(s') &= \Re_{\overline{A}}(s) \\ &= \{(\overline{A}, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\}\end{aligned}$$
- $A = \mathsf{end}(T)$ and $\mathsf{next}(T)(s.\mathsf{val}(T)) \uparrow$.
  Since $s \xrightarrow{A} s'$, $\mathsf{done}(A) = \mathsf{false}$ and $s.\mathsf{next}(A) = s.\mathsf{val}(T)$. Hence
  $$\Re_A(s) = \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\}$$
  Moreover, $s'.\mathsf{done}[T]$, $s'.\mathsf{done}(A)$ and $s'.\mathsf{val}(T) = s.\mathsf{val}(T)$. Hence
  $$\Re_A(s') = \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v \leq_T s.\mathsf{val}(T)\}$$
  Thus $\Re_A(s') = \Re_A(s) \cup \{(A, s.\mathsf{val}(T))\}$. By the assumptions, $s.\mathsf{done}(\overline{A})$, we can also infer $s'.\mathsf{done}(\overline{A})$. Hence
  $$\begin{aligned}\Re_{\overline{A}}(s') &= \Re_{\overline{A}}(s) \\ &= \{(\overline{A}, v) \in \mathsf{ev}(\mathcal{A}) \mid v \leq_T s.\mathsf{val}(T)\}\end{aligned}$$

– $A = \mathsf{end}(T)$ and $\mathsf{next}(T)(s.\mathsf{val}(T)) \downarrow$.
Since $s \xrightarrow{A} s'$, $\mathsf{done}(A) = \mathsf{false}$ and $s.\mathsf{next}(A) = s.\mathsf{val}(T)$. Hence

$$\Re_A(s) = \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\}$$

Moreover, $s'.\mathsf{done}(A) = \mathsf{false}$, $s'.\mathsf{next}(A) = s'.\mathsf{val}(T)$ and $s'.\mathsf{val}(T) = \mathsf{next}(T)(s.\mathsf{val}(T))$. Hence

$$\Re_A(s') = \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T \mathsf{next}(T)(s.\mathsf{val}(T))\}$$

By **C2**, $\Re_A(s') = \Re_A(s) \cup \{(A, s.\mathsf{val}(T))\}$. By the assumptions, $s.\mathsf{done}(\overline{A}) = \mathsf{false}$ and $s'.\mathsf{done}(\overline{A}) = \mathsf{false}$. Moreover

$$s.\mathsf{next}(\overline{A}) = \mathsf{next}(T)(s.\mathsf{val}(T)) = s'.\mathsf{val}(T) = s'.\mathsf{next}(\overline{A})$$

This implies

$$\Re_{\overline{A}}(s') = \Re_{\overline{A}}(s)$$

It follows that $\Re(s') = \Re(s) \cup \{(A, s.\mathsf{val}(T))\}$.

Claim 4. If s is a reachable state of $\mathcal{N}(\mathcal{A})$ then $\Re(s) \in \mathsf{conf}(\mathcal{A})$.

Proof. Straightforward, by induction on the length of the shortest path to s, using Claims 1-3.

Claim 5. If s, s' are reachable states of $\mathcal{N}(\mathcal{A})$ and $s \xrightarrow{A} s'$ then $\Re(s) \overset{A}{\rightsquigarrow} \Re(s')$.

Proof. Straightforward, by combining Claims 2, 3 and 4.

In order to prove that $\Re$ is bijective, we define an inverse function $\mathfrak{S}$ that maps configurations of $\mathcal{A}$ to states of $\mathcal{N}(\mathcal{A})$. Let C be a configuration and let T be a task. Write C_T for the subset of C of events of type T. We consider four cases:

1. If $C_T = \emptyset$ then variable $\mathsf{loc}[T]$ is set to $\mathsf{L1}$, variable $\mathsf{done}[T]$ is set to false, and variable $T.\mathsf{p}$ is set to $v^0_T(\mathsf{p})$, for each parameter $\mathsf{p} \in \mathcal{M}(T)$.
2. If $C_T \neq \emptyset$ and the unique maximal event of C_T (cf Lemma 3) is of the form $(\mathsf{start}(T), v)$, then variable $\mathsf{loc}[T]$ is set to $\mathsf{L2}$, variable $\mathsf{done}[T]$ is set to false, and variable $T.\mathsf{p}$ is set to $v(\mathsf{p})$, for each parameter $\mathsf{p} \in \mathcal{M}(T)$.
3. If $C_T \neq \emptyset$, the unique maximal event of C_T is of the form $(\mathsf{end}(T), v)$ and $\mathsf{next}(T)(v) \downarrow$, then variable $\mathsf{loc}[T]$ is set to $\mathsf{L1}$, variable $\mathsf{done}[T]$ is set to false, and variable $T.\mathsf{p}$ is set to $\mathsf{next}(T)(v)(\mathsf{p})$, for each parameter $\mathsf{p} \in \mathcal{M}(T)$.
4. If $C_T \neq \emptyset$, the unique maximal event of C_T is of the form $(\mathsf{end}(T), v)$ and $\mathsf{next}(T)(v) \uparrow$, then variable $\mathsf{loc}[T]$ is set to $\mathsf{L1}$, variable $\mathsf{done}[T]$ is set to true, and variable $T.\mathsf{p}$ is set to $v(\mathsf{p})$, for each parameter $\mathsf{p} \in \mathcal{M}(T)$.

The following claim directly implies that $\Re$ is injective.

Claim 6. For each reachable state s of network $\mathcal{N}(\mathcal{A})$, $\mathfrak{S}(\Re(s)) = s$.

Proof. Assume s is a reachable state of $\mathcal{N}(\mathcal{A})$. Let $C = \Re(s)$ and $s' = \mathfrak{S}(C)$. We must prove $s' = s$. Assume $T \in \mathcal{T}$. It suffices to prove, $s'.\mathsf{val}(T) = s.\mathsf{val}(T)$, $s'.\mathsf{loc}[T] = s'.\mathsf{loc}[T]$ and $s'.\mathsf{done}[T] = s'.\mathsf{done}[T]$. Let $A = \mathsf{start}(T)$ and $B = \mathsf{end}(T)$. We consider 5 cases:

1. $s.\mathsf{done}[T] = \mathsf{false}$ and $s.\mathsf{loc}[T] = \mathsf{L1}$ and $s.\mathsf{val}(T) = v^0_T$. Then, by Claim 1, $C = \emptyset$. Hence, also $C_T = \emptyset$. By definition of $\mathfrak{S}$, $s'.\mathsf{loc}[T] = \mathsf{L1}$, $s'.\mathsf{done}[T] = \mathsf{false}$ and $s'.\mathsf{val}[T] = v^0_T$. Thus, $s' = s$, as required.
2. $s.\mathsf{done}[T] = \mathsf{false}$ and $s.\mathsf{loc}[T] = \mathsf{L1}$ and $s.\mathsf{val}(T) \neq v^0_T$. Then $s.\mathsf{done}(A) = s.\mathsf{done}(B) = \mathsf{false}$, so

$$C_T = \{(A, v), (B, v) \mid v <_T s.\mathsf{val}(T)\}.$$

By Lemma 4, $v^0_T \leq_T s.\mathsf{val}(T)$. Hence, by assumption $s.\mathsf{val}(T) \neq v^0_T$, $v^0_T <_T s.Val(T)$. Thus $C_T \neq \emptyset$ and the unique maximal event of C_T is of the form (B, w) with $\mathsf{next}(T)(w) = s.\mathsf{val}(T))$. By definition of $\mathfrak{S}$, $s'.\mathsf{loc}[T] = \mathsf{L1}$, $s'.\mathsf{done}[T] = \mathsf{false}$ and $s'.\mathsf{val}[T] = s.\mathsf{val}[T]$. Thus, $s' = s$, as required.
3. $s.\mathsf{done}[T] = \mathsf{false}$, $s.\mathsf{loc}[T] = \mathsf{L2}$ and $\mathsf{next}(T)(s.\mathsf{val}(T)) \uparrow$. Then $s.\mathsf{done}(A) = \mathsf{true}$ and $s.\mathsf{done}(B) = \mathsf{false}$, so

$$C_T = \{(A, v) \mid v \leq_T s.\mathsf{val}(T)\} \cup \{(B, v) \mid v <_T s.\mathsf{val}(T)\}$$

