Hubs for VirtuosoNext: Online Verification of Real-Time Coordinators

Guillermina Cledou[a], José Proença[b], Bernhard H.C. Sputh[c], Eric Verhulst[c]

[a] *HASLab/INESC TEC, Universidade do Minho, Portugal*
[b] *CISTER, ISEP, Portugal*
[c] *Altreonic NV, Belgium*

Abstract

VirtuosoNext™ 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. 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.

This work presents a toolset that supports the building of new services compositionally, using notions borrowed from the Reo coordination language, on which the developer can delegate coordination-related duties. This toolset uses a formal compositional semantics for hubs that captures dataflow and time, formalising the behaviour of existing hubs, and allowing the definition of new ones. Furthermore, it enables the analysis and verification of hubs under our automata interpretation, including time-sensitive behaviour via the Uppaal model checker, usable on http://arcatools.org/hubs. We illustrate the proposed tools and methods by verifying key properties on different interaction scenarios between tasks and a composed hub.

Keywords: Coordination, Uppaal, Real-time OS, Compositional semantics

Email addresses: mgc@inesctec.pt (Guillermina Cledou), pro@isep.ipp.pt (José Proença), bernhard.sputh@altreonic.com (Bernhard H.C. Sputh), bernhard.sputh@altreonic.com (Eric Verhulst)

Preprint accepted in Science of Computer Programming
https://doi.org/10.1016/j.scico.2020.102566 *October 20, 2020*

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.

Towards addressing these limitations, this paper proposes a framework to guide *users of VirtuosoNext* to analyse different hubs and scenarios, and *Altreonic's developers* to help designing hubs that can be included in future versions of VirtuosoNext. This framework supports the specification, composition and analysis of hubs and timed contracts of tasks based on Timed Automata. For example, we can write `{task<t1>(W s!) semaphore(s,t) task<t2>(2 t?) every 3}`, using the notation for our framework, to describe a semaphore hub 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 specify and verify temporal properties of this scenario within our framework, such as *"every time* `s` *fires,*

`t` *will eventually fire in less than 3 time units"*. The verification uses UPPAAL, resorting to an intermediate DSL for the logic that hides locations and auxiliary variables and clocks.

This paper and the proposed framework address hubs that (i) go *beyond what is currently supported by VirtuosoNext*, by describing new hubs (with extra synchronisation and time restrictions, not part of VirtuosoNext), and (ii) allowing hubs to be connected to other hubs directly. The composition of hubs introduces the possibility of specifying complex interaction protocols, inspired by Reo's syntax [2] and real-time semantics [3, 4, 5]. Currently, without these complex protocols, the orchestration code must be intertwined with the tasks' behaviour.

In concrete, this paper provides the following contributions. Parts in bold denote new results regarding the associated conference publication [6]. An extended version of this document is published as a technical report [7].

- Specification of hubs interpreted as **timed** (hub) automata,
  - capturing hubs currently present in VirtuosoNext (without real time),
  - including hubs not present in VirtuosoNext (some **with real time**).
- Online tools (`http://arcatools.org/hubs`) to analyse hubs,
  - using a DSL to specify hubs built by composing simpler hubs,
  - **using a DSL to specify timed contracts of tasks' interactions,**
  - interpreting composed hubs as the composition of their **timed** automata (c.f. [7]),
  - generating graphs and composed automata with dynamic layouts,
  - **introducing a temporal logic focused on interactions,**
  - **generating UPPAAL specifications and logic formulas,**
  - **running UPPAAL to verify properties,** and
  - including other analysis of hubs.

The rest of this paper is organized as follows. Section 2 provides some context on how hubs coordinate tasks in VirtuosoNext, and how we can formally model existing and new hubs. Section 3 presents the software architecture and functionality. Section 4 introduces the verification tools, including timed contracts of tasks, the dynamic temporal logic, and the usage of UPPAAL to verify properties. Section 5 exemplifies how to verify the behaviour of a complex hub under different scenarios. Finally, Sections 6 and 7 discuss some related work and conclude with highlights and future directions, respectively.

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.

Our tools propose the analysis of the behaviour of these hubs, supporting a small specification language for tasks and hubs, and proposing a composition model for hubs with timed behaviour, not currently supported by VirtuosoNext. This section starts by giving a small overview of how tasks are built and composed in VirtuosoNext, followed by a more detailed description over existing hubs, and by an approach to specify more complex time-aware hubs than the ones supported by VirtuosoNext.

