Modeling Task Systems Using Parameterized Partial Orders

Fred Houben
ASML
Veldhoven, The Netherlands
Email: fred.houben@asml.com

Georgeta Igna
ICIS, MBSD group
Radboud University Nijmegen
Nijmegen, The Netherlands
Email: g.igna@cs.ru.nl

Frits Vaandrager
ICIS, MBSD group
Radboud University Nijmegen
Nijmegen, The Netherlands
Email: f.vaandrager@cs.ru.nl

***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 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 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.**

***Keywords*-parameterized partial order; concurrent systems; timed automata**

I. 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 [1], [2]. 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.

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 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, repetitive behavior, task graphs tend to become very large and no longer practical for specification and analysis of application behavior. Following [3], [4], 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 [5] traces, pomsets (partially-ordered multisets) [6], and event structures [7], 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 [8], [9], 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 [10] that aims to leverage existing modeling, analysis, and DSE tools to support model-driven DSE for real-time embedded systems [11]. 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 toolset integrates CPN Tools [8], [9] 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 toolset has introduced the notion of parametrized partial orders [12]. 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. A translation of PPOs to CPN Tools has recently been described in [12]. 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 [15]. It has been successfully used in many domains, e.g. for finding optimal solutions for scheduling problems [16], performance analysis of real-time distributed systems [17], [18], protocol verification [19] and controller synthesis [20]. 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 [21], [22], [23]. 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. This brings us to the two main results of this paper: (a) 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), and (b) 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 [21].

The structure of this paper is as follows. The next section recalls some preliminary definitions regarding labeled transition systems, the underlying semantic notion used throughout the paper. Section III 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 IV explains how this translation is generalized to general embedded system designs, which besides an application (modeled as a PPO) also involve a platform and a mapping. Section V presents performance evaluation results of models generated by comparing them with handcrafted Uppaal models presented before in our papers[1]. Concluding remarks and future work follow in Section VI.

II. 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.

Two labeled transition systems $\mathcal{L}_1 = (S_1, s_0^1, \Sigma_1, \rightarrow)$ and $\mathcal{L}_2 = (S_2, s_0^2, \Sigma_2, \rightarrow)$ 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
- $s \xrightarrow{a} s' \Leftrightarrow f(s) \xrightarrow{a} f(s')$, for all $s, s' \in S_1$, $a \in \Sigma_1$.

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 $\rightarrow' = \{(s, a, s') \mid s, s' \in S' \wedge s \xrightarrow{a} s'\}$.

III. Parameterized Partial Orders

A *parametrized partial order (PPO)* is a partial order that comes equipped with some extra structure to capture repetitive behavior. In [12], a PPO is defined at task level and assumes a precedence relation between tasks. In this

[1]The models generated for Section V are available at http://www.mbsd.cs.ru.nl/publications/papers/fvaan/HIV12.

paper, 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.

A. 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.

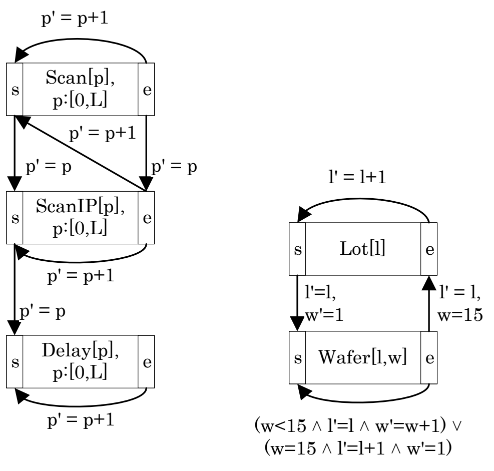
Formally, we assume a universe $\mathcal{P}$ of typed variables called *parameters*. A *valuation* of a set $P \subseteq \mathcal{P}$ of parameters is a function that maps each parameter in P to an element of its domain. We assume that the domain of each parameter is a nonempty set. We write $V(P)$ for 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 $\mathsf{task} : \mathcal{E} \to \mathcal{T}$ and $\mathsf{type} : \mathcal{E} \to \{\mathsf{s}, \mathsf{e}\}$ are given by $\mathsf{task}((t, T)) = T$ and $\mathsf{type}((t, T)) = t$, and embeddings $\mathsf{start} : \mathcal{T} \to \mathcal{E}$ and $\mathsf{end} : \mathcal{T} \to \mathcal{E}$ 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 $A \overset{u}{\to} B$ 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 [21]). 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 $[0, \dots, L]$ representing the number of the current page processed. The constant $L \in \mathbb{N}$ is a bound for the parameter p. A precedence function $A \overset{u}{\to} B$ 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



