Verification of Real-Time Coordination in VirtuosoNext (extended version)

Guillermina Cledou
HASLab/INESC TEC, Universidade do Minho, Portugal
mgc@inesctec.pt

José Proença
CISTER/ISEP & INESC-TEC, Porto, Portugal
pro@isep.ipp.pt

Bernhard H.C. Sputh
Altreonic NV, Belgium
bernhard.sputh@altreonic.com

Eric Verhulst
Altreonic NV, Belgium
bernhard.sputh@altreonic.com

Abstract

VirtuosoNext^{TM} is a distributed real-time operating system (RTOS) featuring a generic programming model dubbed *Interacting Entities*. This paper focuses on these interactions, implemented as so-called *Hubs*. Hubs act as synchronisation and communication mechanisms between the application tasks and implement the services provided by the kernel as a kind of Guarded Protected Action with a well defined semantics. While the kernel provides the most basic services, each carefully designed, tested and optimised, tasks are limited to this handful of basic hubs, leaving the development of more complex mechanisms up to application specific implementations.

In this work we investigate how to support a programming paradigm to compositionally build new services, using notions borrowed from the Reo coordination language, and relieving tasks from coordination aspects while delegating them to the hubs. We formalise the semantics of hubs using an automata model with notions of dataflow and time, identify the behaviour of existing hubs, and propose an approach to build new hubs by composing simpler ones. We also provide open-source tools and methods to analyse and verify hubs under our automata interpretation, including time-sensitive behaviour via the UPPAAL model checker, usable on `http://arcatools.org/hubs`. In a first experiment several hub interactions are combined into a single more complex hub, which raises the level of abstraction and contributes to a higher productivity for the programmer. We illustrate the proposed tools and methods by verifying key properties on different interaction scenarios between tasks and the specified hub. Finally, we investigate the impact on the performance by comparing different implementations on an embedded board.

Keywords and phrases Coordination, timed automata, UPPAAL, Real-time OS, Compositional semantics

Digital Object Identifier 10.5281/zenodo.3818020

1 Introduction

When developing software for resource-constrained embedded systems, optimising the utilization of the available resources is a priority. In such systems, many system-level details can influence time and performance in the execution, such as interactions with the cache, mismatches between CPU clock speed, the speed of the external memory, and connected peripherals, leading to unpredictable execution times. VirtuosoNext [1] is a Real Time operating system developed by the company Altreonic that runs efficiently on a range of small embedded devices, and is accompanied by a set of visual development tools – Visual Designer – that generates the application framework and provides tools to analyse the timing behaviour in detail.

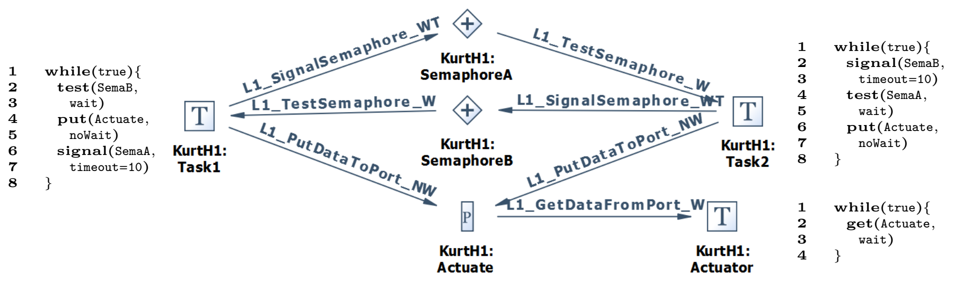
The developer is able to organise a program into a set of individual tasks, scheduled and coordinated by the VirtuosoNext kernel. The coordination of tasks is a non-trivial process. A kernel process uses a priority-based preemptive scheduler deciding which task to run at each time, with hub services used to synchronise and pass data between tasks. A fixed set of hubs is made available by the Visual Designer, which are used to coordinate the tasks. For example, a FIFO hub allows one or more values to be buffered and consumed exactly once, a Semaphore hub uses a counter to synchronise tasks based on counting events, and a Port hub synchronises two tasks, allowing data to be copied between the tasks without being buffered. However, the set of available hubs is limited. Creating new hubs to be included in the mainline distribution is difficult since each hub must be carefully designed, model checked, implemented and tested. It is still possible for users to create specific hubs in their installations, however they would need to fully implement them, losing the assurances of existing hubs.

This paper provides the following contributions. Parts in bold denote new results regarding the associated conference publication [2].

- The formalisation of hubs as **timed** (hub) automata (Section 3),
  - capturing hubs currently present in VirtuosoNext (without real time),
  - suggesting new core hubs for VirtuosoNext (some **with real time**).
- A compositional semantics for hubs as **timed** automata (Section 4).
- Online tools (`http://arcatools.org/hubs`) to analyse hubs (Section 5),
  - using a DSL to specify hubs built by composing simpler hubs,
  - **using a DSL to specify timed contracts of tasks' interactions,**
  - generating graphs and composed automata with dynamic layouts,
  - **introducing a temporal logic focused on interactions,**
  - **generating and running extended UPPAAL specifications and logic formulas,** and
  - including other analysis of hubs.
- Analysis of time traces of hubs on an embedded platform (Section 6).

Using the *existing set of hubs in VirtuosoNext*, we can express in our DSL the scenario `{task<t1>(W s!) semaphore(s,t) task<t2>(2 t?) every 3}`, where a semaphore is connecting 2 tasks via the ports `s` and `t`. Here s waits indefinitely, marked with W, and t waits for at most 2 time units before timing out, trying every 3 time units. We can then specify and verify temporal properties of this scenario using our framework, such as *"every time* `s` *fires,* `t` *will eventually fire in less than 3 time units"*. The verification process uses UPPAAL, by encoding properties and models, hiding from the user the underlying automata, including auxiliary variables and clocks.

This paper and the proposed framework address hubs that go *beyond what is currently supported by VirtuosoNext*, by describing new hubs (not part of VirtuosoNext's), and allowing hubs to be connected to other hubs directly. The new hubs include a synchronous duplicator that requires all output ports to synchronise, and a timer that can buffer a value for a certain time. The composition of hubs maps introduces the possibility of specifying complex interaction protocols, inspired in Reo's syntax [3] and real-time semantics [4, 5, 6]. Currently, without these complex protocols, the orchestration code must be intertwined with the tasks' behaviour. Our tools provide some insights on the code size, required memory, and number of context switches of a composition of hubs, which can be leveraged by Altreonic to produce new hubs.



```
while(true){
  test(SemaB,
    wait)
  put(Actuate,
    noWait)
  signal(SemaA,
    timeout=10)
}
```

```
while(true){
  signal(SemaB,
    timeout=10)
  test(SemaA,
    wait)
  put(Actuate,
    noWait)
}
```

```
while(true){
  get(Actuate,
    wait)
}
```

Figure 1 Example application in VirtuosoNext, whereby two tasks communicate with an actuator in a round robin sequence through two semaphores and a port.

2 Distributed tasks in VirtuosoNext

A VirtuosoNext *system* is executed on a target system, composed of processing *nodes* and communication *links*. Orthogonally, an *application* consists of a number of *tasks* coordinated by *hubs*. Unlike links, hubs are independent of the hardware topology. When building application images, the code generators of VirtuosoNext map tasks and hubs onto specific nodes, taking into account the target platforms. A special *kernel task*, running on each node, controls the scheduling of tasks, the hub services, and the internode communication and routing.

This section starts by giving a small overview of how tasks are built and composed, followed by a more detailed description over existing hubs.

2.1 Example of an architecture

A program in VirtuosoNext is a fixed set of tasks, each running on a given computational node, and interacting with each other via dedicated interaction entities, called hubs. Consider the example architecture in Fig. 1, where tasks Task1 and Task2 send instructions to an Actuator task in a round robin sequence. SemaphoreA tracks the end of Task1 and the beginning of Task2, while SemaphoreB does the reverse, and port Actuate forwards the instructions from each task to the Actuator. In this case two Semaphore hubs were used, depicted by the diamond shape with a '+', and a Port hub, depicted by a box with a 'P'. Tasks and hubs can be deployed on different processing nodes, but this paper will consider only programs deployed in the same node, and hence omit references to nodes. This and similar examples can be found in the VirtuosoNext's manual [7].

2.2 Task coordination via Hubs

Hubs are coordination mechanisms between tasks that coordinate via *put* and *get* service requests to transfer information from one task to another. This can be a data element, the notification of an event occurrence, or some logical entity that needs to be protected for atomic access. A call to a hub constitutes a descheduling point in the tasks' execution. The behaviour depends on which hub is selected, e.g. tasks can simply synchronise (with no data being transferred) or synchronise while transferring data (either buffered or non-buffered). Other hubs include the Resource hub, often used to request atomic access to a resource, and hubs that act as gateways to peripheral hardware.

Any number of tasks can make *put* or *get* requests to a given hub. Such requests will be queued in waiting lists (at each corresponding hub) until they can be served. Waiting lists are ordered by task priority – requests get served by following such an order. In addition, requests can use different interaction semantics. These interaction semantics determine how a task waits on a request to succeed. There are three synchronous and one asynchronous interaction semantics in VirtuosoNext.

Table 1 Examples of existing Hubs in VirtuosoNext

Hub	Waiting Lists for Service Requests
Port	`put` – signals some data entering the port; and `get` – signals some data leaving the port. Both must synchronize to succeed.
Event	`raise` – sets an event, succeeding if not set yet; and `test` – checks if an event happened, in which case succeeds, and clears the event.
DataEvent	`update` – sets an event and buffers some data, overriding any previous data. Always succeeds; `read` – reads the data. Succeeds if the event is set; and `clear` – clears the buffer and the event.
Semaphore	`signal` – signals the semaphore, incrementing an internal counter `c`. Succeeds if `c < MAX`; and `test` – checks if `c > 0`, in which case succeeds, and decrements `c`.
Resource	`lock` – locks a logical resource and buffers the `id` of the requesting task. Succeeds only if the resource is free; and `unlock` – unlocks the resource. Succeeds only if locked by the same task.
FIFO	`enqueue` – buffers some data in the queue. Succeeds if the queue is not full; and `dequeue` – gets data from the queue. Succeeds if the queue is not empty.
Blackboard	`update` – buffers some data, overriding any previews data, incrementing a sequence number. Always succeeds; `read` – reads the data and the sequence number. Succeeds if not empty. Reader tasks can use the sequence number to attest the freshness of the data; and `wipe` – clears the buffer.

Here we focus on the first three. These can be: *waiting* (`W`) – a task waits indefinitely until the request can be served; *non-waiting* (`NW`) – either the requests is served without delay or the request fails; *waiting with time-out* (`WT`) – waits either until the request is served or the specified time-out has expired. In our example in Figure 1, observe that both tasks send signal messages with a timeout of 10ms, wait indefinitely for test messages, and send messages to the actuator without waiting to synchronise.

There are various hubs available, each with its predefined semantics [7]. Table 1 describes some of them and their put and get service request methods.

3 Deconstructing Hubs via Timed Automata

This section formalises *hubs*, using an automata model with variables and time, providing both a syntax (Section 3.1) and a semantics (Section 3.2).

3.1 Syntax

Our previous publication [2] formalises the behavioural semantics of hubs using an automata model with variables, called *Hub Automata*. Here, we present an extension of this model with dense time, as in timed automata [8], which we call *Timed Hub Automata* (THA), which will be able to capture interaction with time-sensitive tasks (Section 5.1).

Informally, a *timed hub automaton* is a finite automaton enriched with *clocks* over $\mathbb{R}_{\geq 0}$, *variables* over a data domain $\mathcal{D}$, and an *initial valuation* of such variables; and where transitions are enriched with *multi-actions*, and logic guards and updates over so-called *clocks* and *updates*. We call *clock constraints* the guards over clocks and *clock resets* the updates over clocks that set a given collection of clocks to zero.