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 [8].

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

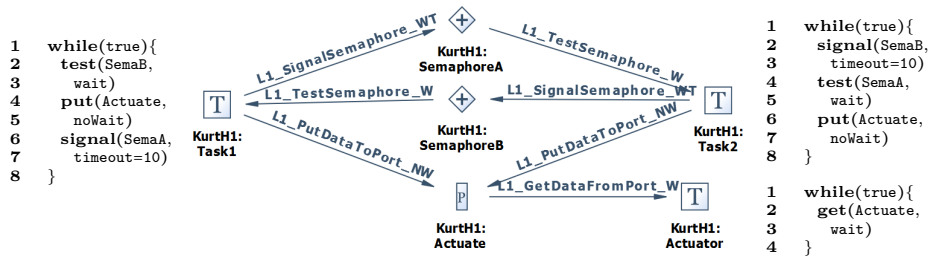

Figure 1: Example application in VirtuosoNext, whereby two tasks communicate with an actuator in a round robin sequence through two semaphores and a port.

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 hubs to request atomic access to a resource or hubs that act as gateways to peripheral hardware.

Any number of tasks can make *put* or *get* requests to a hub. Such requests are queued in waiting lists (at each corresponding hub) until they are served. Waiting lists are ordered by task priority – requests get served by following such an order. Requests can use different interaction semantics, which determine how a task waits on a request to succeed. There are three synchronous and one asynchronous interaction semantics in VirtuosoNext. Here we focus on the first three. These can be: *waiting* (`W`) – a task waits indefinitely until the request is served; *non-waiting* (`NW`) – either the request is served without delay or it 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.

In our tools we can write `task<t1>(W testB?, NW putAct!, 10 signalA!)` to denote a possible contract over the external behaviour of `Task1` in Fig. 1. This contract specifies that the task waits indefinitely to read (?) a value in its port *testB*, after which it tries to write (!) a value to its port *putAct* either succeeding without delay or failing. Finally, it tries to write a value to port *signalA* waiting at most 10 units of time to succeed or fail before it tries to read a value in *testB* again. We further discuss tasks in Section 4.1.

There are various hubs available, each with its predefined semantics [8]. Table 1 describes some of them and their *put* and *get* service request methods.

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.
 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`.
 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.

2.3. Beyond VirtuosoNext: Custom Complex Hubs

We propose an extended selection of hubs, not currently included in VirtuosoNext, to capture extra synchrony and time constraints. These include the ones in Table 2. The Drain hub ignores data values, forcing all participants to synchronise before proceeding; the Duplicator broadcast an input to all its outputs atomically, i.e., all outgoing ports must receive the input before the original sender can resume its execution; and the Timer (called P-Timer in the companion report [7]), parametrised by $t \in \mathcal{N}$, buffers a received value for t time units, and then sends it to its outgoing port.

More complex hubs can be built by plugging existing hubs together, also not currently supported by VirtuosoNext. For example, the composition  denotes a new hub that waits for a given time after receiving a value from its left port, and then synchronously sends it to both of the right ports. Fig. 2 describes a more complex architecture of a sequencer protocol than the one in Fig. 1, which we will use as a running example. Unlike in the sequencer in Fig. 1, the sequencing behaviour is captured by the hub (exogenous coordination), and it is not not scattered among the components (endogenous coordination), making it easier to analyse and adapt or maintain. I.e., tasks in the original architecture 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, $start_i$, and placing

Table 2: Examples of new Hubs not currently in VirtuosoNext

Hub	Waiting Lists for Service Requests
Drain*	$\texttt{put}_1, \texttt{put}_2$ – signals some data entering the ports. Both $\texttt{put}_1$ and $\texttt{put}_2$ must synchronize to succeed.
Duplicator	$\texttt{put}_1, \ldots, \texttt{put}_n$ – signals some data entering the port; and $\texttt{get}_1, \ldots, \texttt{get}_m$ – signals some data leaving the port. Exactly one `put` and all `get` must synchronize to succeed.
Timer	`set` – buffers some data and starts a timer, succeeding if the buffer is empty; and `test` – gets data from the buffer after the timer finishes.