(a) Printer (b) Wafer production

Figure 1: PPO representation

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 [4]. After the start of each lot, 15 wafer tasks are executed in sequence, followed by the end of the lot.

B. 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 [7], [24]). 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 $(B, A) \in E \land U(B, A)(w) = v$.

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 $C' = C \cup \{\alpha\}$.

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 β are in $\mathsf{rev}(\mathcal{A})$, we write $\alpha \leq_{\mathcal{A}} \beta$, if for each configuration $C \in \mathsf{conf}(\mathcal{A})$, $\beta \in C$ implies $\alpha \in C$. The technical lemma below, whose simple proof we omit, states that the $\leq_{\mathcal{A}}$ contains the immediate predecessor relation:

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

The following lemma, which is again easy to prove, 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 $\mathsf{rev}(\mathcal{A})$.

C. 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 $A \overset{u}{\rightarrow} B$ 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 one from the start event to the end event, and the one from the end event to the start event:
$$\mathsf{task}(A) = \mathsf{task}(B) \quad \Rightarrow \quad ((A, B) \in E \Leftrightarrow A \neq B)$$
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_T^0.
- **C2**: $\mathsf{next}(T)$ is injective
- **C3**: u is only defined for reachable valuations:
$$u(v)\downarrow \quad \Rightarrow \quad v_T^0 \leq_T v$$
- **C4**: u is *monotonic*:
$$v \leq_T w \wedge u(w)\downarrow \quad \Rightarrow \quad u(v)\downarrow \wedge\, u(v) \leq_{T'} u(w)$$

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 PPOs of Examples 1 and 2 are restricted.

Lemma 3: Let $\mathcal{A}$ be a restricted PPO with 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 ordering on the set $\{\alpha \in \mathsf{rev}(\mathcal{A}) \mid \mathsf{task}(\alpha) = T\}$ of reachable events of T.

D. 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 the Uppaal 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 L1 or L2. Boolean

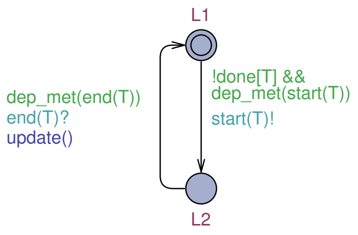


Figure 2: Automaton for task T

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 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:

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

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:

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

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,

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

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

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

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

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

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 was 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

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

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 $v^0_{T'} \leq_{T'} s.\mathsf{val}(T')$. Using the second invariant of Lemma 4, we infer that $\mathsf{next}(T')(s.\mathsf{val}(T')) \uparrow$. Condition **C3** implies that $v^0_{T'} \leq_{T'} w$. 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 $v^0_{T'} \leq_{T'} w$, and Lemma 4 implies $v^0_{T'} \leq_{T'} s.\mathsf{next}(B)$. 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 $s'.\mathsf{val}(T) = s.\mathsf{val}(T)$. Thus $\mathsf{next}(T)(s'.\mathsf{val}(T)) \uparrow$ and $s'.\mathsf{done}(A)$. 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 $s'.\mathsf{val}(T) = s.\mathsf{val}(T)$. Thus $\mathsf{next}(T)(s'.\mathsf{val}(T)) \downarrow$, $s'.\mathsf{done}(A) = \mathsf{false}$, 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
$$C_T = \{(A, v), (B, v) \mid v \leq_T s.\mathsf{val}(T)\}.$$
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
$$\begin{aligned} C_T &= \{(A, v) \mid v <_T \mathsf{next}(T)(s.\mathsf{val}(T))\} \\ &\cup \{(B, v) \mid v <_T s.\mathsf{val}(T)\} \end{aligned}$$
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.