A *clock* c is a logical entity that captures the (continuous and dense) time that has passed since it was last reset, and which can only be inspected or reset. When an automaton evolves over time, all clocks are incremented simultaneously. Initially, all clocks are set to zero.

In the following, we provide a syntax for clock and data constraints, clock and data updates, and THA.

▶ **Definition 1** (Clock and Data Constraints). *A* clock constraint *over a set of clocks C, written $cc \in \mathcal{C}(C)$ is defined by:*

$$cc ::= c \boxdot n \mid c - c \boxdot n \mid cc \wedge cc \mid \top \qquad \text{(clock constraint)}$$

where $c \in C$, $n \in \mathbb{N}$, and $\boxdot \in \{<, \leq, =, >, \geq\}$. A data constraint *(or* guard*) over a set of variables $\mathcal{X}$, written $g \in \Phi(\mathcal{X})$, is defined by:*

$$g \ := \ \top \mid pred(\overline{x}) \mid g \wedge g \mid \neg g \qquad \text{(data constaint)}$$

where $x \in \mathcal{X}$ is a variable, $\overline{x}$ is a sequence of variables, and $pred \in \mathcal{P}red$ is a predicate. The remaining logic connectives can be achieved in the usual way $g \vee g = \neg(\neg g \wedge \neg g)$ and $\bot = \neg\top$.

▶ **Definition 2** (Clock and data updates). *A* clock update *(or* clock reset*) over a set of clocks C, written $r \subseteq C$, is a set of clocks that are set to zero, also written as $c_1 \leftarrow 0; c_2 \leftarrow 0$ instead of $\{c_1, c_2\}$.*

A data update *over a set of variables $\mathcal{X}$, written $u \in \mathcal{U}(\mathcal{X})$, is defined by:*

$$u \ := \ x \leftarrow e \mid u; u \mid u|u \mid \epsilon \qquad \text{(data update)}$$
$$e \ := \ d \mid x \mid f(\overline{x}) \qquad \text{(expression)}$$

where $d \in \mathcal{D}$ is a data value, and $f \in \mathcal{F}$ is a deterministic function without side-effects. The construct $u; u$ denotes sequential composition, $u|u$ denotes parallel composition, and ϵ denotes an empty update.

For example, the update $x \leftarrow 2; (y \leftarrow z + 1 \mid z \leftarrow z * 2)$ starts by setting x to 2, and then sets y to $z + 1$ and z to $z * 2$ in some (a-priori unknown) order. Note that the order of evaluation of the parallel assignments will affect the final result. We avoid non-determinism by following up dependencies (e.g., $z \leftarrow z * 2$ should be executed before $y \leftarrow z + 1$) and by requiring that the order of executing any two independent assignments does not affect the result. This will be formalised later in the paper.

Hubs interact with the environment through ports that represent actions. Let $\mathcal{P}$ be the set of all possible ports uniquely identified. For a $p \in \mathcal{P}$, $\hat{p}$ is a variable holding a data value flowing through port p. We use $\hat{\mathcal{P}}$ to represent the set of all data variables associated to ports in $\mathcal{P}$.

▶ **Definition 3** (Timed Hub Automata (THA)). *A* timed hub automaton *is a tuple $H = (L, \ell_0, P, \mathcal{X}, \delta_0, C, I, \rightarrow)$ where L is a finite set of locations, ℓ_0 is the initial location, $P = P_I \uplus P_O$, is a finite set of ports, with P_I and P_O representing the disjoint sets of input and output ports, respectively, $\mathcal{X}$ is a finite set of internal variables, $\delta_0 : \mathcal{X} \rightarrow \mathcal{D}$ is the initial valuation that maps variables in $\mathcal{X}$ to a value in $\mathcal{D}$, C is a finite set of clocks, $I : L \rightarrow \mathcal{C}(C)$ is the invariant function that assigns clock constraints to locations, and $\rightarrow \subseteq L \times \Phi(\mathcal{X} \cup \hat{\mathcal{P}}) \times \mathcal{C}(C) \times 2^P \times \mathcal{U}(\mathcal{X} \cup \hat{\mathcal{P}}) \times 2^C \times L$ is the transition relation.*

For a given transition $(\ell, g, cc, \omega, u, r, \ell') \in \rightarrow$, also written $\ell \xrightarrow{g,cc,\omega,u,r} \ell'$, ℓ is the source location, g and cc are the guard and clock constraint defining the enabling condition, ω is the set of ports triggering the transition, u is the update triggered, r is the set of clocks to reset, and ℓ' is the target location. The set $\mathcal{X}$ represents internal variables know only by the automaton, while $\hat{P}$ represents external variables, known also by the environment.

A transition $\ell \xrightarrow{g,cc,\omega,u,r} \ell'$ is enabled only if (1) all of its ports ω are ready to be executed simultaneously, and (2) the current variable valuation and clock valuation satisfy the associated guard g and clock constraint cc, respectively. Performing this transition means applying the update u to the current valuation, reseting the clocks in r, and moving to location ℓ'. This is formalised in the follow up section.

Fig. 2 depicts the THA for each of the hubs described in Section 2.2, except the Resource hub (for space restrictions), and includes new hubs that are not yet implemented by VirtuosoNext, marked with '*'. Most of the automata do not include clocks, i.e., they are not time-sensitive, since time constraints will be provided by the environment (the tasks). The new hubs illustrate two categories of hubs not currently exploited in VirtuosoNext: (1) providing a handshake of more than 2 ports, and (2) including time-dependent behaviour. For simplicity, we omit location invariants and guards that are trivially satisfied, i.e. $\top$.

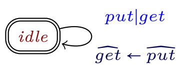
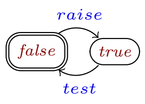
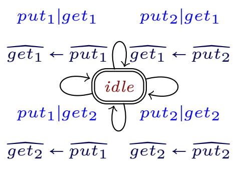
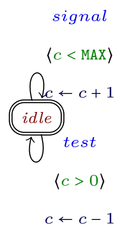
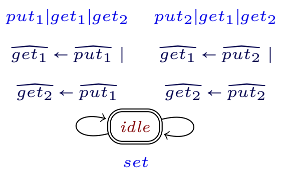
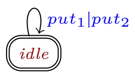
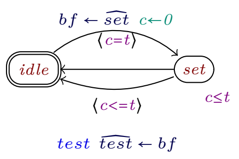
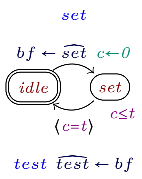
Example: FIFO. Consider the Hub Automaton for the FIFO hub (Fig. 2), implemented using an internal circular queue, with size N and with elements of type T. Initially, the FIFO is at location *idle* and its internal variables are assigned as follows: $c \mapsto 0$, $f \mapsto 0$, $p \mapsto 0$, and $bf_i \mapsto null$ for all $i \in \{0 \ldots N-1\}$. Here c is the current number of elements in the queue, f and p are the pointers to the front and last empty place of the queue, respectively, and each bf_i holds the value of the i-th position in the queue. The FIFO can *enqueue* an element —if the queue is not full ($c < N$)— storing the incoming data value in bf_p, and increasing the c and p counters; or it can *dequeue* an element—if the queue is not empty ($c \geq 1$), updating the corresponding variables.

Note that more than one task can be using the same port of a given hub. In these cases VirtuosoNext selects one of the tasks to be executed, using its scheduling algorithm. The semantics of this behaviour is illustrated in the automaton of Port$^+$, that uses multiple incoming and outgoing ports, denoting all possible combinations of inputs and outputs. This exercise can be applied to any hub other than the Port hub.

Example: Duplicator and Drain. Both the Duplicator and the Drain hubs, inspired by Reo connectors, do not exist yet in VirtuosoNext. The Duplicator is a variation of the Port hub that broadcasts a given input to all its outputs atomically, i.e., all outgoing ports must receive the data message before the original sender can resume its execution. The Drain is another variation of the Port hub that ignores data values, and forces all participating ports to synchronise before proceeding.

Example: P-Timer and B-Timer The P-Timer(t,T) (*precise-timer*) and B-Timer(t,T) (*bounded-timer*) hubs are a variation of DataEvent$(1,T)$ with time constraints. P-Timer(t,T) buffers a received value for precisely t time units, and then sends it to its outgoing port, and B-Timer(t,T) buffers a value for at most t time units, after which the data value is discarded.

Observe that the P-Timer can produce a *timelock*, i.e., a deadlock caused by the model forbidding time to pass, depending on the availability of their ports. More specifically, when it buffers a value for t time units, but the port *test* is not available to communicate. We consider this to be neither a problem of the THA formalism, nor a problem of the P-Timer hub. Instead, we consider it a modelling flaw when combining a P-Timer with an incompatible environment, similarly to deadlocks that arise from combining a Semaphore with an environment that tries to perform a *test* before any *signal* being sent. As such, we find it crucial to provide tools to detect undesired (and desired) behaviour statically, and later propose the use of the UPPAAL model checker for THA.

Hub	Automaton	Hub	Automaton
Port $\mathcal{X} = \varnothing$		Event $\mathcal{X} = \varnothing$	
Port$^+$ $\mathcal{X} = \varnothing$		Semaphore $\mathcal{X} = \{c : \mathbb{N}\}$ $\delta_0 = \{c \mapsto 0\}$	
Duplicator* $\mathcal{X} = \varnothing$		Drain* $\mathcal{X} = \varnothing$	
B-Timer (t,T) $\mathcal{X} = \{bf : T\}$ $C = \{c\}$ $\delta_0 = \{bf \mapsto null\}$		P-Timer (t,T) $\mathcal{X} = \{bf : T\}$ $C = \{c\}$ $\delta_0 = \{bf \mapsto null\}$	

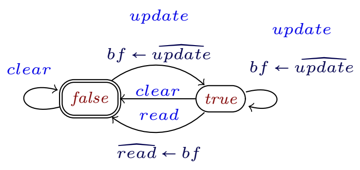
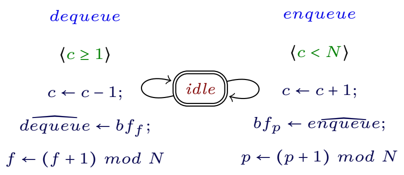
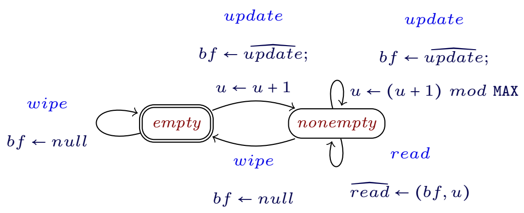
Hub	Automaton
DataEvent (T) $\mathcal{X} = \{bf : T\}$ $\delta_0 = \{bf \mapsto null\}$	
FIFO (N,T) $\mathcal{X} = \{c{:}\mathbb{N}, f{:}\mathbb{N}, p{:}\mathbb{N}\}$ $\cup \{bf_i{:}T \mid i < N\}$ $\delta_0 = \{c, f, p\} \mapsto 0$ $\cup \{bf_i \mapsto null \mid i < N\}$	
Blackboard (T) $\mathcal{X} = \{bf : T, u : \mathbb{N}\}$ $\delta_0 = \{bf \mapsto null\}$ $\cup\{u \mapsto 0\}$	

Figure 2 Automata semantics of hubs – from VirtuosoNext except those with *. Port$^+$ captures how VirtuosoNext interprets multiple calls to the same port.

3.2 Semantics

This section formalises the satisfaction of (clock and data) constraints, the effect of updating (clock and data) valuations, and the evolution of THA.

▶ **Definition 4** (Clock valuation and satisfaction). *A* clock valuation *η for a set of clocks C is a function $\eta : C \to \mathbb{R}_{\geq 0}$ that assigns each clock $c \in C$ to its current value $\eta(c)$. We use $\mathbb{R}^C$ to refer to the set of all clock valuations over a set of clocks C.*

The satisfaction *of a clock constraint cc by a valuation η, written $\eta \vDash cc$, is defined as follows:*

$$\begin{array}{llll} \eta \vDash \top & \textit{always} & \eta \vDash cc_1 \wedge cc_2 & \textit{if } \eta \vDash cc_1 \textit{ and } \eta \vDash cc_2 \\ \eta \vDash c \boxdot n & \textit{if } \eta(c) \boxdot n & \eta \vDash c_1 - c_2 \boxdot n & \textit{if } \eta(c_1) - \eta(c_2) \boxdot n \end{array} \qquad \text{(clock satisfaction)}$$

where $\boxdot \in \{<, \leq, =, >, \geq\}$.