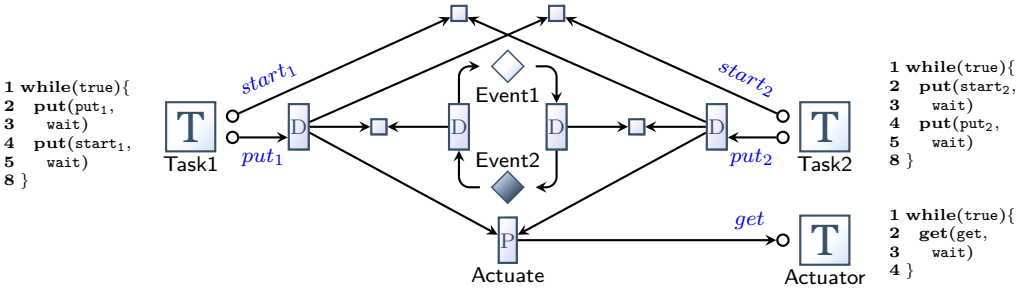

Figure 2: Alternative architecture for the sequencer protocol in Fig. 1.

a value, put_i, unaware of the coordination protocol.

2.4. Formal semantics in a nutshell

The formal semantics of hubs and their composition is given by Timed Hub Automata (THA), which are timed automata [9] based on the timed automata semantics of Reo connectors [10, 4, 5]. This formalisation is not covered in this paper, which focuses on the tools that analyse this behaviour, but can be found in the associated conference publication [6] (without time) and in the companion technical report [7] (with time). Here, we provide an informal description of these automata through examples.

As in timed automata, there is a notion of clock variables that capture the dense time that passes since they were last reset. Initially, all clocks are set to zero, and are incremented simultaneously. THA additionally supports multi-action transitions, meaning all actions execute simultaneously.

Figure 3: The composed THA for the running example in Fig. 2 (left), and the composed Timer and Duplicator example from section Section 2.3 (right).

Example: Custom Alternator. Fig. 3 (left) shows the THA that captures the behaviour specified by the architecture in Fig. 2. Initially the automaton is in location *L1*, and it can move to a new location *L2* by atomically performing actions from the three involved tasks (top transition), namely, put_1, *get*, and $start_2$. While doing so, a special variable associated to port *get*, is assigned with the value sent through port put_1 in $\widehat{get} \leftarrow \widehat{put_1}$. The remaining transition behaves similarly.

Example: Timer ⋈ Duplicator. Fig. 3 (right) shows the THA that captures the behaviour of the composed Timer and Duplicator from Section 2.3, when they are synchronised over the actions *test* and *put*. Initially, the automata is in location *idle*. Whenever the timer is set, the buffer is updated with the value sent through port *set*, namely $\widehat{set}$. In addition, this transition resets a clock $c \leftarrow 0$ before moving to a new location. This location has an invariant, $c \leq T$, i.e., a clock constraint that determines how much time the automaton can spend on such location, in this case, no more that T units of time for some specified $T \in \mathcal{N}$. The automaton waits exactly T time units—indicated by the clock constraint $c = T$ on the outgoing transition—after which it must be tested simultaneously by two tasks through ports get_1 and get_2. Both tasks will receive the stored data in the Timer Hub and the THA returns to the *idle* location.

3. Software Framework

3.1. Software Architecture

The software architecture is illustrated in Fig. 4. The tool is integrated into the ReoLive framework. This framework aggregates various tools, including the *Hubs* module, each being an independent open project on GitHub.

It provides support for generating an interactive website to use the tools, either in a standalone lightweight JavaScript version, or in a Client-Server version that enables the support of off-the-shelf applications from the browser. The off-the-shelf tools include the Uppaal real-time model checker used by the Hubs module to verify temporal properties of the hubs.

The Preo module provides the support to parse and interpret the specified hubs as Reo connectors [11]. These connectors can later be translated into a THA for further analysis by the Hubs module. The Hubs module provides the remaining functionality to compose, analyse, and verify hubs with Uppaal, which is described in the following section.

The modules and the framework are developed in Scala, an object-oriented programming language with functional features [12]. The Client-Server version is compiled into JavaScript using Scala.js[1] to run on the client side, and JVM binaries to run on the server side. The server is based on the Play Framework[2] for Scala. The lightweight and the client side version use the D3.js[3] library to build interactive graphics in JavaScript. 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.