Thus $C_T \neq \emptyset$ and the unique maximal event of C_T is $(A, s.\mathsf{val}(T))$. Hence, by definition of $\mathfrak{S}$, $s'.\mathsf{loc}[T] = \mathsf{L2}$, $s'.\mathsf{done}[T] = \mathsf{false}$ and $s'.\mathsf{val}[T] = s.\mathsf{val}[T]$. Thus, $s' = s$, as required.
4. $s.\mathsf{done}[T] = \mathsf{false}$, $s.\mathsf{loc}[T] = \mathsf{L2}$ and $\mathsf{next}(T)(s.\mathsf{val}(T)) \downarrow$. Then $s.\mathsf{done}(A) = s.\mathsf{done}(B) = \mathsf{false}$ and

$$C_T = \{(A, v) \mid v <_T \mathsf{next}(T)(s.\mathsf{val}(T))\} \cup \{(B, v) \mid v <_T s.\mathsf{val}(T)\}$$

Thus $C_T \neq \emptyset$ and the unique maximal event of C_T is $(A, s.\mathsf{val}(T))$. Hence, by definition of $\mathfrak{S}$, $s'.\mathsf{loc}[T] = \mathsf{L2}$, $s'.\mathsf{done}[T] = \mathsf{false}$ and $s'.\mathsf{val}[T] = s.\mathsf{val}[T]$. Thus, $s' = s$, as required.
5. $s.\mathsf{done}[T] = \mathsf{true}$. Then, by definition of $\Re$, $C_T = \{(A, v), (B, v) \mid v \leq_T s.\mathsf{val}(T)\}$. Hence $C_T \neq \emptyset$ and the unique maximal event of C_T is $(B, s.\mathsf{val}(T))$. By Lemma 4, $\mathsf{next}(T)(s.\mathsf{val}(T)) \uparrow$ and $s.\mathsf{loc}[T] = \mathsf{L1}$. By definition of $\mathfrak{S}$, $s'.\mathsf{loc}[T] = \mathsf{L1}$, $s'\mathsf{done}[T] = \mathsf{true}$ and $s'.\mathsf{val}[T] = s.\mathsf{val}[T]$. Thus $s' = s$, as required.

Claim 7. If s is reachable, $\Re(s) = C$, $C \overset{A}{\rightsquigarrow} C'$ and $s' = \mathfrak{S}(C')$ then $s \xrightarrow{A} s'$.

Proof. By Claim 6, $\mathfrak{S}(C) = s$. Let $\mathsf{task}(A) = T$. Since $C \overset{A}{\rightsquigarrow} C'$, $s.\mathsf{done}[T] = \mathsf{false}$. Hence, in order to prove that s enables an A-transition, it suffices to establish that $\mathsf{dep\_met}(A)$ holds in s. For this, in turn, it suffices to prove, for any incoming precedence edge $B \xrightarrow{u} A$ with $\mathsf{task}(B) \neq \mathsf{task}(A)$, that $\mathsf{dep\_met}(B, u, A)$ holds in s. Let $C' = C \cup \{\alpha\}$ with $\alpha = (A, v)$. Since $C \vdash \alpha$, all immediate predecessors of α are in C. Let $B \xrightarrow{u} A$ be a precedence edge of A and let $\mathsf{task}(B) = T'$. We consider the following cases:

- $C_{T'} = \emptyset$
  Since C contains all immediate predecessors of α, there exists no event (B, w) such that $U(B, A)(w) = v$. Since $C_{T'} = \emptyset$, then $s.\mathsf{done}[T'] = \mathsf{false}$ and $s.\mathsf{loc}[T'] = \mathsf{L1}$, it means that $s.\mathsf{done}(B) = \mathsf{false}$. Knowing that $B \xrightarrow{u} A$ and $s.\mathsf{done}(B) = \mathsf{false}$, the next event of type B, namely $\beta = (B, v^0_{T'})$ will occur in future. If $u(\mathsf{next}(B)) \downarrow$ and β is not an immediate predecessor of α, it follows that $v < u(\mathsf{next}(B))$, meaning that $\mathsf{dep\_met}(B, u, A)$ holds in s. If $u(\mathsf{next}(B)) \uparrow$, by the second case in the definition of $\mathsf{dep\_met}(B, u, A)$, $\mathsf{dep\_met}(B, u, A)$ is true in s.
- $C_{T'} \neq \emptyset$ and (B, w) is the unique maximal event of $C_{T'}$ of the form $(\mathsf{end}(T), w)$ with $\mathsf{next}(T')(w) \uparrow$. This implies that $s.\mathsf{done}[T'] = \mathsf{true}$ and that $s.\mathsf{done}(B)$ holds, therefore $\mathsf{dep\_met}(B, u, A)$ holds is s.
- $C_{T'} \neq \emptyset$ and (B, w) is the unique maximal event of $C_{T'}$ where $\mathsf{next}(T')(w) \downarrow$ and $u(s.\mathsf{next}(B)) \uparrow$. This means that the second condition in the definition of $\mathsf{dep\_met}(B, u, A)$ is true, meaning that $\mathsf{dep\_met}(B, u, A)$ holds is s.
- $C_{T'} \neq \emptyset$ and (B, w) is the unique maximal event of $C_{T'}$ where $\mathsf{next}(T')(w) \downarrow$ and $u(s.\mathsf{next}(B)) \downarrow$. Since $u(B, A)(w) = v$ it means that $u(s.\mathsf{next}(B)) \neq_T v$, and by **C4**, we have that $u(w) <_T u(s.\mathsf{next}(B))$, therefore $\mathsf{dep\_met}(B, u, A)$ holds in s.

We conclude that s enables an A-transition. Suppose $s \xrightarrow{A} s''$. Then, by Claim 5, $\Re(s) \overset{A}{\rightsquigarrow} \Re(s'')$. Since C has only one outgoing A-transition, $\Re(s'') = C'$. Hence, by Claim 6, $s'' = s'$, as required.

Claim 8. $\Re$ is a bijection from the reachable states of $\mathcal{N}(\mathcal{A})$ to $\mathsf{conf}(\mathcal{A})$.

Proof. Straightforward using Claims 1, 4, 6 and 7.

The theorem now follows by combination of the claims.

4 Real-Time Task Systems

In the previous section, we have introduced PPOs as a compact representation of task graphs with repetitive behavior. PPOs provide a convenient formalism to model embedded applications, but need to be incorporated in a larger formalism that supports modeling of the execution platform and of the mapping from applications onto this platform. In this section, we introduce such a formalism, which we name *real-time task systems (RTTS)*, and describe how the translation of PPOs to Uppaal from Section 3 can be lifted to RTTSs.