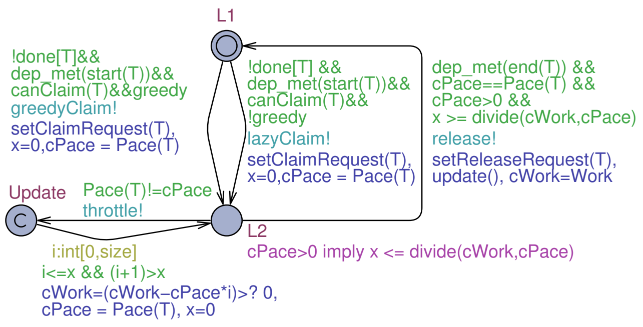


Figure 3: Task Template

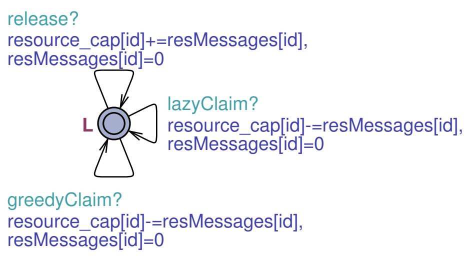

release?
resource_cap[id]+=resMessages[id],
resMessages[id]=0
L
lazyClaim?
resource_cap[id]-=resMessages[id],
resMessages[id]=0
greedyClaim?
resource_cap[id]-=resMessages[id],
resMessages[id]=0

Figure 4: Resource template

- $C_{T'} \neq \emptyset$ 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. ■

IV. Generated Uppaal models

As mentioned in the introduction, we have developed a toolset for exploring embedded system designs, that is centered around an intermediate, Y-chart based representation.

In this representation, a system is described as a combination of three modules: applications, platform and one of their possible mappings. Applications are described as PPOs and the platform as a collection of resources. Each resource is characterized by two parameters: total capacity and pace per time unit. The latter parameter might change during task execution e.g. the pace of a bus depends on the number of tasks that use the bus at some point in time. In this pattern, the mapping module contains details about the number of resources that some tasks claim and release at the beginning and at the end of their execution, respectively.

The generated Uppaal models have a simple structure: each task and resource that appears in the Y-chart representation instantiates a task template or a resource template, respectively, that we detail below.

The task template shown in Figure 3 is an extended version of the untimed task template of Figure 2. The template is enriched with timing and resource constraints like a variable Work for the total amount of work that should be processed and a function Pace(T) which returns the pace at which the data is being processed, that depends on the pace of the resources claimed.

Formally, the templates of Figures 2 and 3 can be related through the notion of a timed step simulation introduced in [25]. Since timed step simulations are compositional [25], we can associate to any reachable state of our timed model a configuration of the PPO that represents the application part of it. Note however that, due to the imposed timing constraints, certain configurations of a PPO cannot be reached in the timed setting.

In Figure 3, the transitions from L1 to L2 encode a start event, and the reverse transition encodes an end event. A start event can occur if variable done evaluates to false, meaning that the last start event has not occurred yet, the dependencies induced by the event precedence functions are satisfied (the dep_met function), and all the resources claimed have enough capacity available to process the task (the canClaim condition). Tasks are scheduled using either a greedy or a lazy policy. If a resource changes its pace, then a throttle channel is urgently enabled in each task automaton that uses the resource at that moment. On the transition between Update and L2 locations, the remaining amount of work is computed (parameter cWork) and also the task pace (variable cPace) is updated. For these updates, a select statement is used in order to under-approximate the time elapsed from the latest pace modification recorded by the clock x (of real type) to the closest integer (variable i) below this value. This approximation is necessary because clock variables cannot be utilized in expressions. Finally, the transition from L2 to L1 location, which encodes an end event, will fire when the dependencies of the end event are satisfied (dep_met(end(T))) and the task duration has elapsed, that is computed by the remaining work divided by the current pace. Moreover, valuation of the parameters of next task instance is computed on this transition.