▶ **Definition 5** (Data valuation and satisfaction). *A* data valuation *δ for a set of variables $\mathcal{X}$ is a function $\delta : \mathcal{X} \to \mathcal{D}$ that assigns every variable $x \in \mathcal{X}$ to its current value $\delta(x)$. We use $\mathcal{D}^{\mathcal{X}}$ to refer to the set of all data valuations over a set of variables $\mathcal{X}$.*

The satisfaction *of a data constraint g by a valuation δ, written $\delta \vDash g$, is defined as follows, assuming an interpretation $\mathcal{I}$ of predicates.*

$$\begin{array}{llll} \delta \vDash \top & \textit{always} & \delta \vDash \phi_1 \wedge \phi_2 & \textit{if } \delta \vDash \phi_1 \textit{ and } \delta \vDash \phi_2 \\ \delta \vDash \bot & \textit{never} & \delta \vDash \phi_1 \vee \phi_2 & \textit{if } \delta \vDash \phi_1 \textit{ or } \delta \vDash \phi_2 \\ \delta \vDash \neg\phi & \textit{if } \delta \nvDash \phi & \delta \vDash pred(\overline{x}) & \textit{if } \mathcal{I}(pred)(\delta(\overline{x})) \textit{ evaluates to } \texttt{true} \end{array} \qquad \text{(data satisfaction)}$$

▶ **Definition 6** (Clock and data updates). *Let $\eta_0(c) = 0$ for all $c \in C$ be the initial clock valuation that sets to 0 all clocks in C. We use $\eta + d$, $d \in \mathbb{R}_{\geq 0}$, to denote the clock assignment that maps all $c \in C$ to $\eta(c) + d$, and let $\eta[r \mapsto 0]$, $r \subseteq C$, be the clock assignment that maps all clocks in r to 0 and agrees with η for all other clocks in $C \smallsetminus r$.*

Given a serialisation function σ that converts general updates into sequences of assignments, the application of an update u to a valuation δ is given by $\delta[\sigma(u)]$, where $\delta[-]$ is defined below.

$$\begin{array}{rl} \delta[x \leftarrow d](x) & = d \\ \delta[x \leftarrow y](x) & = \delta(y) \\ \delta[x \leftarrow f(\overline{y})](x) & = f(\overline{\delta(y)}) \end{array} \qquad \begin{array}{rll} \delta[x \leftarrow e](y) & = \delta(y) & \textit{if } x \neq y \\ \delta[u_1; u_2](x) & = (\delta[u_1])[u_2](x) \end{array}$$

Here a *serialisation function* is any function that maps each data update (c.f. Def. 2) to a new one that contains the same assignments and does not use the parallel operator. Section 4.3 will present the specific serialisation used in our prototype implementation. This serialization function is partial–i.e., it rejects parallel updates with cyclic variable dependencies. We will omit σ when not relevant.

Serialisable THA. We say a THA is *serialisable* if the serialization function is defined for all its updates, and we say it is *serialised*, if non of its updates contains the parallel operator.

The execution of an automaton is defined as sequences of steps that do not violate the guards, and such that each step updates the current variable valuation according to the corresponding update. A configuration $(\ell, \delta, \eta, \alpha)$ captures the current location ℓ, the variable assignment $\delta : \mathcal{X} \to \mathcal{D}$, the clock valuation η, and the set of ports α that were last fired. Each step is either an *action*—denoted by a partial valuation function $\delta_{io} : \hat{\mathcal{P}} \to \mathcal{D}$ that maps variables associated to ports to their current value in the environment—, or a *time delay* $t \in \mathbb{R}_{\geq 0}$. Actions update ℓ, δ, and α, and time delays update η. We use $\delta \uplus \delta_{io}$ to denote the map δ extended with δ_{io}.

Extension for verification. We extend the core syntactic and semantic rules of THA only for verification purposes, discussed further in Section 5.2. We highlight these extensions, which can be omitted without affecting the core semantics of THA. We extend every THA with the set α, mentioned above, and with extra variables and clocks: for every port a there is a clock t_a that is set to 0 when a is fired, and a boolean variable $done_a$ that is initially false and set to true when a fires. We denote the new sets of variables and clocks D_P and C_P. Let S be a set of ports, we use two auxiliary functions $\mathsf{done}(S) = \{done_a \mapsto \mathit{true} \mid a \in S\}$ and $\mathsf{reset}(S) = \{t_a \mapsto 0 \mid a \in S\}$, that set *done* and clock variables of ports in S to *true* and to 0, respectively.

▶ **Definition 7** (Semantics of Hubs). *The semantics of a serialised THA* $H = (L, \ell_0, P, \mathcal{X} \uplus D_P, \delta_0, C \uplus C_P, I, \rightarrow)$ *is given by the rules below, starting on configuration* $(\ell_0, \delta_0, \eta_0, \alpha)$.

$$\text{(act)}\ \frac{\ell \xrightarrow{g,cc,\mathsf{dom}(\delta_{io}),u,r} \ell' \quad \eta \vDash cc \quad \delta \uplus \delta_{io} \vDash g \qquad \eta' = \eta[r \mapsto 0] \uplus \mathsf{reset}(\mathsf{dom}(\delta_{io})) \qquad \delta' = (\delta \uplus \delta_{io})[u] \setminus \hat{P} \cup \mathsf{done}(\mathsf{dom}(\delta_{io}))}{(\ell, \delta, \eta, \alpha) \xrightarrow{\delta_{io}} (\ell', \delta', \eta', \mathsf{dom}(\delta_{io}))}$$

$$\text{(delay)}\ \frac{\eta \vDash I(\ell) \qquad \forall_{t' \in [0,t]} \cdot \eta + t' \vDash I(\ell)}{(\ell, \delta, \eta, \alpha) \xrightarrow{t} (\ell, \delta, \eta + t, \alpha)}$$

Example: FIFO steps. Below is a valid trace of a FIFO hub with size 3 (Fig. 2). In this example there are no clocks nor time constraints, hence time can pass freely, as illustrated in the 2nd step, taking 5 time units while remaining in the same configuration.

$$\begin{array}{rl}
 & (idle,\ \{c \mapsto 0, f \mapsto 0, p \mapsto 0, bf_0 \mapsto null, bf_1 \mapsto null, bf_2 \mapsto null\}, \varnothing) \\
\xrightarrow{\{\widehat{enqueue} \mapsto 42\}} & (idle,\ \{c \mapsto 1, f \mapsto 0, p \mapsto 1, bf_0 \mapsto 42, \quad bf_1 \mapsto null, bf_2 \mapsto null\}, \varnothing) \\
\xrightarrow{5} & (idle,\ \{c \mapsto 1, f \mapsto 0, p \mapsto 1, bf_0 \mapsto 42, \quad bf_1 \mapsto null, bf_2 \mapsto null\}, \varnothing) \\
\xrightarrow{\{\widehat{dequeue} \mapsto 42\}} & (idle,\ \{c \mapsto 0, f \mapsto 1, p \mapsto 1, bf_0 \mapsto 42, \quad bf_1 \mapsto null, bf_2 \mapsto null\}, \varnothing)
\end{array}$$

Example: B-Timer steps. Below is a valid trace of a B-Timer hub with its timer set to 5 (Fig. 2). While in state *idle*, time can pass freely, as illustrated in the 1st step, taking 10 time units while remaining in the same configuration. After the timeout is set, as shown in the 2nd step, clock c can not growth beyond 5 time units. In this case, *test* does not synchronize, and the silent transition in the 4th step, sets the process back to the *idle* state.

$$\begin{array}{rl}
 & (idle,\ \{bf \mapsto null, t \mapsto 5\}, \{c \mapsto 0\}) \\
\xrightarrow{10} & (idle,\ \{bf \mapsto null, t \mapsto 5\}, \{c \mapsto 10\}) \\
\xrightarrow{\{\widehat{set} \mapsto 42\}} & (set,\ \{bf \mapsto 42, \quad t \mapsto 5\}, \{c \mapsto 0\}) \\
\xrightarrow{5} & (set,\ \{bf \mapsto 42, \quad t \mapsto 5\}, \{c \mapsto 5\}) \\
\longrightarrow & (idle,\ \{bf \mapsto 42, \quad t \mapsto 5\}, \{c \mapsto 5\})
\end{array}$$

4 Reconstructing Hubs

Complex hubs can be built by composing simpler hubs, following the same ideas behind Reo [3]. The composition uses two simpler operations: *product* and *synchronisation*. This section starts by defining these two operations, followed by an example and by a suitable definition of serialisation of updates.

4.1 Hub composition

The *product* operation takes two hubs with disjoint ports and variables, and produces a new hub that interleaves and joins transitions from both hubs, i.e. fully concurrent. The *synchronisation* operation is conducted over a Hub Automaton H and it links pairs of ports a and b in P forcing them fire only together.

▶ **Definition 8** (Hub Product (×))**.** *Let H_1 and H_2 be two THA with disjoint sets of ports, variables, and clocks. The product of H_1 and H_2, written $H_1 \times H_2$, is a new Hub Automaton defined as*

$$H = (L_1 \times L_2, (l_{0_1}, l_{0_2}), P_1 \cup P_2, \mathcal{X}_1 \cup \mathcal{X}_2, \delta_{0_1} \cup \delta_{0_2}, C_1 \cup C_2, I, \rightarrow)$$

where $I(\ell_1, \ell_2) = I_1(\ell_1) \wedge I_2(\ell_2)$ and $\rightarrow$ is defined as follows.

$$\text{(prod-left)}\ \frac{\ell_1 \xrightarrow{g_1, cc_1, \omega_1, u_1, r_1} \ell_1'}{(\ell_1, \ell_2) \xrightarrow{g_1, cc_1, \omega_1, u_1, r_1} (\ell_1', \ell_2)} \qquad \text{(prod-right)}\ \frac{\ell_2 \xrightarrow{g_2, cc_2, \omega_2, u_2, r_2} \ell_2'}{(\ell_1, \ell_2) \xrightarrow{g_2, cc_2, \omega_2, u_2, r_2} (\ell_1, \ell_2')}$$

$$\text{(prod-both)}\ \frac{\ell_1 \xrightarrow{g_1, cc_1, \omega_1, u_1, r_1} \ell_1' \qquad \ell_2 \xrightarrow{g_2, cc_2, \omega_2, u_2, r_2} \ell_2'}{(\ell_1, \ell_2) \xrightarrow{g_1 \wedge g_2\,,\, c_1 \wedge c_2\,,\, \omega_1 \cup \omega_2\,,\, u_1 | u_2\,,\, r_1 \cup r_2} (\ell_1', \ell_2')}$$

▶ **Definition 9** (Hub Synchronisation (Δ))**.** *Let $H = (L, l_0, P, \mathcal{X}, \delta_0, C, I, \rightarrow)$ be a THA, a and b two ports in P, and x_{ab} a fresh variable. The synchronisation of a and b is given by $\Delta_{a,b}(H)$, defined below.*

$$\begin{aligned}
\Delta_{a,b}(H) &= (L, l_0, (P \backslash \{a, b\}), \mathcal{X} \cup \{x_{ab}\}, \delta_0, C, I, \rightarrow') \\
\rightarrow' &= \{\ell \xrightarrow{g,c,\omega,u,r} \ell' \mid a \notin \omega \text{ and } b \notin \omega\} \cup \\
&\quad\ \{\ell \xrightarrow{g',c,\omega',u',r} \ell' \mid \ell \xrightarrow{g,c,\omega,u,r} \ell', a \in \omega, b \in \omega, \omega' = \omega \backslash \{a, b\}, \\
&\qquad\qquad\qquad\quad g' = g[x_{ab}/\hat{a}][x_{ab}/\hat{b}],\ u' = u[x_{ab}/\hat{a}][x_{ab}/\hat{b}]\}
\end{aligned}$$

where $g[x/y]$ and $u[x/y]$ are the logic guard g and the update u that result from replacing all occurrences of variable y with x, respectively.