3.2. Software Functionalities

We implemented a tool that *composes*, *simplifies*, *analyses*, and *verifies* THA, available to use online or download on `http://arcatools.org/hubs`. We organise the functionality by widgets, as depicted in Fig. 5. Our current implementation allows specifications of composed hubs and tasks using a textual representation based on Preo [13, 14] and Treo [15], by means of the following widgets: (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

[1] `https://www.scala-js.org`
[2] `https://www.playframework.com/`
[3] `https://d3js.org/`

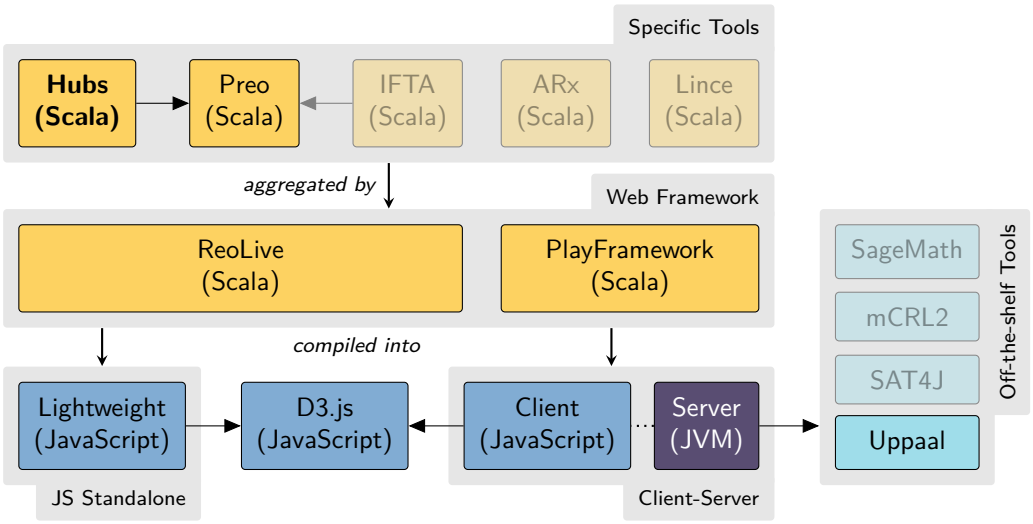

Figure 4: Software Architecture.

behavioural properties, relying on UPPAAL running on our servers, and their result ⑧ together with the associated UPPAAL models and formulas.

4. Verification tools

This section describes how tasks are abstracted and specified in our formal framework (Section 4.1), presents a temporal logics fine-tuned to THA to specify timed properties (Section 4.2), and describes an encoding of formulas and hubs into UPPAAL's temporal logic and timed automata, respectively (Section 4.3).

4.1. 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.