The resource template has a simpler structure as shown in Figure 4. The transitions with the greedyClaim and lazyClaim channels fire at the occurrence of a start event, whereas the transition with the release channel fires when an end event occurs. The channels in this template are broadcast which implies that all resources that instantiate this, interact with any task where an event occurs. The resMessages array is shared between tasks and resources, and it is updated by a task in the setClaimRequest and setReleaseRequest functions with the amount of resource capacities claimed or

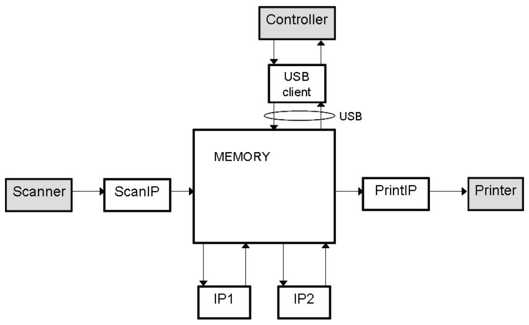


Figure 5: Océ printer architecture

released, respectively. At the occurrence of an event in a task, the resources that are used by the task have a non-zero value placed at their position in the resMessages array. Further, they subtract or add this value from their available current capacity.

V. Experiments

We now turn to an experimental evaluation of Uppaal models generated from Y-chart based representations that use the PPO theory. We compare these models with handcrafted models that have been presented in [21], built for printing systems. In the handcrafted models, each application has been modeled as a single automaton that contains all its component tasks. This way of modeling is more natural for design engineers but less efficient to analyze as we see further. All the experiments are 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. The printer architecture is depicted in Figure 5.

The cases explored here are depicted in Figures 6, 7 and 8. We use the same notation as in Figure 1. In addition, the rectangles contain, between parentheses, the task durations. The labels on the arrows specify the precedence function and the resources handed over. The circles encode resources and within parentheses their maximum capacity is listed (one by default). The dashed lines represent resource claims. The difference between the amount claimed and what is handed over is released at the end of a task.

In each experiment we computed the fastest time in which all tasks are completed (also called makespan) by using greedy scheduling. Three performance metrics were used to evaluate each experiment: the peak memory usage (the 'Mem' column) and running time (the 'Time' column) of Uppaal, and the total number of states explored. Table I gives the comparison results for the Direct Copy application in parallel with the Simple Print and Table II shows the Direct Copy case in parallel with Process from Store. The first two columns in each table indicate how many task instances are processed in total for each component task of an application.

To combat state space explosion, we used the sweep line method of Uppaal [26]. For this, each model was