4.1 Definition of RTTS

In a real-time task system, the application is modeled using a PPO $\mathcal{A}$ together with a function w that gives the size of each task. The platform is modeled abstractly using a set $\mathcal{R}$ of resources and a function cap. Resources in $\mathcal{R}$ can be anything ranging from CPUs, memory, communication bandwidth and dedicated processing blocks, to devices such as scanners and printers. Each resource r has $\mathsf{cap}(r)$ units available, e.g., 3 CPUs, $133MB$ memory, and $10Mb/s$ bandwidth. The mapping from application to platform is specified using functions cl, h and ρ. Function cl specifies, for each task T and resource r, a bound of the number of units of r that can be allocated to T. In practice, resources are often handed over from one task to another, for instance when one task has created a file that is being procesed further by another task. This handover of resources is specified by function h. Finally, function ρ specifies the pace at which a task progresses, given the resources that have been allocated to it.

Formally, a *real-time task system (RTTS)* is a tuple $\mathcal{RTTS} = (\mathcal{A}, \mathcal{R}, \mathsf{cap}, \mathsf{cl}, \mathsf{h}, \mathsf{w}, \rho)$, where:[1]

- $\mathcal{A} = (\mathcal{T}, \mathcal{M}, E, U)$ is a PPO.
- $\mathcal{R}$ is a finite set of resources.
- $\mathsf{cap} : \mathcal{R} \to \mathbb{N}$ is a function that specifies for each resource the total number of units that is available.
- $\mathsf{cl} : \mathcal{T} \to (\mathcal{R} \to \mathbb{N})$ is a function that specifies, for each resource, the maximum number of units that a task may claim. A task may not claim more resources than what is available: for each $T \in \mathcal{T}$, $\mathsf{cl}(T) \leq \mathsf{cap}$.
- $\mathsf{h} : E \to (\mathcal{R} \to \mathbb{N})$ specifies the resources handed over from one task to another via edges of the PPO. We require that resources may be handed over only to start events: for all $A \in \mathcal{E}$ and $T \in \mathcal{T}$,
$$\mathsf{h}(A, \mathsf{end}(T)) = \mathbf{0},$$
where $\mathbf{0} : \mathcal{R} \to \mathbb{N}$ is given by $\mathbf{0}(r) = 0$, for $r \in \mathcal{R}$.
- $\mathsf{w} : \mathcal{T} \to \mathbb{N}$ is a function that specifies the *size* of each task, i.e., the amount of work that has to be done.
- $\rho : \mathcal{T} \times (\mathcal{R} \to \mathbb{N}) \to \mathbb{N}$ is a function that specifies the *pace* at which each task is processed, given the resources that have been assigned to it. We require that the pace increases monotonically with the number of resources available: for all $T \in \mathcal{T}$ and $a, a' \in \mathcal{R} \to \mathbb{N}$,
$$a \leq a' \Rightarrow \rho(T, a) \leq \rho(T, a').$$
However, the pace will not increase any further once the maximum number of resources that a task may claim has been allocated: $\rho(T, a) \leq \rho(T, \mathsf{cl}(T))$.

We call a resource r *static* if each task in the system may only progress when its maximum claim for resource r has been assigned. Formally, $r \in \mathcal{R}$ is *static* if, for all $T \in \mathcal{T}$ and for all $a \in \mathcal{R} \to \mathbb{N}$,
$$a \leq \mathsf{cl}(T) \wedge \rho(T, a) > 0 \Rightarrow a(r) = \mathsf{cl}(T)(r).$$

[1] In this section, we use functions of type $\mathcal{R} \to \mathsf{N}$, where $\mathcal{R}$ is some set of resources. Operations and predicates on N are extended to such function by pointwise extension. For instance, for $f, g : \mathcal{R} \to \mathsf{N}$, we say that $f \leq g$ iff $\forall r \in \mathcal{R} : f(r) \leq g(r)$, and we define $f + g : \mathcal{R} \to \mathsf{N}$ by $(f + g)(r) = f(r) + g(r)$.

Resources that are not static are called *dynamic*. A task that uses a dynamic resource may run faster if we assign more units of this resource to it. A typical example of a dynamic resource is communication bandwidth.

4.2 Semantics of RTTS

A *task instance* of PPO $\mathcal{A}$ is a pair $\beta = (T, v)$, where $T \in \mathcal{T}$ and $v \in V(T)$. We write $\mathsf{task}(\beta) = T$ and use $\mathsf{start}(\beta)$ and $\mathsf{end}(\beta)$ to denote the start and end event, respectively, that correspond to β. We let $\mathsf{ti}(\mathcal{A})$ denote the set of task instances of PPO $\mathcal{A}$. We say that a task instance β is *done* in configuration C if $\mathsf{end}(\beta) \in C$, we say that β is *active* in C if $\mathsf{start}(\beta) \in C$ and $\mathsf{end}(\beta) \notin C$, and we say that β is *waiting* in C if $\mathsf{start}(\beta) \notin C$. Clearly each task instance is either done, active or waiting, in any given configuration C. We define:

$$\begin{aligned}\mathsf{done}(C) &= \{\beta \in \mathsf{ti}(\mathcal{A}) \mid \mathsf{end}(\beta) \in C\},\\ \mathsf{active}(C) &= \{\beta \in \mathsf{ti}(\mathcal{A}) \mid \mathsf{start}(\beta) \in C \wedge \mathsf{end}(\beta) \notin C\},\\ \mathsf{waiting}(C) &= \{\beta \in \mathsf{ti}(\mathcal{A}) \mid \mathsf{start}(\beta) \notin C\}.\end{aligned}$$

A timed configuration records the precise global state of the modeled system at some point during execution, that is, the set of events that have occurred, the resources that have been allocated to each task instance, and the completion level of each task instance. Formally, a *timed configuration* of $\mathcal{RTTS}$ is a triple (C, O, θ) where:

- $C \in \mathsf{conf}(\mathcal{A})$.
- $O : \mathsf{ti}(\mathcal{A}) \to (\mathcal{R} \to \mathbb{N})$ specifies allocation of resources to task instances. We require that O does not allocate more resources than what is available in total: $\sum_{\gamma \in \mathsf{ti}(\mathcal{A})} O(\gamma) \leq \mathsf{cap}$. Moreover, we require, for each $\beta \in \mathsf{ti}(\mathcal{A})$ with $\mathsf{task}(\beta) = T$,
  1. O does not allocate more resources to β than T may claim: $O(\beta) \leq \mathsf{cl}(T)$.
  2. If β is waiting in C, the only resources allocated to β are those that have been handed over by preceding events: $\beta \in \mathsf{waiting}(C) \Rightarrow$
  $$O(\beta) = \sum_{\alpha \in C \mid \alpha \mapsto \mathsf{start}(\beta)} \mathsf{h}(\mathsf{ev\_type}(\alpha), \mathsf{start}(T))$$
  3. If β is active in C, enough resources are allocated to β for the handover to its successors upon termination: $\beta \in \mathsf{active}(C) \Rightarrow$
  $$O(\beta) \geq \sum_{\alpha \in \mathsf{ev}(\mathcal{A}) \mid \mathsf{end}(\beta) \mapsto \alpha} \mathsf{h}(\mathsf{end}(T), \mathsf{ev\_type}(\alpha))$$
  4. If β is done in C, no resources are allocated to it: $\beta \in \mathsf{done}(C) \Rightarrow$
  $$O(\beta) = \mathbf{0}$$

$$\frac{C \xrightarrow{\mathsf{start}(\beta)} C'}{(C, O, \theta) \xrightarrow{\mathsf{start}(\beta)} (C', O', \theta)}$$

$$\frac{\begin{array}{c} d \in \mathsf{R}_{\geq 0} \\ \forall \gamma \notin \mathsf{active}(C) : \theta'(\gamma) = \theta(\gamma) \\ \forall \gamma \in \mathsf{active}(C) : \theta'(\gamma) = \theta(\gamma) - d \cdot \rho(\mathsf{task}(\gamma), O(\gamma)) \end{array}}{(C, O, \theta) \xrightarrow{d} (C, O, \theta')}$$

$$\frac{C \xrightarrow{\mathsf{end}(\beta)} C'}{(C, O, \theta) \xrightarrow{\mathsf{end}(\beta)} (C', O', \theta)}$$