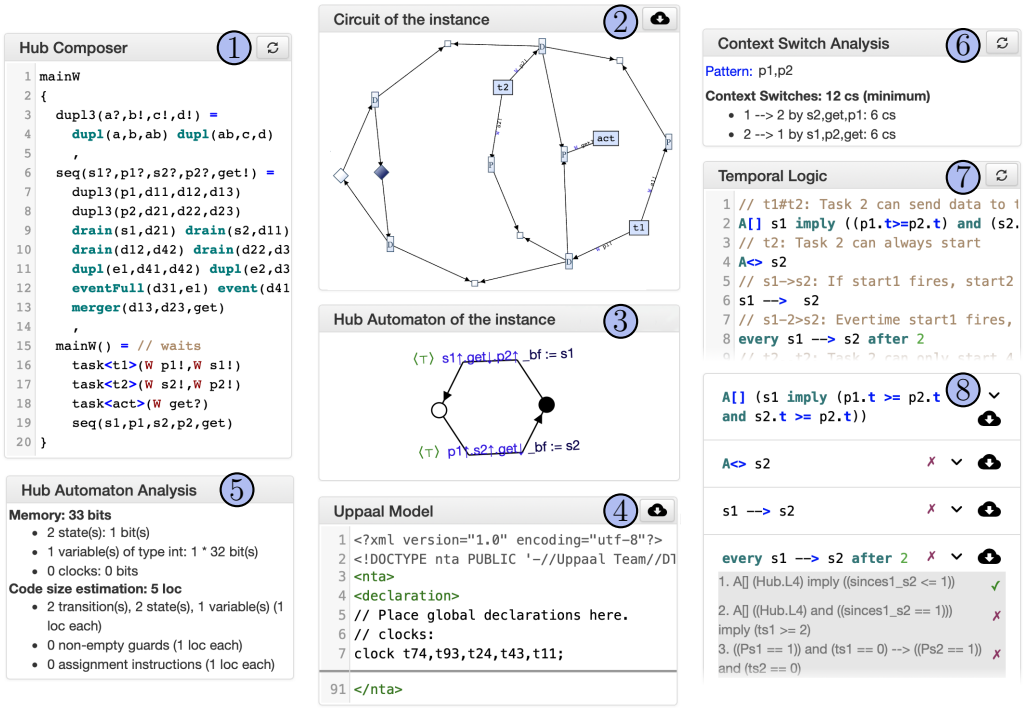

Figure 5: Screenshot of the widgets in the online analyser for VirtuosoNext's hubs.

Contracts for tasks can be specified by the following grammar.

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

This syntax has been briefly mentioned in Section 2.2. 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 or 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 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

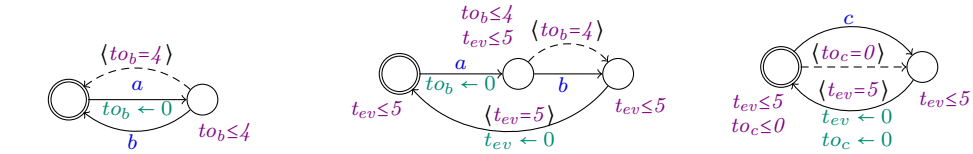

Figure 6: Timed hub automata of specific tasks.

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.

The three examples above produce the timed automata in Fig. 6, explained in more detail in the companion report [7]. These automata use clock variables t_{ev} to capture the time since the beginning of a round, and to_x, for each port x, to capture the time since x is ready to fire. Furthermore they use dashed arrows to denote lower priority transitions, based on Uppaal's notion of channel priority, not covered in our THA semantics.

4.2. 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, extended with new operators to describe the behaviour of the systems in terms of **actions**, i.e., on ports that fire, rather than locations. We propose a concrete syntax that closely follows that used by Uppaal's model checker, and define its semantics by formalising its satisfaction relation. Section 4.3 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 is evaluated over paths of the underlying transition system, while a state formula is evaluated over a single state of such system. The syntax and semantics of TCTL properties are formalised below.

Definition 1 (TCTL for THA). *A valid property over a THA consists of a* path formula π *given by the following grammar*

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

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

where $a, b \in \mathcal{P}$ *are ports,* $n \in \mathbb{N}$, $\boxtimes \in \{\Box, \Diamond\}$, $pred(\overline{x})$ *is a predicate over variables* x *used by the THA,* ρ *is an* a*-formula defined below, and* cc *is a clock constraint defined below using* $\boxdot \in \{<, \leq, ==, >, \geq\}$ *and* c *to range over clocks.*

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

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

Informally, *state properties* describe what must hold for a given state (which includes the time value assigned to clocks), and *path properties* describe what must hold while evolving the automaton. For example, $a.\texttt{done}$ holds if a has fired at least once, $a.\texttt{doing}$ holds if a was the last port to be fired, and $a\ \texttt{refiresAfterOrAt}\ 5$ holds in states where, if a fired before, then it cannot refire unless 5 units of time have passed. Regarding path properties, $\texttt{A}\boxtimes\ \psi$ holds if $\boxtimes\ \psi$ holds for all possible paths, while its `E` counterpart holds if $\boxtimes\ \psi$ 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 $\texttt{A}\,\Box\,(\psi_1\ \texttt{imply}\ (\texttt{A}\,\Diamond\ \psi_2))$.[4] The latter holds if, for all paths with a state that satisfies ψ_1, ψ_2 must be satisfied by one of the succeeding states; i.e., whenever ψ_1 holds, always eventually ψ_2 holds. Finally, `every` a `-->` b `after` 5 holds if, whenever a fires, b will fire after 5 or more time units without a firing again until b has fired.