The composition of two THA consists of their product followed by the synchronisation of the connected ports.

▶ **Definition 10** (Hub Composition (⋈))**.** *Let H_1 and H_2 be two Hub Automata with disjoint sets of ports and variables, and let $\{(a_0, b_0), \dots, (a_n, b_n)\}$ be a finite (possibly empty) set of ports bindings, such that for each pair (a_i, b_i) for $0 \leq i \leq n$ we have that $(a_i, b_i) \in P_{I_{H_1}} \times P_{O_{H_2}}$ or $(a_i, b_i) \in P_{O_{H_1}} \times P_{I_{H_2}}$. The composition of H_1 and H_2 over such a set is defined as follows.*

$$H_1 \bowtie_{(a_0,b_0),\dots,(a_n,b_n)} H_2 = \Delta_{a_0,b_0} \dots \Delta_{a_n,b_n}(H_1 \times H_2)$$

Intuitively, composing two automata (⋈) means putting them in parallel (×), and then restricting their behaviour by forcing shared ports to go together (Δ). The first step joins concurrent transition into new transitions, placing updates in parallel. This emphasises the need for a serialisation process that guarantees a correct evaluation order of values to data in ports, which is the focus of Section 4.3. Notice that the composition of THA can generate a **non-serialisable** automaton, in which case the two automata are incompatible and cannot be composed.

Example Fig. 3 shows the composition of two Hub Automaton: a P-Timer, and a Duplicator with two output points. The composed automaton (right) illustrates the behaviour of the two hubs when synchronised over the actions *test* and *put*: whenever a timer is set and the buffer updated, the hub waits exactly t time units after which it must be tested simultaneously by two tasks through *get₁* and *get₂*. Both tasks will receive the stored data in the P-Timer Hub, before setting the timer to idle. Synchronised ports are removed from the composed model, and variables associated to such ports are renamed accordingly, i.e. $\widehat{test}$ and $\widehat{put}$, are both renamed to $x_{\widehat{test\text{-}put}}$.

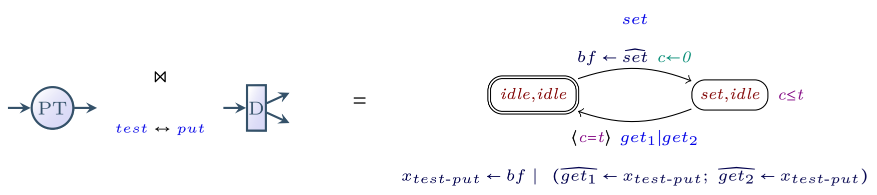


Figure 3 Example of composition between two Hub Automata, where a P-Timer automaton is composed with a Duplicator automaton by synchronising on actions *test* and *put* (left), resulting in the composed automaton on the right.

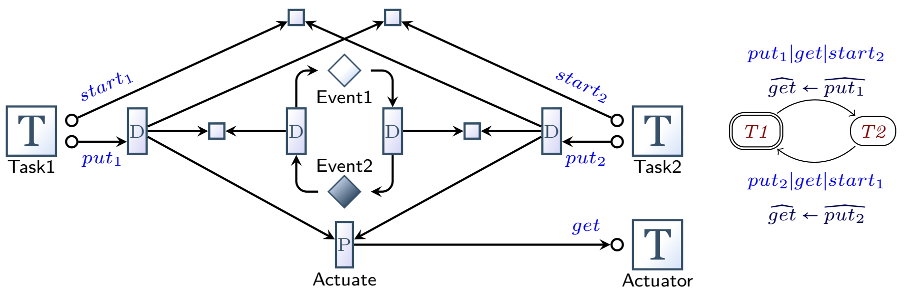


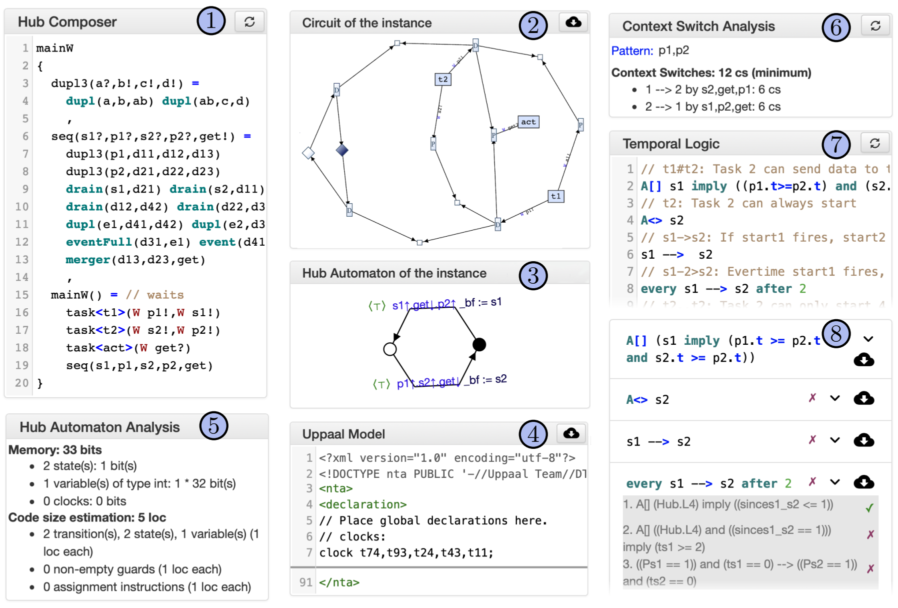
Figure 4 Alternative architecture for the example in Fig. 1 – Reo connector (left) and its THA (right) after updates have been serialised and simplified.

4.2 Example: Round Robin tasks

Consider the example architecture in Fig. 1, consisting of 3 independent hubs. Such architectures with independent hubs can be combined into a single hub, but it brings little or no advantage because it will produce all possible interleavings and state combinations. In this case, the joint automaton has 1 state and 26 transitions, representing the possible non-empty combinations of transitions from the 3 hubs. More concretely, the set of transitions is the union of the 5 sets below, abstracting away data, where p_i, s_i and t_i denote the *put*, *signal* and *test* actions of task i, respectively, and g denotes the *get* action of the actuator.

$$\begin{aligned} \mathtt{P} &= \{p_1|g\,,\ p_2|get\} \\ \mathtt{A}\|\mathtt{B} &= \{s_1, s_2, t_1, t_2\} \end{aligned} \qquad \begin{aligned} \mathtt{A\&B} &= \{x_1|x_2 \ \mid\ x_1 \in \{s_1,t_1\}, x_2 \in \{s_2,t_2\}\} \\ \mathtt{P\&A}\|\mathtt{B} &= \{p_i|g|x \ \mid\ i \in \{1,2\}, x \in \mathtt{A}\|\mathtt{B}\} \\ \mathtt{P\&A\&B} &= \{p_i|g|x \ \mid\ i \in \{1,2\}, x \in \mathtt{A\&B}\} \end{aligned}$$

We propose an alternative hub that exploits shared ports, depicted in Fig. 4, built by composing a set of primitives from Fig. 2. The primitive ◆ has the same behaviour as ◇ but its initial state is *true*. This alternative hub extracts the coordination protocol from the tasks and places it into the hub. I.e., tasks in the original hub are responsible to use the semaphores and the actuator in the right order to have an alternating behaviour; in the new hub they alternate between starting and placing a value, unaware of the coordination protocol. In this alternative hub, when a task sends a data value to the actuator, the coordinator interprets it as the end of its round. Furthermore, it requires each task to send only when the other is ready to start – a different behaviour could be implemented to buffer the end of a task round (as in Fig. 1) by modifying the hub, without modifying the tasks.



```
mainW
{
  dupl3(a?,b!,c!,d!) =
    dupl(a,b,ab) dupl(ab,c,d)
    ,
  seq(s1?,p1?,s2?,p2?,get!) =
    dupl3(p1,d11,d12,d13)
    dupl3(p2,d21,d22,d23)
    drain(s1,d21) drain(s2,d11)
    drain(d12,d42) drain(d22,d3
    dupl(e1,d41,d42) dupl(e2,d3
    eventFull(d31,e1) event(d41
    merger(d13,d23,get)
    ,
  mainW() = // waits
    task<t1>(W p1!,W s1!)
    task<t2>(W s2!,W p2!)
    task<act>(W get?)
    seq(s1,p1,s2,p2,get)
}
```

Pattern: p1,p2

Context Switches: 12 cs (minimum)
- 1 --> 2 by s2,get,p1: 6 cs
- 2 --> 1 by s1,p2,get: 6 cs

```
// t1#t2: Task 2 can send data to t
A[] s1 imply ((p1.t>=p2.t) and (s2.
// t2: Task 2 can always start
A<> s2
// s1->s2: If start1 fires, start2
s1 -->  s2
// s1-2>s2: Evertime start1 fires,
every s1 --> s2 after 2
```

```
A[] (s1 imply (p1.t >= p2.t
and s2.t >= p2.t))
```

`A<> s2` ✗

`s1 --> s2` ✗

`every s1 --> s2 after 2` ✗

1. A[] (Hub.L4) imply ((sinces1_s2 <= 1)) ✓
2. A[] ((Hub.L4) and ((sinces1_s2 == 1))) imply (ts1 >= 2) ✗
3. ((Ps1 == 1)) and (ts1 == 0) --> ((Ps2 == 1)) and (ts2 == 0) ✗

Memory: 33 bits
- 2 state(s): 1 bit(s)
- 1 variable(s) of type int: 1 * 32 bit(s)
- 0 clocks: 0 bits

Code size estimation: 5 loc
- 2 transition(s), 2 state(s), 1 variable(s) (1 loc each)
- 0 non-empty guards (1 loc each)
- 0 assignment instructions (1 loc each)

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!DOCTYPE nta PUBLIC '-//Uppaal Team//DT
3 <nta>
4 <declaration>
5 // Place global declarations here.
6 // clocks:
7 clock t74,t93,t24,t43,t11;