Fig. 3: Semantics of real-time task systems

- $\theta : \mathsf{ti}(\mathcal{A}) \to \mathbb{R}_{\geq 0}$ specifies for each task instance the amount of work that remains to be done. We require, for each $\beta \in \mathsf{ti}(\mathcal{A})$,
$$\begin{aligned} 0 \leq \theta(\beta) &\leq \mathsf{w}(\mathsf{task}(\beta)) \\ \beta \in \mathsf{done}(C) &\Rightarrow \theta(\beta) = 0 \\ \beta \in \mathsf{waiting}(C) &\Rightarrow \theta(\beta) = \mathsf{w}(\mathsf{task}(\beta)) \end{aligned}$$

The *timed configuration structure* of $\mathcal{RTTS}$ is the LTS $\mathcal{TC}(\mathcal{RTTS})$ with as states the timed configurations of $\mathcal{RTTS}$, as initial state the timed configuration $(\emptyset, O_0, \theta_0)$, where, for all β, $O_0(\beta) = \mathbf{0}$ and $\theta_0(\beta) = \mathsf{w}(\mathsf{task}(\beta))$, as actions the set $\mathsf{ev}(\mathcal{A}) \cup \mathbb{R}_{\geq 0}$, and a transition relation that is defined by the rules in Figure 3. These rules describe how $\mathcal{RTTS}$ may evolve from timed configuration (C, O, θ) to timed configuration (C', O', θ') due to the occurrence of an event or through passage of time.

Note that the rules of Figure 3 do not impose any relationship between the resource allocations before and after an event. In fact, a priori we allow for a complete reshuffling of the resource allocation whenever an event occurs. Also, there are no constraints on the time at which a new task instance starts. In practice, of course, we need to impose restrictions on timing and on how resource allocations may change. This is achieved using so-called *scheduling rules*, which we will discuss in the next subsection.

4.3 Scheduling rules

The timed configuration structures defined in the previous subsection are highly nondeterministic LTSs. After each event there may be an arbitrary, complete reallocation of resources between tasks. Also, the choice when to start a new task instance is entirely left open (there can be an arbitrary delay). We allow for delays and reallocation of resources in our semantics because this is what happens in the embedded systems that we model. However, in applications we will typically adopt a number of *scheduling rules* that severely reduce (or even eliminate)

the nondeterminism within a real-time task system, and impose constraints on the timing of events. Effectively, these rules cut away certain transitions and configurations in the semantics of the real-time task system that we have defined in Figure 3. Below we give two examples of commonly used, generic scheduling rules. Additional scheduling rules may be defined for each application.

1. *Nonlaziness*
   A task may be *lazy* or *nonlazy*. Semantically, this means that we remove all the time passage transitions $(C, O, \theta) \xrightarrow{d} (C', O', \theta')$, such that $d > 0$ and an event $\mathsf{start}(\beta)$ with $\mathsf{task}(\beta)$ nonlazy is enabled in (C, O, θ). In this way, we express that no time delay is allowed whenever an instance of a nonlazy task is enabled and sufficient resources are available: immediately either this or a competing task will start.
2. *Preemption*
   A task may be *preemptive* or *non-preemptive*. An instance of a preemptive task may be interrupted while running (that is, the pace may become 0 due to resources that are taken away), while this is not possible for any instance of a non-preemptive task. Semantically, we reduce the timed configuration structure by removing all configurations (C, O, θ) with $\rho(\mathsf{task}(\gamma), O(\gamma)) = 0$, for some event instance $\gamma \in \mathsf{active}(C)$ with $\mathsf{task}(\gamma)$ non-preemptive.

Example 3. Figures 4-6 depict, in a graphical manner, an RTTS that models a printer application. This case study will also be used in Section 6 for the framework performance evaluation. The underlying PPOs are depicted using the notational conventions explained already in Example 1. We assume that all tasks are nonlazy and nonpreemptive. The ovals encode resources and the parentheses contain their maximum capacity available (one if not mentioned). All the resources are static, except for USBup and USBdown. The dynamic resources USBup and USBdown obey the additional scheduling rule:

> **USB**: A timed configuration (C, O, θ) is only allowed if, when $O(\alpha)(\mathsf{USBup})$ and $O(\beta)(\mathsf{USBdown})$ are both positive, for some task instances α and β, then they are both equal to 3Mb/s. If only one is positive then it is equal to 4Mb/s.

The idea is that if two processes use the bus simultaneously, transmission takes place at a lower pace.

The dashed lines between resources and tasks indicate resource claims. Precedence edges are annotated with the number of resource units handed over. The total resource claim of a task can be obtained by taking the resource claims indicated by dotted lines plus all the resources that are handed over via incoming precedence edges of the start event minus all the resources that are handed over via outgoing precedence edges of the start event. Thus, for instance, task IP1 in Figure 4 claims 1 unit of resource IP1 and 24 units of resource RAM. Task

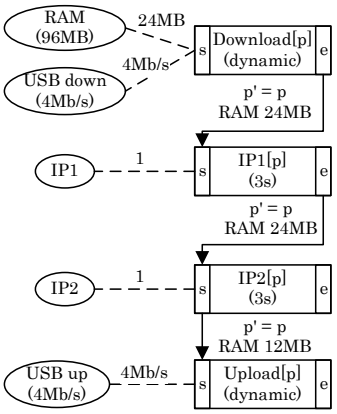


Fig. 4: Process from Store

Scan in Figure 6 claims 1 unit of resource Scanner and 0 units of resource RAM.

The rectangles denoting tasks may contain, between parentheses, task durations. If a task T is annotated with a task duration t seconds then this means that the total amount of work is t units and the pace is 1 unit per second if the task has all the resources that it claims, and 0 units per second otherwise. For tasks that use dynamic resource USBup, the duration is determined by the expression dynamic that is defined by:

$$\begin{aligned}\mathsf{dynamic} \equiv\ & \text{if USBup } = 4Mb/s \text{ then } 3s \text{ else} \\ & \text{if USBup } = 3Mb/s \text{ then } 4s \text{ else } 0.\end{aligned}$$

For tasks that use dynamic resource USBdown, the corresponding predicate is defined by:

$$\begin{aligned}\mathsf{dynamic} \equiv\ & \text{if USBdown } = 4Mb/s \text{ then } 3s \text{ else} \\ & \text{if USBdown} = 3Mb/s \text{ then } 4s \text{ else } 0.\end{aligned}$$

For these tasks, the total amount of work is $12Mb$ and the pace is either $4Mb/s$ or $3Mb/s$. The reader may verify that the timed configuration of this RTTS is a deterministic LTS. In particular, after each event the allocation of resources to task instances is uniquely determined.