The formal definition of satisfaction of a formula π for a given THA H and state s, written $H, s \vDash \pi$, can be found in the companion report [7]. This grammar is enriched with a special clock $a.\texttt{t}$ for each port a, denoting the time since a fired last time (or since the beginning of the execution), and with syntactic sugar for state formulas, summarised below. We write $\bigwedge$ to indicate the generalised `and` for multiple state formulas, and P to denote the set of all ports used by a given THA.

$$\begin{array}{rcl@{\qquad}rcl}
a & \triangleq & a.\texttt{doing and } a.\texttt{t} == 0 & \psi_1\ \texttt{imply}\ \psi_2 & \triangleq & \texttt{not}\ \psi_1\ \texttt{or}\ \psi_2 \\
\psi_1\ \texttt{or}\ \psi_2 & \triangleq & \texttt{not}\ (\texttt{not}\ \psi_1\ \texttt{and not}\ \psi_2) & a\ \texttt{refiresBefore}\ n & \triangleq & a.\texttt{t} < n \\
\texttt{nothing} & \triangleq & \bigwedge_{a \in P}\ \texttt{not}\ a.\texttt{doing} & a\ \texttt{refiresBeforeOrAt}\ n & \triangleq & a.\texttt{t} \leq n
\end{array}$$

4.3. Under the hood: verification via Uppaal

This subsection describes how we verify THA using Uppaal. More precisely, it describes informally how a THA is encoded as a timed automaton in

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

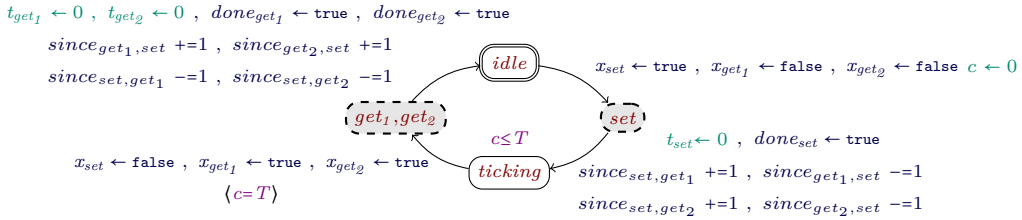

Figure 7: Encoded UPPAAL's automaton of the THA from Fig. 3; dashed locations are *committed*.

UPPAAL, and how TCTL formulas for THA are viewed in UPPAAL's TCTL, based on examples. The *automata encoding* introduces new data variables 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 the companion report [7].

4.3.1. Encoding Automata by Example

Recall the THA of the Timer ⋈ Duplicator hub depicted in Fig. 3. Its corresponding timed automaton in UPPAAL is depicted in Fig. 7, which introduces new *locations*, *clocks*, and *data variables*. These includes, for each port a, the clock t_a (to capture a.`t`) and variable $done_a$ (to capture a.`done`). Other added variables and locations are described below.

Locations that represent actions being fired are depicted with dashed lines, and are associated to sets of ports that triggered them. These are marked as *committed locations* in UPPAAL, in which time is not allowed to proceed. Hence, to know if *set* has just been fired, one can check if the automaton is in any of these special committed locations associated to the *set* port.

Data variables (x and *since*) Every port a yields a variable x_a, set to `true` when a was fired in the last set of fired ports. Every pair of different ports (a, b) yields a variable $since_{a,b}$, with $0 \leq since_{a,b} \leq 2$ (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 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 variables 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.

Priority: Recall that tasks are modelled as timed automata with a notion of priority (Section 4.1). This priority is meant only to prevent ports from discarding data and timing out 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 a set of dummy channels $prio_p$ that can always be fired[5], 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 automata, 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.

4.3.2. Encoding Formulas by Example

The UPPAAL[6] model checker supports a subset of TCTL formulas for timed automata [16], which we took into account when proposing the logic for THA. 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.

[5] This is technically achieved using a broadcast channel in UPPAAL.

[6] http://www.uppaal.org/

Table 3: Examples of encodings of THA TCTL formulas into UPPAAL.

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

To refer to the committed locations introduced in Section 4.3.1 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 automaton that has no committed states, and by adding new variable assignments that are never read.

5. Example: Verifying the sequencer protocol

Recall our running example illustrated in Fig. 2 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 Table 4 (left column). Notice that the first scenario corresponds to the protocol in Fig. 2. On same table we list 5

Table 4: Verification of the sequencer hub under different scenarios.