91 </nta>
```

Figure 5 Screenshot of the widgets in the online analyser for VirtuosoNext's hubs.

4.3 Serialisation of Updates

The application of an update u (Def. 6) requires a serialisation function σ that converts an update with parallel constructs into a sequence of assignments. This function, described in detail in [2], preserves dependencies between variables and rejects updates that have variables with circular dependencies. Later, it discards intermediate and unnecessary assignments.

For example, consider the transition $(set, idle) \xrightarrow{c=t, get_1|get_2, u_1|u_2} (idle, idle)$ from Fig. 3, where $u_1 = x_{test\text{-}put} \leftarrow bf$ and $u_2 = \widehat{get_1} \leftarrow x_{test\text{-}put}; \widehat{get_2} \leftarrow x_{test\text{-}put}$. Here, u_2 depends on a variable produced by u_1. Thus, a serialisation of $u_1|u_2$ is $u_s = u_1; u_2$. Once serialised, u_s has an *intermediate assignment*, $x_{test\text{-}put} \leftarrow bf$, which can be removed by replacing appearances of $x_{test\text{-}put}$ with bf, leading to $\widehat{get_1} \leftarrow bf; \widehat{get_2} \leftarrow bf$, reducing the number of assignments and variables needed.

Unnecessary assignments can be assignments to internal variables that are never consulted or assignments that depend only on undefined variables. They arise when connecting hubs that support data passing with hubs that do not. For example, consider the hub in Fig. 3 with an Event instead of the P-Timer Hub. After serialization, the resulting automaton has two transitions: $\ell_0 \xrightarrow{raise} \ell_1$ and $\ell_1 \xrightarrow{get_1|get_2, \widehat{get_1} \leftarrow x_{put\text{-}test}; \widehat{get_2} \leftarrow x_{put\text{-}test}} \ell_0$. The updates on the latter depend on an undefined variable, $x_{put_1\text{-}test}$, and are discarded. Similarly, connecting a two entry Port Hub to the input of the Event Hub produces updates with unused variables.

5 Online verification tools

We implemented a prototype that composes, simplifies, analyses, and verifies THA, available to use online or download.[1] Fig. 5 depicts the widgets available in the prototype. The generated automata can be used to produce either new hubs or dedicated tasks that perform coordination, which we address in Section 6.

[1] http://arcatools.org/hubs

These generated automata can also be formally analysed to provide key insight information regarding the usefulness and drawbacks of such a hub, and their behaviour can be formally verified. Our current implementation allows specifications of composed hubs and tasks using a textual representation based on ReoLive [9, 10] (top left part of Fig. 5), and produces:

(1) the editor to specify the hub;
(2) the architectural view of the hub;
(3) the simplified automaton of the hub;
(4) the timed automaton to be imported by Uppaal model checker;
(5) a summary of some structural properties of the automaton, such as required memory, size estimation of the code, information about which hubs' ports are always ready to synchronise;
(6) an interactive panel to produce the minimum number of context switches for a given trace; and
(7) an interactive panel to verify a list of given timed behavioural properties, relying on Uppaal running on our servers, and their result (8) together with the associated Uppaal models and formulas.

The software is developed in Scala, an object-oriented programming language with functional features [11]. The code is compiled both into JVM binaries that are executed on a server, and into JavaScript using ScalaJS[2] to produce the interactive web page. Note that currently the server is only used to model-check properties using Uppaal, and everything else is computed by the browser using the generated JavaScript libraries.

The rest of this section describes how tasks are abstracted and specified in our formal framework (Section 5.1), presents a temporal logics fine-tuned to THA to specify timed properties (Section 5.2), illustrates how to verify several properties of our running example under different scenarios (Section 5.3), and describes an encoding of formulas and hubs into Uppaal's temporal logic and timed automata, respectively (Section 5.4).

5.1 Tasks

Tasks in our implementation denote *contracts* capturing the order and time bounds of the expected interactions of task components. These are modelled as THA, extended with a notion of priority supported by Uppaal, and are used to describe *scenarios* of our hubs. When verifying if the architecture in Fig. 1 deadlocks, tasks can be used to specify a scenario, e.g., where Task1 and Task2 execute periodically every 10ms, and the Actuator executes periodically every 2ms.

Contracts for tasks can be specified by the following grammar.

$$tk := \texttt{task<}name\texttt{>(}port^*\texttt{)}\ [\texttt{every}\ n] \qquad mode := \texttt{W} \mid \texttt{NW} \mid n$$
$$port := mode\ name\ io \qquad io := \texttt{!} \mid \texttt{?}$$

For example, `task<T1>(W a?, 4 b!)` specifies a task that tries to read a value on its port `a`, waiting indefinitely (`W`), followed by a call to write a value to port `b` with a timeout of 4 time units, after which it loops again following the same behaviour forever . This example, when extended with `every 5`, will periodically run every 5 time units. In our interpretation of a periodic run, every round of the execution of this task takes exactly 5 time units, and repeats forever. In each round a fires once and b either fires of times-out; hence a can wait at most 10 time units between 2 fires (when it fires at the beginning and end of consecutive rounds). If after 5 time units after the start

[2] https://wwws.scala-js.org

of a round `a` fires and `b` cannot fire, then `b` will timeout and not fire for that round. As another example, `task<T2>(NW c!) every 5` will periodically try to send a value to port `c` every 5 time units, without waiting when it fails to fire. After 5 time units from the beginning of a round, if `c` did not fire then it will either fire or timeout, giving priority to firing. These three examples produce the THA in Fig. 6.

Observe that some edges have dashed lines; these capture a notion of *priority*: edges with dashed lines have lower priority, to avoid failing to communicate when their connected hubs allow communication. Intuitively, since THA are non-deterministic, we want to avoid the execution of a timeout transition if there exists as well an enabled transition were the automaton can perform some action. This notion of priority is not included in our semantics, but it exists in UPPAAL, and we will exploit it during verification. This will be further detailed in Section 5.4.

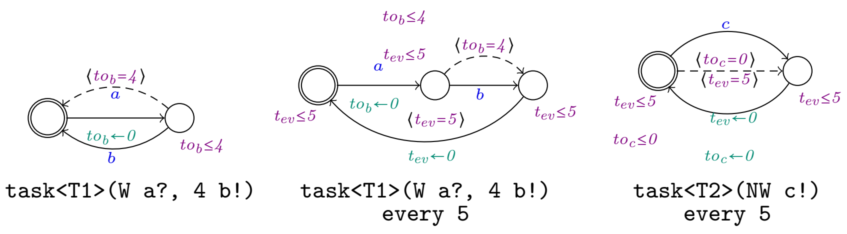


Figure 6 Timed hub automata of specific tasks.

5.2 Temporal logic for THA

This section proposes a subset of Timed Computation Tree Logic (TCTL) for timed hub automata. This logic can be seen as a subset of UPPAAL TCTL, agnostic of locations, with new operators to reason about the behaviour of the systems focused on **actions**, i.e., on ports that are fired. We proposed a concrete syntax that closely follows that used by UPPAAL's model checker, and define its semantics by formalising its satisfaction relation. Section 5.4 provides more details on the mapping from the proposed TCTL subset into UPPAAL's TCTL, and describes how it is implemented by our online prototype.

TCTL properties are described using *path formulas* and *state formulas*. A path formula quantifies over paths of the underlying transition system, while a state formula quantifies over a single state of such system. The syntax and semantics of TCTL properties are formalised below.

▶ **Definition 11** (TCTL for THA). *A valid property over a THA consists of a* path formula π *given by the following grammar*

$$\pi ::= \texttt{A}\boxtimes\ \psi \mid \texttt{E}\boxtimes\ \psi \mid \psi\ \texttt{->}\ \psi \mid \texttt{every}\ a\ \texttt{->}\ b\ [\texttt{after}\ n] \qquad \text{(path-formula)}$$

$$\psi ::= \rho \mid cc \mid g \mid \texttt{not}\ \psi \mid \psi\ \texttt{and}\ \psi \mid \texttt{deadlock} \qquad \text{(state-formula)}$$

where $a, b \in \mathcal{P}$ are ports, $n \in \mathbb{N}$, $\boxtimes \in \{\Box, \Diamond\}$, cc is a clock constraint, g is a guard, and ρ is defined as follows

$$\rho ::= a.\texttt{doing} \mid a\ \texttt{refiresAfter}\ n \mid a\ \texttt{refiresAfterOrAt}\ n \qquad (a\text{-formula})$$

Informally, *state properties* describe what must hold for a given state (i.e., location and clock valuation), and *path properties* describe what must hold while evolving the automaton. For example, a.`doing` holds if a was the last port to be fired, and a `refiresAfterOrAt` 5 holds in states where, if a fired before, then it cannot refire unless 5 units of time have passed. Regarding path properties,

$\texttt{A}\boxdiamond\ \psi$ holds if $\boxdiamond\ \psi$ holds for all possible paths, while its $\texttt{E}$ counterpart holds if $\boxdiamond\ \psi$ holds for some path. Along an execution path p, $\Box\ \psi$ holds if ψ holds for all states along p, $\Diamond\ \psi$ holds if a state along p satisfies ψ, and ψ_1 `->` ψ_2 is a shorthand for $\texttt{A}\Box(\psi_1\ \texttt{imply}\ (\texttt{A}\Diamond\ \psi_2))$.[3] Finally, `every` a `->` b `after` 5 holds if, whenever a fires, b will fire before a fires again, after 5 or more time units.

Recall the extension for verification in page 8, using t_a to capture the time since a last fired, and $done_a$ to capture if a has been performed. We now enrich our syntax with syntactic sugar for state formulas, summarised below. We write $\bigwedge$ to indicate the generalised `and` for multiple state formulas.

$$\begin{array}{rclrcl}
a.\texttt{t} & = & t_a & a.\texttt{done} & = & done_a \\
a & = & a.\texttt{doing and } a.\texttt{t} == 0 & \psi_1\ \texttt{imply}\ \psi_2 & = & \texttt{not}\ \psi_1\ \texttt{or}\ \psi_2 \\