5 Generated Uppaal models

We have implemented a tool that generates Uppaal models for a subclass of RTTSs. Our tool supports specification of lazy and nonlazy tasks, and of preemptive and nonpreemptive tasks. It accepts any RTTS $\mathcal{RTTS} = (\mathcal{A}, \mathcal{R}, \mathsf{cap}, \mathsf{cl}, \mathsf{h}, \mathsf{w}, \rho)$ in which:

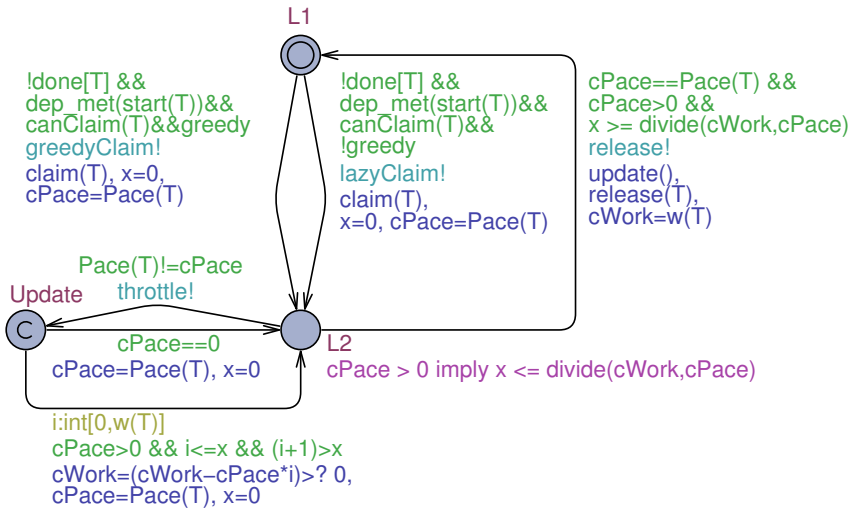


Fig. 7: Automaton for task T

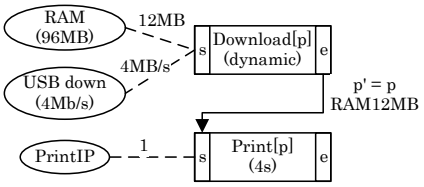


Fig. 5: Simple Print

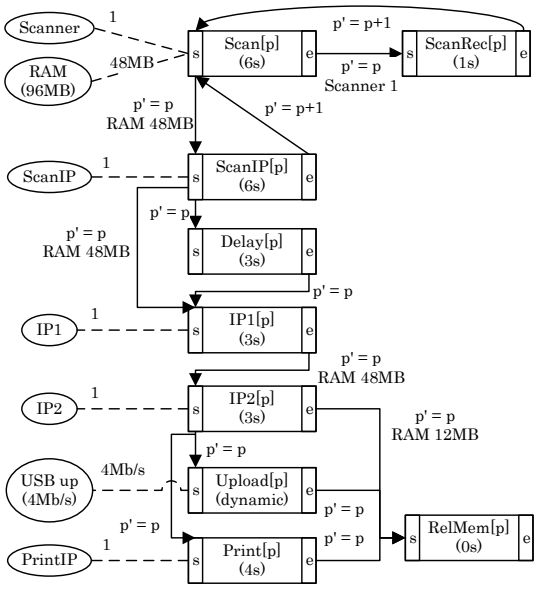


Fig. 6: Direct Copy

1. $\mathcal{A}$ is a restricted PPO such that, for each $T \in \mathcal{T}$, the only incoming edge of $\mathsf{end}(T)$ comes from $\mathsf{start}(T)$.
   This assumption allows us to obtain a translation of RTTSs to Uppaal by extending the translation of PPOs to Uppaal that we defined in Section 3.
2. Handover of resources from task T to task T' is only allowed if the precedence function for the corresponding PPO edge is bijective.
   This assumption simplifies the treatment of handovers, since whenever an instance of a task T is enabled, that is, all its immediate predecessors are done, the total number of resources that has been handed over to this task instance is always:
   $$\sum_{A \in \mathcal{E} \mid (A, \mathsf{start}(T)) \in E} \mathsf{h}(A, \mathsf{start}(T)).$$
3. Additional scheduling rules enforce that, for each timed configuration, the resource allocation is uniquely determined by the configuration of the PPO. Formally, we require that there exists a function $\mathsf{alloc} : \mathsf{conf}(\mathcal{A}) \to (\mathsf{ti}(\mathcal{A}) \to (\mathcal{R} \to \mathbb{N}))$ such that, for each timed configuration (C, O, θ), (C, O, θ) is allowed by the scheduling rules if and only if $O = \mathsf{alloc}(C)$. We require that $\mathsf{alloc}(\emptyset) = O_0$ and that, for each configuration C, $O = \mathsf{alloc}(C)$ satisfies the five conditions on O required for timed configurations.
   This assumption simplifies the translation since it eliminates the need to record, in each state, the exact allocation of resources to all the task instances.

It is easy to verify that the RTTS described in Example 3 satisfies the above conditions. In particular, the resource allocation O is uniquely determined by the configuration C of the PPO: (1) For a task instance β that is waiting in C, the only resources allocated are those that have been handed over by preceding task instances. (2) By definition of a timed configuration, no resources

can be allocated to a task instance β that is done in C. (3) If β is active in C, then for each static resource r, $O(\beta)(r) = \mathsf{cl}(\mathsf{task}(\beta))(r)$ (all tasks in the example are nonpreemptive), and for USBup and USBdown the resource allocation is uniquely determined by rule **USB**.

Let $\mathcal{RTTS}$ be an RTTS and alloc be a resource allocation function that satisfy the above restrictions. We define $\mathcal{N}(\mathcal{RTTS}, \mathsf{alloc})$ as the LTS induced by the parallel composition of the Uppaal timed automaton shown in Figure 7, for each $T \in \mathcal{T}$. Below we explain the various predicates and functions used in this timed automaton. The automaton uses the same global shared variables as the automaton for PPOs described in Section 3:

$$\{T.\mathsf{p}, \mathsf{loc}[T], \mathsf{done}[T] \mid T \in \mathcal{T} \wedge \mathsf{p} \in \mathcal{M}(T)\}.$$

As in Section 3, variable $T.\mathsf{p}$ gives the value of p in the next event of T that will occur. Variable $\mathsf{loc}[T]$ records whether the automaton is in location L1 or L2. Even though there is now an extra location Update in the automaton, the domain of $\mathsf{loc}[T]$ remains unchanged: during the brief excursions to location Update, variable $\mathsf{loc}[T]$ keeps value L2. As in Section 3, Boolean variable $\mathsf{done}[T]$ records whether the last event of T has been executed.

In addition, the timed automaton maintains three local variables: integer cWork records the latest estimate of the work that remains to be done, integer cPace records the current pace of task T, and clock x records the time that has elapsed since the start of the task or the last change of pace. Initially $\mathsf{cWork} = \mathsf{w}(T)$, $\mathsf{cPace} = 0$ and $\mathsf{x} = 0$. Finally, for each resource r, variable $\mathsf{resource\_cap}[r]$ records the number of units of resource r that are still available. Initially, $\mathsf{resource\_cap}[r] = \mathsf{cap}(r)$, for each r.

The automaton shown in Figure 7 has the same structure as the automaton of Figure 2 in Section 3 and uses the exact same functions dep_met and update. However, in order to handle resource allocation and timing, the automaton is equipped with some additional structure. Predicate $\mathsf{canClaim}(T)$ checks whether enough resources are available to start the next instance of task T:

$$\begin{aligned}\mathsf{canClaim}(T) = \rho(T, \mathsf{resource\_cap} &+ \sum_{A \in \mathcal{E} \mid (A, \mathsf{start}(T)) \in E} \mathsf{h}(A, \mathsf{start}(T)) \\ &- \sum_{B \in \mathcal{E} \mid (\mathsf{start}(T), B) \in E} \mathsf{h}(\mathsf{start}(T), B)) > 0\end{aligned}$$

Predicate $\mathsf{canClaim}(T)$ evaluates to true and the next instance of T may start if enough resources can be allocated such that, together with the resources that have been handed over make the pace positive.