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

`task<T1>(W put1!, W st1! )`
`task<T2>(W st2! , W put2!)`
`task<Ac>(W get?)`
	✓	✗	✗	✗	✗

`task<T1>(W put1!, W st1! ) every 3`
`task<T2>(W st2! , W put2!) every 3`
`task<Ac>(W get?)`
	✓	✓	✓	✗	✓

`task<T1>(NW put1!, NW st1! ) every 3`
`task<T2>(NW st2! , NW put2!) every 3`
`task<Ac>(W get?)`
	✓	✓	✓	✓	✓

`task<T1>(3 put1!, 3 st1! ) every 6`
`task<T2>(3 st2! , 3 put2!) every 6`
`task<Ac>(W get?)`
	✓	✓	✓	✗	✓

`task<T1>(NW put1!, 3 st1! ) every 2`
`task<T2>(W st2! , 3 put2!) every 3`
`task<Ac>(W get?)`
	✓	✓	✗	✗	✓

different properties that we find of relevance, and whether these are satisfied under each scenario (right column). These properties are described below, together with a discussion regarding their satisfaction on the scenarios.

$\psi_{t1\#t2} = \{\texttt{A}\,\square\ start_1\ \texttt{imply}\ ((put_1.\texttt{t} \geq put_2.\texttt{t})\ \texttt{and}\ (start_2.\texttt{t} \geq put_2.\texttt{t}))\}$

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.

$\psi_{t2} = \{\texttt{A}\,\diamond\ start_2\}$

Task 2 must start eventually. This liveness property checks if Task 2 must run. Only the first scenario fails to satisfy this property, because "`W st!`" is allowed to wait an unbounded amount of time. Hence, there is no guarantee it will run when it decides to wait *forever*. The other scenarios use a "`every`" construct that bounds the waiting time.

$\psi_{s1\to s2} = \{start_1\ \texttt{-->}\ start_2\}$

If $start_1$ fires, $start_2$ must eventually fire. This liveness property describes continuous progress. The first scenario does not satisfy it because it can wait forever, and the last one because it **deadlocks**. The

deadlock occurs after `T1` finishes the 1st round firing both ports, and it fails to fire put_1 in the 2nd round. Both `T1` and `T2` wait to fire *start* in their 2nd round, and time cannot pass at the end of the `T1`'s round.

$\psi_{s1 \xrightarrow{2} s2} = \{$ **every** $start_1$ **-->** $start_2$ **after** 2 $\}$

Everytime $start_1$ *fires,* $start_2$ *must eventually fire before* $start_1$ *again, and wait at least 2 time units before firing* $start_2$. This is a variation of the previous property with a periodicity. All but the 3rd scenario fail to satisfy this property: the 1st scenario fails because rounds can be faster than 2; the 2nd and 4th fail because $start_1$ can be executed at the end of a round, and put_2 at the beginning of the following round; the last scenario fails for the same reason $\psi_{s1 \to s2}$ does.

$\psi_{\leq 9} = \{$ **A** $\Box$ $start_2$ **refiresBeforeOrAt** 9 $\}$

Task 2 starts within 9 time units after finishing a previous round. Only the first scenario fails, since it can take an infinite amount of time between two fires of $start_2$. The 2nd scenario can take up to 6 time units between fires of $start_2$, the 4th can take up to 9 time units when $start_2$ fires at the beginning of a round, and right before timing out in the follow up round (6+3 time units).

Observe 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.

6. Related work

The global architecture of VirtuosoNext RTOS, including the interaction with hubs, has been formally analysed using TLA+ by Verhulst et al. [1], focusing on untimed properties regarding how hubs are implemented within VirtuosoNext. Recently, we proposed an approach to formalise hubs through hub automata [6], focused on the interactions, aiming at the analysis of hubs built compositionally. Here, we use hub automata extended with time [7], proposing a dynamic logic to express temporal properties focusing on ports.

Timed Hub Automata is inspired by existing automata-based models for Reo [2, 17, 3, 5], involving data, variables, and time. 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. Unlike these approaches, we introduce a notion of sequential and parallel updates, and facilitate the verification process for the end user, 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.

Formal analysis of RTOS are more typically focused on the scheduler, which is not the focus of this work. The following are examples of relevant scheduling analysis. Ha et al. [18] used theorem provers to analyse schedulers for avionics software. Carnevali et al. [19] used 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] analysed and model checked 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] proposed a framework to synthesise optimised schedulers that consider delays introduced by interaction between tasks, interpreting scheduling as a game that requires minimising the time between subsequent context switches.