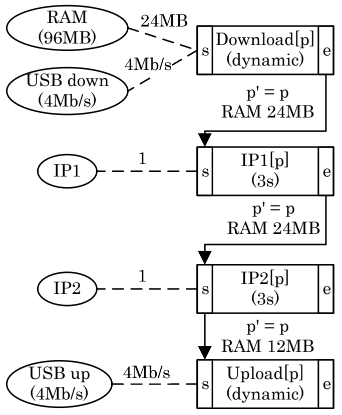


Figure 6: Process from Store

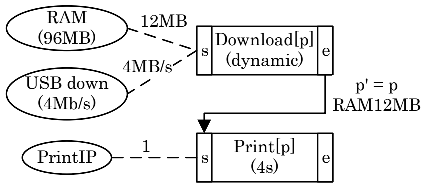


Figure 7: Simple Print

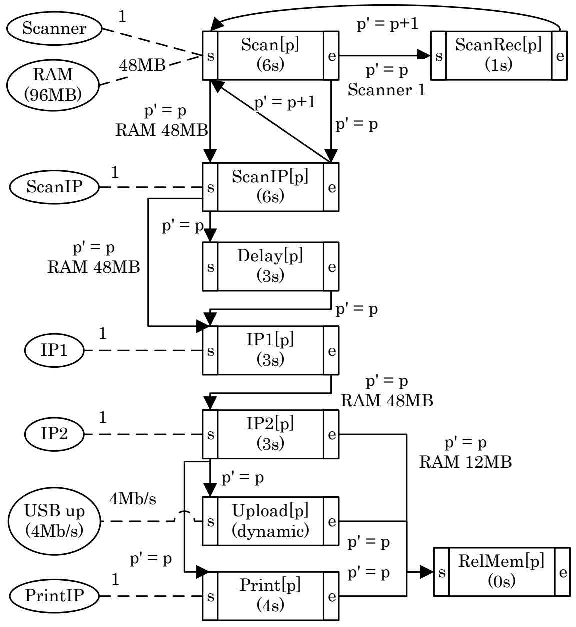


Figure 8: Direct Copy

Table I: 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
		5432	0.60	23	413
7	10	5480	1.60	71	10578
		5748	1.60	71	3050
35	50	12808	11.31	367	149926
		9572	17.00	367	48196
70	100	26568	27.92	737	433816
		18480	51.42	737	155491
334	500	598996	279.70	3585	6843592
		282732	961.98	3585	3038099
667	1000	2321768	1304.87	7166	25206064
		1076552	3964.64	7166	11704000
903	1355	4165896	1937.88	9705	45225661
		1962576	7805.70	9705	21272017
904	1356	O.M.	O.M.	O.M.	O.M.
		1965192	7655.01	9715	21302397
1460	1960	O.M.	O.M.	O.M.	O.M.
		4052524	18199.70	15117	44117751

Table II: 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
		5916	0.60	15	411
10	20	7540	4.10	114	47551
		7332	7.90	114	20118
25	50	21352	19.51	279	334606
		14420	48.03	279	135453
120	240	586172	384.66	1324	8255421
		244920	1122.34	1324	3269408
240	480	2555392	1857.78	2644	33327861
		1008512	4824.23	2644	13162088
303	606	4077452	2419.45	3337	53223828
		1573952	8196.21	3337	21007415
304	608	O.M.	O.M.	O.M.	O.M.
		1583572	8155.78	3348	21146664
480	960	O.M.	O.M.	O.M.	O.M.
		4057584	23394.71	5284	52819448

annotated with progress measures (i.e. task instance) that Uppaal uses during the analysis to store only the states where the progress measures are weakly monotonically increasing, or occasionally decreasing.

The state space of the generated models is significantly smaller than the state space of the handcrafted models (reduced between 41% and 71%). There are two causes that lead to the large difference in the number of states explored during the analysis of the two models. Firstly, the resource template of the handcrafted models has three states: Idle, Running and one to model a recovery phase that some resources like Scanner require. In the generated models, the recovery phase is modeled as an additional task. The other cause is the tasks that need more than one resource for processing. In the generated models, one can easily model multi-resource claims using broadcast channels and a shared array, as explained in Section IV. By contrast, in the handcrafted models, a multi-resource claim was modeled by third party automata. These automata were placed between the application automata and resource automata. For each resource required, we added an extra third party automaton. This extra automaton registers the claim to the resource automaton then it waits for the resource automaton to become available. When it is available it sends the request. On completion of the processing, it sends an end event to the application automaton.

Tables I and II also show up to a 61% decrease in the peak memory used by Uppaal during the analysis. However, analysis of the generated models requires more time. This happens due to the parametric representation that characterizes the generated models where a lot of details were encoded into functions. Some of these functions require a lot of time to be evaluated due to the conditions or function calls that they incorporate.

VI. Conclusions

PPOs are a simple extension of partial orders, but expressive enough to compactly represent large task graphs with repetitive behavior. We have 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. We also presented experiments which demonstrate that the Uppaal models obtained through this 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 [27], [28] 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 [4] to the PPO settings. This approach reduces the complexity of scheduling problems by exploiting the 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.

Acknowledgment: 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. Balarin, *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer, 1997.

[2] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *ASAP*, IEEE CS, 1997, pp. 338–349.

[3] N. v. d. Nieuwelaar, J. v. d. Mortel-Fronczak, and J. Rooda, "Design of supervisory machine control," in *European Control Conference*, 2003.