\psi_1\ \texttt{or}\ \psi_2 & = & \texttt{not}\ (\texttt{not}\ \psi_1\ \texttt{and not}\ \psi_2) & a\ \texttt{refiresBefore}\ n & = & a.\texttt{t} < n \\
\texttt{nothing} & = & \bigwedge_{a\in P}\ \texttt{not}\ a.\texttt{doing} & a\ \texttt{refiresBeforeOrAt}\ n & = & a.\texttt{t} \le n
\end{array}$$

Satisfaction of TCTL Given a path formula π, a THA H, and a state $s = (\ell, \delta, \eta, \alpha)$ we write $H, s \vDash \pi$ to denote that π is satisfied by H in state s.

Notation Given a state $s = (\ell, \delta, \eta, \alpha)$, we write $s.\ell$, $s.\delta$, $s.\eta$, and $s.\alpha$ to denote the respective components. Given a path p and a number $i \in \mathbb{N}$, we write p_i to denote the i-th state of p.

The definition of $H, s \vDash \pi$ is presented in Fig. 7. The core auxiliary function ***Path***, which we only describe informally, receives an automata H and a state s and returns a set ps of all possible paths in H from s. In turn, each of these paths $p \in ps$ is a (possibly infinite) sequence $s_1, s_2, \ldots$ of states, following the semantic rules from Def. 7.

5.3 Example: Verifying the sequencer protocol

Recall our running example illustrated in Fig. 4 of a sequencer protocol. We illustrate the proposed specification constructs for tasks and time-sensitive behavioural properties by verifying different properties under different scenarios, i.e., connecting tasks with different models of interaction to the hub. The goal is to provide some insight on how to use our tools to understand the different expected behaviours of a hub in different scenarios.

We create 5 different scenarios with 2 producer tasks and an actuator task, varying on how the producer tasks interact with the hub. More specifically, using wait, non-wait, and timeout calls to the hubs, at different periodicities. These scenarios are presented in the left column of Table 2. On the right of the same table we list 6 different properties that we find of relevance, and whether these are satisfied under each scenario. These properties are described below, together with a discussion regarding their satisfaction on the scenarios.

$\psi_{t1\#t2} = \{\texttt{A}\Box\ \boldsymbol{start_1}\ \texttt{imply}\ ((\boldsymbol{put_1}.\texttt{t} \ge \boldsymbol{put_2}.\texttt{t})\ \texttt{and}\ (\boldsymbol{start_2}.\texttt{t} \ge \boldsymbol{put_2}.\texttt{t}))\}$

Task 1 can start only if Task 2 was the last one to run, and when Task 2 is not running (or just finishing).

This is a core functional requirement of the hub: guaranteeing exclusivity. All scenarios satisfy this property, as desired, supporting the claim that the hub is successfully imposing this requirement.

[3] As in UPPAAL, nested path formulas are not supported explicitly. However, some are introduced through specific constructs like ψ `-->` ψ.

$H, s \vDash a.\texttt{doing}$	if $a \in s.\alpha$
$H, s \vDash a\ \texttt{refiresAfter}\ n$	if $\big(s.\delta(done_a)$ and $s \xrightarrow{\delta_{io}} s'$ and $a \in \mathsf{dom}(\delta_{io})\big)$ then $s.\eta(a.\mathtt{t}) > n$
$H, s \vDash a\ \texttt{refiresAfterOrAt}\ n$	if $\big(s.\delta(done_a)$ and $s \xrightarrow{\delta_{io}} s'$ and $a \in \mathsf{dom}(\delta_{io})\big)$ then $s.\eta(a.\mathtt{t}) \geq n$
$H, s \vDash cc$	if $s.\eta \vDash cc$
$H, s \vDash g$	if $s.\delta \vDash g$
$H, s \vDash \texttt{not}\ \psi$	if $H, s \nvDash \psi$
$H, s \vDash \psi_1\ \texttt{and}\ \psi_2$	if $H, s \vDash \psi_1$ and $H, s \vDash \psi_2$
$H, s \vDash \texttt{deadlock}$	if $\forall_{t \in \mathbb{R}_{\geq 0}} : \forall_{\delta_{io}} : \forall_{s', s''} :$ not $s \xrightarrow{t} s' \xrightarrow{\delta_{io}} s''$
$H, s \vDash \texttt{A}\Box\ \psi$	if $\forall_{p \in \boldsymbol{Path}(H,s)} : \forall_{s' \in p} : H, s' \vDash \psi$
$H, s \vDash \texttt{A}\Diamond\ \psi$	if $\forall_{p \in \boldsymbol{Path}(H,s)} : \exists_{s' \in p} : H, s' \vDash \psi$
$H, s \vDash \texttt{E}\Box\ \psi$	if $\exists_{p \in \boldsymbol{Path}(H,s)} : \forall_{s' \in p} : H, s' \vDash \psi$
$H, s \vDash \texttt{E}\Diamond\ \psi$	if $\exists_{p \in \boldsymbol{Path}(H,s)} : \exists_{s' \in p} : H, s' \vDash \psi$
$H, s \vDash \psi_1\ \texttt{->}\ \psi_2$	if $H, s \vDash \texttt{A}\Box(\psi_1\ \texttt{imply}\ (\texttt{A}\Diamond\ \psi_2))$ (Note: this is an abuse of notation for simplicity.)
$H, s \vDash \texttt{every}\ a\ \texttt{->}\ b$	if $\left\{ \begin{array}{l} \forall_{p \in \boldsymbol{Path}(H,s)} : \forall_{i \in \mathbb{N}} : \text{if } a \in p_i.\alpha \text{ then} \\ \quad \exists_{j \geq i} : \big(b \in p_j.\alpha \text{ and } \forall_{i < k \leq j} : a \notin p_k.\alpha\big) \end{array} \right.$
$H, s \vDash \texttt{every}\ a\ \texttt{->}\ b\ \texttt{after}\ n$	if $\left\{ \begin{array}{l} \forall_{p \in \boldsymbol{Path}(H,s)} : \forall_{i \in \mathbb{N}} : \text{if } a \in p_i.\alpha \text{ then} \\ \quad \exists_{j \geq i} : \big(b \in p_j.\alpha \text{ and } (\forall_{i < k \leq j} : a \notin p_k.\alpha) \text{ and} \\ \qquad p_j.\eta(a.\mathtt{t}) \geq n \text{ and } \forall_{i \leq k < j} : b \notin p_k.\alpha\big) \end{array} \right.$

Figure 7 Satisfaction of TCTL formulas

Table 2 Verification of the sequencer hub under different scenarios.

Scenario	$\psi_{t1\#t2}$	ψ_{t2}	$\psi_{s1\rightarrow s2}$	$\psi_{s1 \overset{2}{\rightarrow} s2}$	$\psi_{t2\cdots t2}$	$\psi_{\leq 9}$

`task<T1>(W put!,W st! )`
`task<T2>(W st!, W put!)`
`task<Ac>(W get)`
	✓	✗	✗	✗	✗	✗

`task<T1>(W put!,W st! ) every 3`
`task<T2>(W st!, W put!) every 3`
`task<Ac>(W get)`
	✓	✓	✓	✗	✗	✓

`task<T1>(NW put!,NW st! ) every 3`
`task<T2>(NW st!, NW put!) every 3`
`task<Ac>(W get)`
	✓	✓	✓	✓	✓	✓

`task<T1>(3 put!,3 st! ) every 6`
`task<T2>(3 st!, 3 put!) every 6`
`task<Ac>(W get)`
	✓	✓	✓	✗	✗	✓

`task<T1>(NW put!,3 st! ) every 2`
`task<T2>(W st!, 3 put!) every 3`
`task<Ac>(W get)`
	✓	✓	✗	✗	✓	✓

$\psi_{t2} = \{\texttt{A} \Diamond\ \textit{start}_2\}$

Task 2 must start eventually.

This is a liveness property, to check Task 2 must run. Only the first scenario fails to satisfy this property, because our interpretation of "`W st!`" is that the task can chose to wait as long as it wants. Hence, there is no guarantee that the Task 2 will run, when the tasks decide to wait *forever*. In the other scenarios there is a round clock ("`every`" construct) that forces actions to be taken within a time bound, and timeouts or failures to communicate will still allow Task 2 to run because they have lower priority than having communication.

$\psi_{s1\to s2} = \{\textit{start}_1\ \texttt{->}\ \textit{start}_2\}$

If $\textit{start}_1$ fires, $\textit{start}_2$ must eventually fire.

This is a stronger liveness property than the previous one, that is a requirement to have continuous progress. Only two scenarios fail to satisfy this property: the first one, because it allows eternal waiting after firing $\textit{start}_1$, and the last one, because it **deadlocks**. This deadlock is not clear at first sight: the tasks can successfully finish a full round, where Task 1 will finish first. Once Task 1 restarts a new round, it will timeout the port $\textit{put}_1$, leading to a state where both Task 1 and Task 2 are waiting to fire *start*, without a timeout option. At the end of the round of Task 1 the system cannot evolve: it cannot fire any *start* port and time cannot pass (due to the "`every`" construction).

$\psi_{s1\xrightarrow{2}s2} = \{\texttt{every}\ \textit{start}_1\ \texttt{->}\ \textit{start}_2\ \texttt{after}\ 2\}$

Everytime $\textit{start}_1$ fires, $\textit{start}_2$ must eventually fire before $\textit{start}_1$ again, and wait at least 2 time units before firing $\textit{start}_2$.

This is a variation of the previous property, with the key difference of a mandatory waiting period. The choice of $\textit{start}_1$ on the left and $\textit{start}_2$ on the right side is to capture the change of rounds (by the `every` construct). Consequently, all scenarios fail to satisfy this property with the exception of 3rd scenario: the 1st scenario fails because rounds can be faster than 2; the 2nd and 4th scenarios fail because $\textit{start}_1$ can be executed at the end of a round, and $\textit{put}_2$ at the beginning of the following round; the last scenario fails for the same reason $\psi_{s1\to s2}$ does.

$\psi_{t2\ldots t2} = \{\texttt{A}\Box\ \textit{start}_2\ \texttt{imply}\ (\textit{put}_2.\texttt{t} \geq 2)\ \texttt{or}\ \texttt{not}(\textit{put}_2.\texttt{done})\}$

Task 2 can only start 2 time units after finishing a previous round.

This property checks if Task 2 has a minimum time delay between rounds. Only the 3rd and the 5th scenario satisfy this property – the others allow Task 2 to fire $\textit{put}_2$ at the end of a round, and immediately fire $\textit{start}_2$ at the beginning of the next round. Note that, if we modify the scenario with timeouts by either reducing the timeout or by increasing the periodicity, this property will already hold – as long as the latest time $\textit{put}_2$ can fire is at least 2 time units before the end of the round.

$\psi_{\leq 9} = \{\texttt{A}\Box\ \textit{start}_2\ \texttt{refiresBeforeOrAt}\ 9\}$

Task 2 starts within 9 time units after finishing a previous round.

This property checks an upper bound for the longest time it can wait between firing $\textit{start}_2$ twice. Only the first scenario fails, since it can take an infinite amount of time between two actions. For example, the 2nd scenario can take up to 6 time units between executions, if $\textit{start}_2$ fires at the beginning of a round, and at the end of the follow up round (hence $3 + 3$ time units). The 4th scenario can take the longest: up to 9 time units, if $\textit{start}_2$ fires at the beginning of a round, and in the follow up round fires right before timing out (hence $6 + 3$ time units).