7. Conclusions

This article presents a toolset to construct 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 (http://arcatools.org/hubs).

The tools benefits both users of VirtuosoNext and Altreonic's developers. The former can experiment how existing hubs behave in different timed scenarios; while the latter can use it to help designing new custom-made hubs, and potentially incorporate them into a future version of VirtuosoNext.

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 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. [13, 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); 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); and by the FCT within project ECSEL/0016/2019 and the ECSEL Joint Undertaking (JU) under grant agreement No 876852. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey.

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] 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. doi:10.1016/j.scico.2005.10.008.

[3] 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. doi:10.1007/s10270-006-0009-9.

[4] 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.

[5] 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.

[6] 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. doi:10.1007/978-3-030-22397-7_15.

[7] G. Cledou, J. Proença, B. H. C. Sputh, E. Verhulst, Verification of Real-Time Coordination in VirtuosoNext (extended version) (May 2020). doi:10.5281/zenodo.3818020.

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

[9] J. Bengtsson, W. Yi, Timed Automata: Semantics, Algorithms and Tools, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 87–124. doi:10.1007/978-3-540-27755-2_3.

[10] S. Meng, F. Arbab, On resource-sensitive timed component connectors, in: M. M. Bonsangue, E. B. Johnsen (Eds.), Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings, Vol. 4468 of Lecture Notes in Computer Science, Springer, 2007, pp. 301–316. doi:10.1007/978-3-540-72952-5_19.

[11] F. Arbab, Reo: a channel-based coordination model for component composition, Math. Struct. Comput. Sci. 14 (3) (2004) 329–366. doi:10.1017/S0960129504004153.

[12] M. Odersky, L. Spoon, B. Venners, Programming in scala, Artima Inc, 2008.

[13] 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.

[14] J. Proença, A. Madeira, Taming hierarchical connectors, in: H. Hojjat, M. Massink (Eds.), Fundamentals of Software Engineering - 8th International Conference, FSEN 2019, Tehran, Iran, May 1-3, 2019, Revised Selected Papers, Vol. 11761 of Lecture Notes in Computer Science, Springer, 2019, pp. 186–193. doi:10.1007/978-3-030-31517-7_13.

[15] K. Dokter, F. Arbab, Treo: Textual syntax for reo connectors, in: S. Bliudze, S. Bensalem (Eds.), Proceedings of the 1st International Workshop on Methods and Tools for Rigorous System Design, MeTRiD@ETAPS 2018, Thessaloniki, Greece, 15th April 2018, Vol. 272 of EPTCS, 2018, pp. 121–135. doi:10.4204/EPTCS.272.10.

[16] 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.

[17] 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.

[18] V. Ha, M. Rangarajan, D. D. Cofer, H. Rueß, B. Dutertre, Feature-based decomposition of inductive proofs applied to real-time avionics software: An experience report, in: A. Finkelstein, J. Estublier, D. S. Rosenblum (Eds.), 26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom, IEEE Computer Society, 2004, pp. 304–313. doi:10.1109/ICSE.2004.1317453.

[19] L. Carnevali, G. Lipari, A. Pinzuti, E. Vicario, A formal approach to design and verification of two-level hierarchical scheduling systems, in: A. B. Romanovsky, T. Vardanega (Eds.), Reliable Software Technologies - Ada-Europe 2011 - 16th Ada-Europe International Conference on Reliable Software Technologies, Edinburgh, UK, June 20-24, 2011. Proceedings, Vol. 6652 of Lecture Notes in Computer Science, Springer, 2011, pp. 118–131. doi:10.1007/978-3-642-21338-0_9.

[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. Jongmans, F. Arbab, Scheduling games for concurrent systems, in: A. Lluch-Lafuente, J. Proença (Eds.), Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings, Vol. 9686 of Lecture Notes in Computer Science, Springer, 2016, pp. 84–100. doi:10.1007/978-3-319-39519-7_6.

[22] N. Kokash, C. Krause, E. P. de Vink, Reo + mcrl2: A framework for model-checking dataflow in service compositions, Formal Asp. Comput. 24 (2) (2012) 187–216. doi:10.1007/s00165-011-0191-6.