[4] M. Hendriks, B. van den Nieuwelaar, and F. Vaandrager, "Recognizing finite repetitive scheduling patterns in manufacturing systems," in *MISTA*, University of Nottingham, 2003, pp. 291–319.

[5] A. Mazurkiewicz, "Trace theory," in *Petri Nets: Applications and Relationships to Other Models of Concurrency*, LNCS 255, Springer, 1987, pp. 278–324.

[6] V. Pratt, "Modeling concurrency with partial orders," *Int. Journal of Parallel Programming*, vol. 15, pp. 33–71, 1986.

[7] G. Winskel, "An introduction to event structures," in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, Springer, 1989, pp. 364–397.

[8] K. Jensen, L. Kristensen, and L. Wells, "Coloured Petri nets and cpn tools for modelling and validation of concurrent systems," *STTT*, vol. 9, pp. 213–254, 2007.

[9] K. Jensen and L. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.

[10] Octopus toolset homepage, http://dse.esi.nl, 2011.

[11] T. Basten, E. Van Benthum, M. Geilen, M. Hendriks, F. Houben, G. Igna, F. Reckers, S. De Smet, L. 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*, LNCS 6415, Springer, 2010, pp. 90–105.

[12] 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*, IEEE CS, 2011, pp. 13–18.

[13] S. Stuijk, M. Geilen, and T. Basten, "Sdf^3: Sdf for free," in *ACSD*, IEEE CS, 2006, pp. 276–278.

[14] G. Behrmann, A. David, and K. Larsen, "A tutorial on Uppaal," in *SFM-RT 2004*, LNCS 3185, Springer, 2004, pp. 200–236.

[15] R. Alur and D. L. Dill, "A theory of timed automata," *TCS*, vol. 126, pp. 183–235, 1994.

[16] Y. Abdeddaïm, A. Kerbaa, and O. Maler, "Task graph scheduling using timed automata," in *IPDPS*, IEEE CS, 2003, pp. 237.

[17] M. Hendriks and M. Verhoef, "Timed automata based analysis of embedded system architectures," *IPDPS*, IEEE CS, 2006.

[18] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour, "Influence of different system abstractions on the performance analysis of distributed real-time systems," in *EMSOFT*, ACM, 2007, pp. 193–202.

[19] J. Berendsen, B. Gebremichael, F. Vaandrager, and M. Zhang, "Formal specification and analysis of zeroconf using Uppaal," *ACM TECS*, vol. 10, no. 3, 2011.

[20] 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*, LNCS 5469, Springer, 2009, pp. 90–104.

[21] G. Igna, V. Kannan, Y. Yang, T. Basten, M. Geilen, F. Vaandrager, M. Voorhoeve, S. Smet, and L. Somers, "Formal modeling and scheduling of datapaths of digital document printers," in *FORMATS*, LNCS 5215, Springer, 2008, pp. 170–187.

[22] G. Igna and F. Vaandrager, "Verification of printer datapaths using timed automata," in *ISoLA*, LNCS 6415, Springer, 2010, pp. 412–423.

[23] I. AlAttili, F. Houben, G. Igna, S. Michels, F. Zhu, and F. Vaandrager, "Adaptive scheduling of data paths using Uppaal Tiga," in *QFM*, Electronic Proceedings in TCS, vol. 13, 2009, pp. 1–12.

[24] R. v. Glabbeek and G. Plotkin, "Configuration structures, event structures and Petri nets," *TCS*, vol. 410, no. 41, pp. 4111–4159, 2009.

[25] J. Berendsen and F. Vaandrager, "Compositional abstraction in real-time model checking," in *FORMATS*, LNCS 5215, Springer, 2008, pp. 233–249.

[26] S. Christensen, L. Kristensen, and T. Mailund, "A sweep-line method for state space exploration," in *TACAS*, LNCS 2031, Springer, 2001, pp. 450–464.

[27] D. Peled, "Ten years of partial order reduction," in *CAV*, LNCS 1427, Springer, 1998, pp. 17–28.

[28] K. L. McMillan, "Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits," in *CAV*, LNCS 663, Springer, 1992, pp. 164–177.