Parameter greedy specifies if task T is nonlazy (true) or lazy (false). The automaton has two transitions from location L1 to L2. If task T is nonlazy we take the transition labeled with urgent broadcast channel greedyClaim, whereas if task T is lazy we take the transition labeled with (nonurgent) broadcast channel lazyClaim. In this way we ensure that a nonlazy task starts as soon as it becomes enabled, whereas the start of a lazy task may be postponed.

Function $\mathsf{claim}(T)$ first sets location variable $\mathsf{loc}[T]$ to L2. Let s be the resulting global state of the Uppaal model. We may then compute the corresponding PPO-configuration by applying[2] the function $\Re$ introduced in the proof of Theorem 1: $C = \Re(s)$. Next, we may compute the resource allocation function associated to C: $O = \mathsf{alloc}(C)$. Once we know which resources have been assigned, function claim computes which resources are still available in the new state and updates the value of resource_cap:

$$\mathsf{resource\_cap} = \mathsf{cap} - (\sum_{\gamma \in \mathsf{ti}(\mathcal{A})} O(\gamma))$$

Function $\mathsf{Pace}(T)$ computes the pace of the task instance of T that is currently active. If s is the current global state of the model then

$$\mathsf{Pace}(T) = \rho(T, \mathsf{alloc}(\Re(s))(T, s.\mathsf{val}(T)))$$

Whenever the automaton starts a new instance of task T by jumping from location L1 to location L2, it resets clock x and sets variable cPace to $\mathsf{Pace}(T)$. Assuming the pace remains unchanged, the time needed to complete the task instance is $\frac{\mathsf{cWork}}{\mathsf{cPace}}$. Since in Uppaal we may only compare clocks to integers, we sometimes need to slightly overapproximate the task duration: the automaton may leave location L2 when $\mathsf{x} == \mathsf{divide}(\mathsf{cWork}, \mathsf{cPace})$. Here function divide divides its two arguments and takes the ceiling.

At any point during execution of an instance of task T, due to the start or completion of some other task instance, the allocation of resources and hence the pace of T may change. Whenever this happens, the automaton for task T instantly jumps to location Update via a transition labelled with urgent broadcast channel throttle. From location Update the automaton instantly jumps back to location L2.[3] Since Update is committed, no other automaton may perform a transition in between. There are two cases. If task T was preempted ($\mathsf{cPace} == 0$), the amount of work to be done remains unchanged: the automaton sets cPace to the new value, resets clock x, and jumps to L2. If task T was running ($\mathsf{cPace} > 0$) then the remaining amount of work is $\mathsf{cWork} - \mathsf{cPace} * \mathsf{x}$. Since Uppaal does not permit the use of clock variables in integer expressions, we (conservatively) approximate this value. Using a select statement, we pick the largest integer i that does not exceed x and decrement cWork with $\mathsf{cPace} * \mathsf{i}$. In addition, we set cPace to the new value, reset clock x, and jump to L2.

[2] Formally, function $\Re$ takes as input global states of the Uppaal model described in Section 3. We may apply $\Re$ to global states of the extended model described in this section by removing all the additional state variables.

[3] The intermediate location Update is required since in Uppaal clock guards are not allowed on edges labeled with urgent channels.

A task instance may end and the automaton may jump back from L2 to L1 when the work has been completed: since we assume that in the PPO the only incoming edge of $\mathsf{end}(T)$ comes from $\mathsf{start}(T)$, we do not need to check anymore if $\mathsf{dep\_met}(\mathsf{end}(T))$ holds. On the transition from L2 to L1, function update() (introduced in Section 3) sets $\mathsf{done}[T]$ to true if the last event for task T has occurred and otherwise updates the parameters of T. Variable cWork is reinitialized to $\mathsf{w}(T)$, and function $\mathsf{release}(T)$ sets $\mathsf{loc}[T]$ to L1, computes which resources are still available in the new state and updates the value of resource_cap accordingly, in exactly the same way as function $\mathsf{claim}(T)$.

Of course, we would like to prove that, for any $\mathcal{RTTS}$ and resource allocation alloc that satisfies our constraints, the LTS $\mathcal{N}(\mathcal{RTTS}, \mathsf{alloc})$ induced by our Uppaal translation is behaviorally equivalent with the timed configuration structure $\mathcal{TC}(\mathcal{RTTS})$, pruned according to function alloc. Unfortunately, due to the throttle transitions in the Uppaal model and the overapproximations which they induce, there is no exact correspondence. Previous experiments with this type of approximations [23,24] suggest that in practical applications (from the printing domain) the errors that they introduce are minimal, but it remains to be investigated if (and how) these observations can be formalized using some notion of schedule robustness.

Below we establish that in the special case in which all tasks are non-preemptive and all resources are static, and moreover work divided by pace is always an integer, there is an exact correspondence between the semantics of an RTTS and the semantics of its Uppaal translation. It is not possible to prove that the two structures are isomorphic: from a timed configuration we can not infer a unique value for the local clock of a task automaton in location L1. Instead, we establish a bisimulation between the two semantic structures.

Theorem 2. *Let $\mathcal{RTTS}$ be a real-time task system, with all tasks non-preemptive and all resources static, let* alloc *be a function that maps configurations to resource allocations, and assume $\mathcal{RTTS}$ and* alloc *satisfy the constraints listed at the beginning of this section. Assume further that, for each task T and resource allocation a with $\rho(T,a) > 0$, $\mathsf{w}(T)/\rho(T,a)$ is an integer. Then the LTS $\mathcal{TC}(\mathcal{RTTS})$, pruned according to* alloc*, and the LTS $\mathcal{N}(\mathcal{RTTS}, \mathsf{alloc})$ are bisimilar.*

Proof. Since all tasks of $\mathcal{RTTS}$ are nonpreemtive, an active task instance always has a positive pace. Moreover, since all tasks of $\mathcal{RTTS}$ are static,

$$a \leq \mathsf{cl}(T) \wedge \rho(T,a) > 0 \Rightarrow a = \mathsf{cl}(T).$$

This means that the pace of an active instance of task T is constant and always equal to $\rho(T, \mathsf{cl}(T))$. Hence, in a run of the Uppaal model location, the automaton for any task T will never reach location Update. Hence, if s is a reachable state of the Uppaal semantics with $s.\mathsf{loc}[T] = \mathsf{L2}$, then $s.T.\mathsf{cPace} = \rho(T, \mathsf{cl}(T))$ and $s.T.\mathsf{cWork} = \mathsf{w}(T)$.

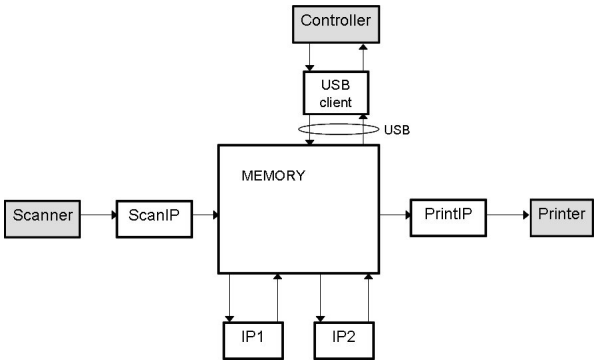


Fig. 8: Océ printer architecture

Let s be a reachable state of the Uppaal semantics. We define $f(s)$ to be the timed configuration (C, O, θ) where $C = \Re(s)$, $O = \mathsf{alloc}(C)$ and, for each $\beta \in \mathsf{ti}(\mathcal{A})$ with $\mathsf{task}(\beta) = T$,

$$\theta(\beta) = \begin{cases} \mathsf{w}(T) & \text{if } \beta \in \mathsf{waiting}(C) \\ \mathsf{w}(T) - s.T.\mathsf{x} \cdot \rho(T, \mathsf{cl}(T)) & \text{if } \beta \in \mathsf{active}(C) \\ 0 & \text{if } \beta \in \mathsf{done}(C) \end{cases}$$

Observe that (C, O, θ) is a timed configuration since by Claim 4 in the proof of Theorem 1, $C = \Re(s)$ is a configuration; by the assumption about alloc, $O = \mathsf{alloc}(C)$ satisfies the 5 conditions for O required for timed configuration, and by construction θ also satisfies the conditions required for a timed configuration. Since $O = \mathsf{alloc}(C)$, the timed configuration is allowed by the scheduling rules.