Observations The selected set of scenarios and properties above illustrate the style of properties that one can verify using our logic and tasks. An important observation is that *tasks can decide to wait*, even if the hub is ready to communicate, although *tasks can be forced to communicate* when a timeout is reached or at the end of a round. This was a design decision based on the more traditional semantics of timed automata. Alternatively, we could make the tasks to communicate *as soon as possible*, shifting the decision of waiting to the hubs; technically, this would involve introducing a notion of *urgent* action to our semantics, already supported by UPPAAL. For simplicity, we decided to leave this for future work.

Another relevant observation is that the firing of ports takes zero time in our model, based on timed automata. Hence, in any of our scenarios, it is possible to run a full round in zero time. Furthermore, a possible trace in the first scenario is an infinite stream of communication without time passing, known in the literature as a Zeno path, which should be avoided. Our notion of periodicity provides some control over forcing time to evolve, but other mechanisms could be added, such as introducing time delays between actions, or requiring each port to take some amount of time to fire.

5.4 Under the hood: verification via Uppaal

This subsection describes how we verify THA using UPPAAL. More precisely, how it proposes an encoding of THA as UPPAAL's timed automata, and an encoding of TCTL formulas for THA to UPPAAL's TCTL. The *automata encoding* introduces new data variables and clocks to reason about which ports have been fired, and new intermediate locations to distinguish when an action is about to fire from when it actually fires. The *TCTL encoding* converts the references to ports into references to locations or to the new variables, following closely the notion of satisfaction of TCTL described in Section 5.2.

Note that this subsection is not as formal and detailed as the previous ones, since we neither formalise UPPAAL's semantics nor present a proof of correctness of the encodings. Instead, we guide the reader throw key examples to describe the encodings. This choice is mainly due to the large similarity of the syntaxes of automata and formulas, and because this paper focuses more on tools than on theory, i.e., on providing practical machinery to help developers analysing and verifying hubs.

5.4.1 Encoding Automata by Example

Recall the timer hub connected to the duplicator hub, depicted in Fig. 3. Its encoding into a timed automata in UPPAAL is illustrated in Fig. 8, which introduces new *locations*, *clocks*, and *data variables*. These includes, for each port a, the clock t_a and variable $done_a$, introduced in Section 3.2. The resulting automata discards data information because the existing tools are focused on timed properties, but in the future we plan to support UPPAAL's data operations in our encodings.

Locations are depicted with dashed lines, and are associated to sets of ports that triggered them. These are marked as *committed locations* in UPPAAL, meaning that they do not allow time to proceed, and have priority over non-committed locations to proceed. Hence, to know if *set* has just been fired, one can check if the automata is in any of these special committed locations associated to the *set* port.

Data variables (x) Every port a yields a variable x_a, set to $\top$ when port a was fired in the last set of fired ports, I.e., it represents whether $a \in \boxed{\alpha}$.

Data variables ($since$) Every pair of different ports (a, b) yields a variable $since_{a,b}$, which is a number between 0 and 2 (considering that $2 + 1 = 2$ and $0 - 1 = 0$), roughly denoting the number of times a fired since b was last fired. More precisely, $since_{set,get_1}$ is 0 if *set* never fired, it is 1

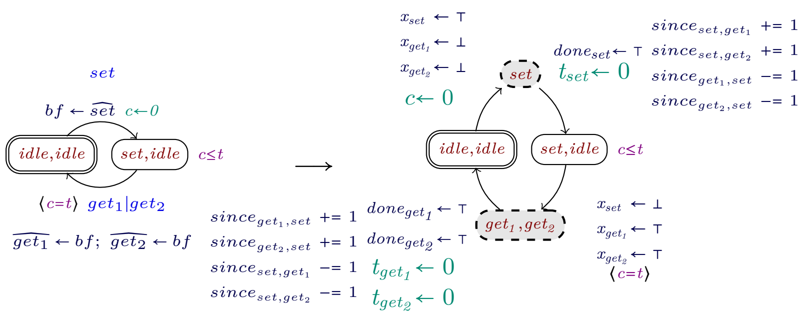


Figure 8 Encoding the simplified automata from Fig. 3 (left) into a UPPAAL's automata (right); dashed locations are *committed*.

if it was fired once since the last time get_1 was fired (or from the beginning), and it is 2 if it was fired more than once since the last time get_1 was fired. These *since* constructs are used when verifying formulas like `every` a `->` b, where for each a fired, b should fire without a firing in between.

Optimisation: Observe that there is a large number of new variables and clocks, and also a large number of extra (committed) locations. In practice we do not add all variables and extra locations, but only the ones needed by each individual rule. Hence, verifying 4 properties will generate 4 (potentially different) UPPAAL automata, each simplified to include only the needed artefacts, and including the encoded property to be verified. For simplicity, we do not present here the simpler automata versions with less variables, clocks, or locations.

Priority: Recall the notion of priority mentioned when describing tasks (Section 5.1), where we used dashed arrows in automata to depict low-priority transitions. This priority is meant only to avoid ports from discarding data and timeout when the hub is ready to communicate. This is encoded in UPPAAL using its notion of *channel priority*. *Channels* in UPPAAL are labels of transitions in automata used to synchronise with channels of neighbour automata. Our encoding does not rely on channels since it produces a single automaton, but we introduce here a set of dummy channels $prio_p$ that can always be fired,[4] where $p \in \mathbb{Z}$ denotes the priority of the channel (higher numbers mean higher priority). Transitions in an automaton are marked with priority 0 if it synchronizes with other automaton, and with priority −1 if it denotes a timeout. During composition, priorities of transitions that go together are added up, reducing the priority of transitions with more timeouts.

5.4.2 Encoding Formulas by Example

The UPPAAL[5] model checker supports a subset of TCTL formula for timed automata [12], which we took into account when proposing the logic for THA. Similar to the proposed logic, properties consist of *path* and *state formulas*.

▶ **Definition 12** (Uppaal TCTL). *Given a network of UPPAAL's timed automata, a valid property over the network consists of a* path formula *given by the following grammar*

$$\pi_u ::= \texttt{A}\boxed{\,}\ \psi_u \mid \texttt{E}\Diamond\ \psi_u \mid \psi_u\ \texttt{->}\ \psi_u \qquad \text{(UPPAAL path formula)}$$

$$\psi_u ::= ta.\ell \mid cc \mid g \mid \texttt{not}\ \psi_u \mid \psi_u \odot \psi_u \mid \texttt{deadlock} \qquad \text{(UPPAAL state formula)}$$

[4] This is technically achieved using a broadcast channel in UPPAAL.
[5] http://www.uppaal.org/

where $cc \in \mathcal{C}(C)$ *and* $g \in \Phi(\mathcal{X})$ *are clock constraints and guards over clocks* C *and variables*[6] $\mathcal{X}$ *known in the network;* $ta.\ell$ *represents location* ℓ *in the automaton named* ta*; and* $\odot \in \{\texttt{and}, \texttt{or}, \texttt{imply}\}$*.*

The key differences with our logic are: the use of locations ($ta.\ell$) in state formulas, the absence of references to actions (or ports) and their associated clocks, and the absence of the `every`-path formula. Hence, when encoding our logic into Uppaal's TCTL, each of the missing constructs are mimicked using the extra variables and clocks, and using references to known locations in the automata encoding.

To refer to the committed locations introduced in Section 5.4.1 we will use the following shorthand, where a is a port:

$$\mathsf{cmt}(a) = \begin{cases} \ell_1 \ \texttt{or} \ \ldots \ \texttt{or} \ \ell_n & \text{if } \{\ell_1, \ldots, \ell_n\} \text{ are the locations where } a \text{ appears;} \\ \texttt{false} & \text{otherwise.} \end{cases}$$

The encoding of examples of key formulas is presented in Table 3 – the general encoding of formulas follows the same structure as in these examples, and is omitted in this paper. This proof relies on the fact that the observable behaviour is not modified by adding new intermediate committed states to an automata that has no committed states, and by adding new variable assignments that are never read.

Table 3 Examples of encodings of THA TCTL formulas into Uppaal

TCTL	Encoding to Uppaal
$\texttt{A}\Diamond\ put_2 \ \texttt{and}\ get$	$\texttt{A}\Diamond\ x_{put_2}\ \texttt{and}\ t_{put_2} = 0\ \texttt{and}\ x_{get}\ \texttt{and}\ t_{get} = 0$
$\texttt{A}\Box\ act.\texttt{doing or nothing}$	$\texttt{A}\Box\ x_{act}\ \texttt{or}\ (\texttt{not}\ x_{get}\ \texttt{and not}\ x_{put_1}\ \texttt{and not}\ x_{put_2})$
$\texttt{every}\ put_1\ \texttt{->}\ put_2\ \texttt{after}\ 2$	$\begin{cases} x_{put_1}\ \texttt{->}\ x_{put_2} \\ \texttt{A}\Box\ \mathsf{cmt}(put_2)\ \texttt{imply}\ since_{put_1,put_2} \leq 1 \\ \texttt{A}\Box \left(\begin{array}{l} \mathsf{cmt}(put_2)\ \texttt{and} \\ since_{put_1,put_2} = 1 \end{array} \right)\ \texttt{imply}\ t_{put_1} \geq 2 \end{cases}$
$\texttt{A}\Diamond\ put_1\ \texttt{refiresAfterOrAt}\ 2$	$\texttt{A}\Diamond\ \left(done_{put_1}\ \texttt{and}\ \mathsf{cmt}(put_1)\right)\ \texttt{imply}\ t_{put_1} \geq 2$

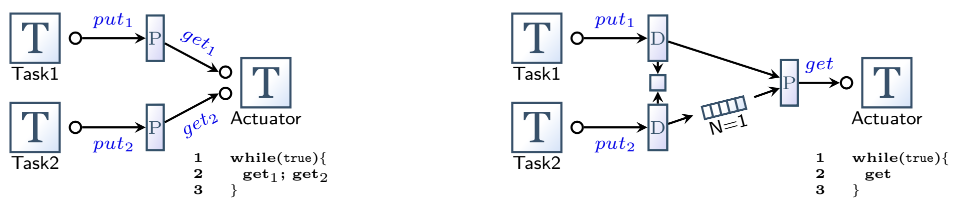
6 Executing hubs

We compare the two architectures from Section 4.2, using a variation of these, and provide both an analytical comparison, using different metrics, and a performance comparison, executing them in an embedded board. This comparison does not use hubs with timed behaviour, since VirtuosoNext does not support these, and the timed behaviour is mainly captured by tasks.

6.1 Scenarios

We compare four different scenarios in our evaluation, using the architectures from Section 4.2, and compile and execute them on a TI Launchpad EK-TM4C1294XL[7] board with a 120MHz 32-bit ARM Cortex-M4 CPU.

[6] Uppaal supports two predefined types for data variables: `int` and `bool`; and Array and record types can be defined over these types. Integers range over $[-32768, 32767]$.
[7] http://www.ti.com/tool/EK-TM4C1294XL#