It is routine to check that the relation R that relates s and $f(s)$, for each reachable state of the Uppaal semantics, is a bisimulation relation.

6 Experiments

We now turn to an experimental evaluation of Uppaal models generated from the RTTS representation. We compare these models with handcrafted models that have been presented in [23]. In the handcrafted models, an application is represented as a collection of use cases, each use case being modeled as a single automaton that contains all tasks. This way of modeling is more natural for design engineers but less efficient to analyze as proven below. The case study used for comparison is about Océ printing systems. The printer architecture studied here is depicted in Figure 8 with the RTTS of Example 3. We computed for each experiment the fastest time in which all tasks were completed (also called makespan) assuming that all tasks were nonlazy. All experiments were performed with Uppaal, version 4.1.2, on a Sun Fire X4440 server with 16 cores (AMD Opteron 8356, 2.3GHz) and 128 Gb of DDR2 RAM.

Table 1: Direct copy(dc) || Simple print(sp) Case - Comparison Handcrafted Models(grey) vs. Generated Models (O.M. - out of memory)

#dc	#sp	Mem (KB)	Time (s)	Make-span(s)	States Explored
2	3	4500	0.50	23	1130
		5124	0.40	23	413
7	10	5480	1.60	71	10578
		5408	1.10	71	3050
35	50	12808	11.31	367	149926
		9184	13.51	367	48196
70	100	26568	27.92	737	433816
		18016	43.74	737	155491
334	500	598996	279.70	3585	6843592
		282220	898.98	3585	3038099
667	1000	2321768	1304.87	7166	25206064
		1076196	3702.10	7166	11704000
903	1355	4165896	1937.88	9705	45225661
		1962636	7165.40	9705	21272017
904	1356	O.M.	O.M.	O.M.	O.M.
		1964952	7173.70	9715	21302397
1460	1960	O.M.	O.M.	O.M.	O.M.
		4053164	17055.36	15117	44117751

Table 2: Direct copy(dc) || Process from store (pfs) Case - Comparison Handcrafted Models(grey) vs. Generated Models (O.M. - out of memory)

#dc	#pfs	Mem (KB)	Time (s)	Make-span(s)	States Explored
1	2	4456	0.40	15	704
		5516	0.40	15	411
10	20	7540	4.10	114	47551
		6936	6.41	114	20118
25	50	21352	19.51	279	334606
		13984	40.14	279	135453
120	240	586172	384.66	1324	8255421
		244260	991.61	1324	3269408
240	480	2555392	1857.78	2644	33327861
		1007540	4245.12	2644	13162088
303	606	4077452	2419.45	3337	53223828
		1572420	7053.88	3337	21007415
304	608	O.M.	O.M.	O.M.	O.M.
		1582848	7157.23	3348	21146664
480	960	O.M.	O.M.	O.M.	O.M.
		4056016	20827.78	5284	52819448

Three performance metrics were used to evaluate each experiment: the peak memory usage (column 'Mem') and running time (column 'Time') of Uppaal, and the total number of states explored. Table 1 gives the comparison results for PPO Direct Copy in parallel with the Simple Print and Table 2 shows the Direct Copy case in parallel with Process from Store. The first two columns in each table give the total number of task instances processed for each PPO.

To combat state space explosion, we applied the sweep line method of Uppaal [34]. As a consequence, each model was annotated with progress measures (i.e. parameter valuation) that Uppaal used during the analysis to store only the states where the progress measures were weakly monotonically increasing, or occasionally decreasing.

The state space obtained from the generated models is between 41% and 71% smaller than the one obtained from the handcrafted models. Therefore, the state space explosion problem emerges later in the analysis of the RTTS-based models, and we could analyze a higher number of task instances.

There are two causes that lead to the large difference in the sizes of the state spaces generated. Firstly, in handcrafted modeling approach each resource was modeled by a separate automaton that comprises three states: Idle, Running and one to model a recovery phase that some resources like Scanner require. The first two states corresponded to the L1 and L2 in the task automaton. However, the extra state was not needed in the generated models, the recovery phase being modeled as a separate task. The other cause comes from the tasks that claim more than one resource. In the generated models, one could easily model this multi-resource claim by checking the number of resource units available (see Figure 7). By contrast, in the handcrafted models, multi-resource claim was modeled with the help of third party automata placed between the RTTS and resource automata, an extra third party automaton being added for each resource claimed. This extra automaton registered the claim to the resource automaton then it waited for the resource automaton to become available. When it was available, it sent the request to the resource automaton. On completion of the processing, it sent an end event to the RTTS automaton.

Tables 1 and 2 also show up to a 61% decrease in the peak memory used by Uppaal during the analysis. However, analysis of the generated models required more time. This was the price to pay for the parametric representation of these models, where a lot of details were encoded into functions. The evaluation of some of these functions (e.g. dep_met) required a lot of time due to the conditions or function calls that they contain.

7 Conclusions

PPOs are a simple extension of partial orders, but expressive enough to compactly represent large task graphs with finite repetitive behavior. We presented a translation from a subclass of PPOs to Uppaal, together with a correctness proof that the transition system induced by a Uppaal model is isomorphic to the configuration structure of a PPO.

This paper introduces real-time task systems (RTTS), a general model for real-time embedded systems that uses PPOs for modeling applications. A distinguishing

feature of RTTSs is the ability to express that the pace of a task depends on the number of resources that have been allocated to it. We described a translation to Uppaal for a significant subclass of RTTSs.

Finally, we presented experiments which demonstrate that the Uppaal models obtained through our translation are more tractable than handcrafted models of the same systems used in earlier case studies.

As explained in this paper, when the applications (use cases) of a real-time embedded system design are described using PPOs, then we have a well-defined partial order structure on the corresponding events. Due to competition for resources and timing constraints, only a fragment of all the interleavings of this partial order will be possible in the full system model. Nevertheless, it will be interesting to see if partial order reduction techniques [35,36] will allow us to exploit the inherent structure of PPOs to alleviate the state space explosion problem when analyzing the full system model.

Another interesting topic for future research is to adapt the results of [5] to the PPO settings. This approach reduces the complexity of scheduling problems by exploiting the finite repetitive structure of tasks: it reduces a scheduling problem to a problem containing a minimal number of identical repetitions, and after solving this smaller problem, the computed schedule is expanded to a schedule for the original, more complex scheduling problem.

In addition, we plan to extend the translation with more scheduling rules that we have encountered while modeling the behavior of practical systems.

The experiments that we described in Section 6 of this paper, like the design-space exploration algorithms that have currently been implemented in the Octopus toolset, are entirely non-parametric: only after the values of all timing parameters, CPU speeds, buffer sizes, etc have been fixed, we may compute the reachable states of our model using Uppaal and compute performance metrics such as the makespan. It would be very interesting to explore the possibility of a parametric analysis in order to speed up the design-space exploration, along the lines that have been explored in [37,38].

*Acknowledgment*An extended abstract of this paper appeared as [1]. This research has been carried out as part of the OCTOPUS project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program. This research was also supported by European Community's FP7 Programme under grant agreement no 214755 (QUASIMODO). We thank Twan Basten, Alexandre David, Martijn Hendriks, Nikola Trčka, Marc Voorhoeve, for inspiring discussions on the topic of this paper. We dedicate this paper to the memory of Marc Voorhoeve, 1950 - 2011, who devised the notion of a PPO.

References

1. F. Houben, G. Igna, and F. Vaandrager, "Modeling task systems using parameterized partial orders," in *Proceedings 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2012),* Beijing China, April 16-19, 2012, M. Di Natale, Ed. IEEE Computer Society, 2012, pp. 317–327.
2. F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-software co-design of embedded systems: the POLIS approach.* Kluwer Academic Press, 1997.
3. B. Kienhuis, E. F. Deprettere, K. A. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *ASAP*. IEEE Computer Society, 1997, pp. 338–349.
4. N. v. d. Nieuwelaar, J. v. d. Mortel-Fronczak, and J. Rooda, "Design of supervisory machine control," in *European Control Conference*, 2003.
5. M. Hendriks, B. van den Nieuwelaar, and F. Vaandrager, "Recognizing finite repetitive scheduling patterns in manufacturing systems," in *MISTA 2003.* The University of Nottingham, 2003, pp. 291–319.
6. A. W. Mazurkiewicz, "Compositional semantics of pure place/ transition systems," in *European Workshop on Applications and Theory of Petri Nets*, ser. Lecture Notes in Computer Science. Springer, 1987, vol. 340, pp. 307–330.
7. V. Pratt, "Modeling concurrency with partial orders," *International Journal of Parallel Programming*, vol. 15, pp. 33–71, 1986.
8. G. Winskel, "An introduction to event structures," in *REX Workshop*, ser. Lecture Notes in Computer Science, vol. 354. Springer, 1988, pp. 364–397.
9. K. Jensen, L. M. Kristensen, and L. Wells, "Coloured petri nets and cpn tools for modelling and validation of concurrent systems," *STTT*, vol. 9, no. 3-4, pp. 213–254, 2007.
10. K. Jensen and L. M. Kristensen, *Coloured Petri Nets - Modelling and Validation of Concurrent Systems.* Springer, 2009.
11. Octopus toolset homepage, http://dse.esi.nl, 2011.
12. T. Basten, E. van Benthum, M. Geilen, M. Hendriks, F. Houben, G. Igna, F. Reckers, S. de Smet, L. J. Somers, E. Teeselink, N. Trčka, F. Vaandrager, J. Verriet, M. Voorhoeve, and Y. Yang, "Model-driven design-space exploration for embedded systems: The octopus toolset," in *ISoLA (1)*, ser. Lecture Notes in Computer Science, vol. 6415. Springer, 2010, pp. 90–105.
13. S. Stuijk, M. Geilen, and T. Basten, "Sdf^3: Sdf for free," in *ACSD.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 276–278.
14. G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *SFM*, ser. Lecture Notes in Computer Science, vol. 3185. Springer, 2004, pp. 200–236.
15. N. Trčka, M. Voorhoeve, and T. Basten, "Parameterized partial orders for modeling embedded system use cases: Formal definition and translation to coloured petri nets," in *ACSD*, june 2011, pp. 13 –18.
16. R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

17. Y. Abdeddaïm, A. Kerbaa, and O. Maler, "Task graph scheduling using timed automata," in *IPDPS*. IEEE Computer Society, 2003, p. 237.
18. M. Mikucionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, J. S. Pedersen, and P. Hougaard, "Schedulability analysis using uppaal: Herschel-planck case study," in *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part II*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 6416. Springer, 2010, pp. 175–190.
19. M. Hendriks and M. Verhoef, "Timed automata based analysis of embedded system architectures," in *IPDPS*. IEEE, 2006.
20. S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour, "Influence of different abstractions on the performance analysis of distributed hard real-time systems," *Design Autom. for Emb. Sys.*, vol. 13, no. 1-2, pp. 27–49, 2009.
21. J. Berendsen, B. Gebremichael, F. W. Vaandrager, and M. Zhang, "Formal specification and analysis of zeroconf using uppaals," *ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 3, p. 34, 2011.
22. F. Cassez, J. J. Jessen, K. G. Larsen, J.-F. Raskin, and P.-A. Reynier, "Automatic synthesis of robust and optimal controllers - an industrial case study," in *HSCC*, ser. Lecture Notes in Computer Science, vol. 5469. Springer, 2009, pp. 90–104.
23. G. Igna, V. Kannan, Y. Yang, T. Basten, M. Geilen, F. W. Vaandrager, M. Voorhoeve, S. de Smet, and L. J. Somers, "Formal modeling and scheduling of datapaths of digital document printers," in *FORMATS*, ser. Lecture Notes in Computer Science, vol. 5215. Springer, 2008, pp. 170–187.
24. G. Igna and F. W. Vaandrager, "Verification of printer datapaths using timed automata," in *ISoLA (2)*, ser. Lecture Notes in Computer Science, vol. 6416. Springer, 2010, pp. 412–423.
25. I. AlAttili, F. Houben, G. Igna, S. Michels, F. Zhu, and F. W. Vaandrager, "Adaptive scheduling of data paths using uppaal tiga," in *QFM*, ser. EPTCS, vol. 13, 2009, pp. 1–11.
26. C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.
27. A. K. Mok, A. X. Feng, and D. Chen, "Resource partition for real-time systems," in *IEEE Real Time Technology and Applications Symposium*. IEEE Computer Society, 2001, pp. 75–84.
28. S. Chakraborty, S. Künzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," in *DATE*. IEEE Computer Society, 2003, pp. 10 190–10 195.
29. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the SymTA/S approach," *IEE Proceedings Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, 2005.
30. C. Norström, A. Wall, and W. Yi, "Timed automata as task models for event-driven systems," in *RTCSA*. IEEE Computer Society, 1999, pp. 182–189.
31. P. Krcál, M. Stigge, and W. Yi, "Multi-processor schedulability analysis of preemptive real-time tasks with variable execution times," in *FORMATS*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4763, pp. 274–289.
32. E. Fersman, P. Krcál, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *Inf. Comput.*, vol. 205, no. 8, pp. 1149–1172, 2007.
33. R. J. van Glabbeek and G. D. Plotkin, "Configuration structures, event structures and petri nets," *CoRR*, vol. abs/0912.4023, 2009.
34. S. Christensen, L. M. Kristensen, and T. Mailund, "A sweep-line method for state space exploration," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 2031. Springer, 2001, pp. 450–464.
35. D. Peled, "Ten years of partial order reduction," in *CAV*, ser. Lecture Notes in Computer Science, vol. 1427. Springer, 1998, pp. 17–28.
36. K. L. McMillan, "Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits," in *CAV*, ser. Lecture Notes in Computer Science, vol. 663. Springer, 1992, pp. 164–177.
37. A. Cimatti, L. Palopoli, and Y. Ramadian, "Symbolic computation of schedulability regions using parametric timed automata," in *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*. IEEE Computer Society, 2008, pp. 80–89.
38. A. Simalatsar, Y. Ramadian, K. Lampka, S. Perathoner, R. Passerone, and L. Thiele, "Enabling parametric feasibility analysis in real-time calculus driven performance evaluation," in *Proceedings of the 14th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, R. K. Gupta and V. J. Mooney, Eds. ACM, 2011, pp. 155–164.