```
while(true){
  get1; get2
}
```

```
while(true){
  get
}
```

Figure 9 Architectural view of scenarios S$_{2\text{-}ports}$ (left) and S$_{altern}$ (right).

- **S$_{orig}$** the initial architecture as in Fig. 1;
- **S$_{custom}$** using a custom-made hub that follows the automaton in Fig. 4 without any data transfer;
- **S$_{altern}$** using a custom-made hub that acts as S$_{custom}$, but discarding the `start` queues, and assuming that tasks start as soon as possible (Fig. 9 right); and
- **S$_{2\text{-}ports}$** simple architecture with two ports, each connecting a task to the actuator, also discarding the `start` queue, whereas the actuator is responsible to impose the alternating behaviour (Fig. 9 left).

Observe that S$_{altern}$ and S$_{2\text{-}ports}$ are meant to produce the same behaviour, but only the latter is compiled and executed. While S$_{altern}$ assumes that the actuator is oblivious of who sends the instructions, S$_{2\text{-}ports}$ relies on the actuator to perform the coordination task.

6.2 Analytic comparison

We claim that the alternative architecture requires less memory and requires less context switches (and hence is expected to execute faster). Memory can be approximated by adding up the number of variables and states. The original example uses a stateless hub (a Port) and two Semaphores, each also stateless but with an integer variable each—hence requiring the storage of 2 integers. The refined example requires 2 states and no variables (after simplification), hence a single bit is enough to encode its state.

Table 4 lists possible sequence of context switches for each of the 4 proposed scenarios, for each round where both tasks send an instruction to the actuator. Observe that S$_{orig}$ requires the most context switches for each pair of values sent (17), while S$_{2\text{-}ports}$ and S$_{altern}$ require the least (9).

Note that conceptually the original architecture further requires the tasks to be well behaved, in the sense that a task should not signal/test a semaphore more times than the other task tests/signals it. In the refined architecture functionality is better encapsulated: tasks abstract from implementing coordination behaviour and focus only on sending data to the actuator, while the coordinator handles the order in which tasks are enabled to send the data. This contributes to a better understanding of the behaviour of both the tasks and the coordination mechanism. In addition, knowing the semantics of each hub and looking at the architecture in Fig. 1 is not enough to determine the behaviour of the composed architecture, but it requires to look at the implementation of the tasks to get a better understanding of what happens. However, in Fig. 4 these two premises are sufficient to understand the composed behaviour.

6.3 Measuring execution times on the target processor

We compiled, executed, and measured the execution of 4 systems: (1) S$_{orig}$, (2) a variation of S$_{custom}$ implemented as a dedicated task, which we call $Task[S_{custom}]$, (3) a variation of S$_{custom}$ that abstracts away from the actual instructions (implemented as a native hub, which we call $NoData[S_{custom}]$), and (4) S$_{2\text{-}ports}$. The results of executing 1000 rounds using our TI Launchpad

Table 4 Possible sequence of context switches between the Kernel task (executing the hubs) and the user tasks for each scenario.

	S_{orig}	S_{custom}	$S_{2\text{-}ports}$ & S_{altern}
1	Kernel $\rightarrow$ Actuator	Kernel $\rightarrow$ Actuator	Kernel $\rightarrow$ Actuator
2	Actuator $\xrightarrow{get}$ Kernel	Actuator $\xrightarrow{get}$ Kernel	Actuator $\xrightarrow{get}$ Kernel
3	Kernel $\rightarrow$ Task2	Kernel $\rightarrow$ Task1	Kernel $\rightarrow$ Task1
4	Task2 $\xrightarrow{signalB}$ Kernel	Task1 $\xrightarrow{put}$ Kernel	Task1 $\xrightarrow{put}$ Kernel
5	Kernel $\rightarrow$ Task1	Kernel $\rightarrow$ Task2	Kernel $\rightarrow$ Actuator
6	Task1 $\xrightarrow{testB}$ Kernel	Task2 $\xrightarrow{start}$ Kernel	Actuator $\xrightarrow{get}$ Kernel
7	Kernel $\rightarrow$ Task1	Kernel $\rightarrow$ Actuator	Kernel $\rightarrow$ Task2
8	Task1 $\xrightarrow{put}$ Kernel	Actuator $\xrightarrow{get}$ Kernel	Task2 $\xrightarrow{put}$ Kernel
9	Kernel $\rightarrow$ Actuator	Kernel $\rightarrow$ Task2	Kernel $\rightarrow$ Actuator
10	Actuator $\xrightarrow{get}$ Kernel	Task2 $\xrightarrow{put}$ Kernel	(Repeat from #2)
11	Kernel $\rightarrow$ Task1	Kernel $\rightarrow$ Task1	
12	Task1 $\xrightarrow{signalA}$ Kernel	Task1 $\xrightarrow{start}$ Kernel	
13	Kernel $\rightarrow$ Task2	Kernel $\rightarrow$ Actuator	
14	Task2 $\xrightarrow{testA}$ Kernel	(Repeat from #2)	
15	Kernel $\rightarrow$ Task2		
16	Task2 $\xrightarrow{put}$ Kernel		
17	Kernel $\rightarrow$ Actuator		
18	(Repeat from #2)		

board are presented below, whereas the end of each round consists of the actuator receiving an instruction from both tasks (i.e., 500 values from each task).

	S_{orig}	$\textit{Task}[S_{custom}]$	$\textit{NoData}[S_{custom}]$	$S_{2\text{-}ports}$
Time (ms)	41.88	64.27	32.19	21.16

These numbers provide some insight regarding the cost of coordination. On one hand, avoiding the loop of semaphores can double the performance (S_{orig} vs. $S_{2\text{-}ports}$). On the other hand, replacing the loop of semaphores by a dedicated hub that includes interactions with the actuator can reduce the execution time to around 75% (S_{orig} vs. $\textit{NoData}[S_{custom}]$). Note that this dedicated hub does not perform data communication, and the tasks do not send any data in any of the scenarios. Finally, $\textit{Task}[S_{custom}]$ reflects the cost of building a custom hub as a user task, connected to the coordinated tasks using extra (basic) hubs, which can be seen as the price for the flexibility of complex hubs without the burden of implementing a dedicated hub.

7 Related work

The global architecture of VirtuosoNext RTOS, including the interaction with hubs, has been formally analysed using TLA+ by Verhulst et al. [1]. More concretely, the authors specify a set of concrete hubs, their waiting lists, and the priority of requests, and use the TLC model checker to verify a set of safety properties over these. Recently, we proposed an approach to formalise existing and new hubs through hub automata [2], focused on the interactions, abstracting away waiting lists, and aiming at the analysis of more complex hubs built compositionally. Here, we extend hub automata with time and propose a dynamic temporal logic to express temporal properties focusing on ports behaviour. By including time, we can model tasks as THA and verify how hubs behave when coordinating tasks with different timed behaviour, such as periodic tasks, tasks that timeout, and tasks that wait forever.

The automata model proposed here is inspired by Reo's parametrised constraint automata [3], constraint automata with memory cells [13], and Reo's semantics using timed automata [4, 6], which are extensions of constraint automata to reason about data-dependent and time-dependent coordination mechanisms. Parametrised constraint automata consist of symbolic representations of automata, where states can store variables, which are updated or initialised when transiting; while constraint automata with memory cells treats variables as first-class objects, as in here, allowing to efficiently deal with infinite data domains. The semantics based on timed automata provide encodings of Reo connectors using the same notion of time used by Uppaal, as we do, and further exploit the notion of automata composition embedded in Uppaal. Both data-approaches use data constraints as a way to assign values to ports, and define updates as a way to modify internal variables. Unlike these approaches, we introduce a notion of sequential and parallel updates, and invest a large effort to facilitate the verification process, by providing support for a fine-tuned language for specifying logical properties agnostic of locations and for describing timed scenarios. We avoid exposing the user to Uppaal, using a similar automata model that is better suited for multiple actions.

The composition and the restrictions imposed here on the input and output ports are similar to those introduced by Interface Automata [14] to deal with the composition of open systems. However, [14] imposes additional restrictions to ensure automata compatibility, i.e. whenever an automaton is ready to send an output, which is an input of the other, the latter should be able to receive it.

Finite-memory automata [15] and nominal automata [16, 17] are models that deal with infinite alphabets, focusing on the expressiveness of their variants and on the decidability of some of their properties, which is not the goal of this paper. Finite-memory automata use substitution instead of equality tests over the input alphabet with the support of a finite set of registers (variables) associated to the automata, and nominal automata are based on nominal sets, which can be seen as infinite sets with a finite representation.

Formal analysis of RTOS are more typically focused on the scheduler, which is not the focus of this work. For example, theorem provers have been used to analyse schedulers for avionics software [18]. Carnevali et al. [19] use preemptive Time Petri Nets to support exact scheduling analysis and guide the development of tasks with non-deterministic execution times in an RTOS with hierarchical scheduling. Dietrich et al. [20] analyse and model check all possible execution paths of a real-time system to tailor the kernel to particular application scenarios, resulting in optimisations in execution speed and robustness. Dokter et al. [21] propose a framework to synthesise optimised schedulers that consider delays introduced by interaction between tasks. Scheduling is interpreted as a game that requires minimising the time between subsequent context switches.

8 Conclusions

This paper proposes an approach to build and analyse hubs in VirtuosoNext, which are services used to orchestrate interacting tasks in a Real Time OS that runs on embedded devices. When using VirtuosoNext, programmers can orchestrate individual tasks by using a set of core hubs, provided as services by the OS. More complex interaction mechanisms must be encoded within the tasks, which is hard to debug and maintain.

Our proposed formal framework provides mechanisms to design and implement complex hubs that can be formally analysed and verified to provide the same level of assurance that predefined hubs provide. Currently, the framework allows to (1) **construct** complex hubs out of simpler ones, (2) **verify** timed properties using a variation of TCTL used by UPPAAL tailored to reason about interactions with hubs, and (3) **analyse** some aspects of the hubs such as: memory used, estimated lines of codes, always available ports, and minimum number of context switches required to perform certain behaviour. This is publicly available both to run online using our web interface, and to download and execute locally.[8]

Preliminary tests on a typical set of scenarios have confirmed our hypothesis that using dedicated hubs to perform custom coordination can result in performance improvements. In addition, we claim that moving coordination aspects away from tasks enables a better understanding of the tasks and hubs behaviour, and provides better visual feedback regarding the semantics of the system. The tools provide benefits both for users of VirtuosoNext and for Altreonic's developers. The former can use it to experiment how existing hubs behave in different timed scenarios; while the latter can use it to help designing new hubs, and can potentially incorporate it into a future version of VirtuosoNext to support custom-made hubs.

Ongoing work to extend our formal framework includes:

- **variability support** to analyse and improve the development of families of systems in VirtuosoNext, since VirtuosoNext provides a simple and error-prone mechanism to allow topologies to be applied to the same set of tasks;
- **code refactoring and generation** applied to existing (on-production) VirtuosoNext programs, probably adding new primitive hubs, by extracting the coordination logic from tasks and into new complex hubs; and
- **analysis extension** to support a wider range of analysis to Hub Automata, such as the model checking of liveness and safety properties using other tools, e.g. mCRL2 (c.f. [9, 22]).

Acknowledgements

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-029946 (DaVinci). This work is also partially supported by National Funds through FCT/MCTES, within the CISTER Research Unit (UIDB/04234/2020); by the Norte Portugal Regional Operational Programme (NORTE 2020) under the Portugal 2020 Partnership Agreement, through ERDF and also by national funds through the FCT, within project NORTE-01-0145-FEDER-028550 (REASSURE); and by the Operational Competitiveness Programme and Internationalization (COMPETE 2020) under the PT2020 Partnership Agreement, through ERDF, and by national funds through the FCT, within project POCI-01-0145-FEDER-029119 (PReFECT).

References

1 E. Verhulst, R. T. Boute, J. M. S. Faria, B. H. Sputh, V. Mezhuyev, Formal Development of a Network-Centric RTOS: software engineering for reliable embedded systems, Springer Science & Business Media, 2011. doi:10.1007/978-1-4419-9736-4.

2 G. Cledou, J. Proença, B. H. C. Sputh, E. Verhulst, Coordination of Tasks on a Real-Time OS, in: H. Riis Nielson, E. Tuosto (Eds.), Coordination Models and Languages, Springer International Publishing, Cham, 2019, pp. 250–266.

[8] http://arcatools.org/hubs

3 C. Baier, M. Sirjani, F. Arbab, J. J. M. M. Rutten, Modeling component connectors in Reo by constraint automata, Science of Computer Programming 61 (2) (2006) 75–113.

4 F. Arbab, C. Baier, F. S. de Boer, J. J. M. M. Rutten, Models and temporal logical specifications for timed component connectors, Software and System Modeling 6 (1) (2007) 59–82.

5 N. Kokash, M. M. Jaghoori, F. Arbab, From timed Reo networks to networks of timed automata, Electron. Notes Theor. Comput. Sci. 295 (2013) 11–29. doi:10.1016/j.entcs.2013.04.004.

6 G. Cledou, J. Proença, L. S. Barbosa, Composing families of timed automata, in: M. Dastani, M. Sirjani (Eds.), Fundamentals of Software Engineering - 7th International Conference, FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers, Vol. 10522 of Lecture Notes in Computer Science, Springer, 2017, pp. 51–66. doi:10.1007/978-3-319-68972-2_4.

7 A. NV, OpenComRTOS-Suite Manual and API Manual (1.4.3.3), http://www.altreonic.com/sites/default/files/OpenComRTOS_API-Manual.pdf.

8 R. Alur, D. L. Dill, A theory of timed automata, Theoretical Computer Science 126 (2) (1994) 183 – 235. doi:http://dx.doi.org/10.1016/0304-3975(94)90010-8.

9 R. Cruz, J. Proença, Reolive: Analysing connectors in your browser, in: M. Mazzara, I. Ober, G. Salaün (Eds.), Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers, Vol. 11176 of Lecture Notes in Computer Science, Springer, 2018, pp. 336–350. doi:10.1007/978-3-030-04771-9_25.

10 J. Proença, A. Madeira, Taming hierarchical connectors, in: Fundamentals of Software Engineering - 8th International Conference, FSEN 2019, Tehran, Iran, 2019, Lecture Notes in Computer Science (to appear), 2019.

11 M. Odersky, L. Spoon, B. Venners, Programming in scala, Artima Inc, 2008.

12 G. Behrmann, A. David, K. G. Larsen, A Tutorial on Uppaal, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 200–236. doi:10.1007/978-3-540-30080-9_7.

13 S.-S. Jongmans, T. Kappé, F. Arbab, Constraint automata with memory cells and their composition, Science of Computer Programming 146 (2017) 50 – 86, special issue with extended selected papers from FACS 2015. doi:https://doi.org/10.1016/j.scico.2017.03.006.

14 L. de Alfaro, T. A. Henzinger, Interface-Based Design, Springer Netherlands, Dordrecht, 2005, pp. 83–104. doi:10.1007/1-4020-3532-2_3.

15 M. Kaminski, N. Francez, Finite-memory Automata, Theor. Comput. Sci. 134 (2) (1994) 329–363. doi:10.1016/0304-3975(94)90242-9.

16 A. Kurz, T. Suzuki, E. Tuosto, On Nominal Regular Languages with Binders, in: L. Birkedal (Ed.), Foundations of Software Science and Computational Structures, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 255–269.

17 L. Schröder, D. Kozen, S. Milius, T. Wißmann, Nominal Automata with Name Binding, in: Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, 2017, pp. 124–142. doi:10.1007/978-3-662-54458-7_8.

18 V. Ha, M. Rangarajan, D. Cofer, H. Rues, B. Dutertre, Feature-Based Decomposition of Inductive Proofs Applied to Real-Time Avionics Software: An Experience Report, in: Proceedings of the 26th International Conference on Software Engineering, ICSE '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 304–313.
URL http://dl.acm.org/citation.cfm?id=998675.999435

19 L. Carnevali, G. Lipari, A. Pinzuti, E. Vicario, A Formal Approach to Design and Verification of Two-Level Hierarchical Scheduling Systems, in: A. Romanovsky, T. Vardanega (Eds.), Reliable Software Technologies - Ada-Europe 2011, Springer, Berlin, Heidelberg, 2011, pp. 118–131.

20 C. Dietrich, M. Hoffmann, D. Lohmann, Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis, ACM Trans. Embed. Comput. Syst. 16 (2) (2017) 35:1–35:25. doi:10.1145/2950053.

21 K. Dokter, S.-S. Jongmans, F. Arbab, Scheduling Games for Concurrent Systems, in: A. Lluch Lafuente, J. Proença (Eds.), Coordination Models and Languages, Springer, Cham, 2016, pp. 84–100.

22 N. Kokash, C. Krause, E. P. de Vink, Reo + mCRL2: A framework for model-checking dataflow in service compositions, FAC 24 (2) (2012) 187–216.